The Java™ Language Specification

Second Edition

The Java™ Series

Lisa Friendly, Series Editor
Bill Joy, Technical Advisor

The Java™ Programming Language
Ken Arnold and James Gosling
ISBN 0-201-63455-4

The Java™ Language Specification, Second Edition
James Gosling, Bill Joy, Guy Steele and Gilad Bracha
ISBN 0-201-31008-2

The Java™ Virtual Machine Specification, Second Edition
Tim Lindholm and Frank Yellin
ISBN 0-201-63452-X

The Java™ Application Programming Interface,
Volume 1: Core Packages
James Gosling, Frank Yellin, and the Java Team
ISBN 0-201-43294-3

The Java™ Application Programming Interface,
Volume 2: Window Toolkit and Applets
James Gosling, Frank Yellin, and the Java Team
ISBN 0-201-63459-7

The Java™ Tutorial: Object-Oriented Programming for the Internet
Mary Campione and Kathy Walrath
ISBN 0-201-63454-6

The Java™ Class Libraries: An Annotated Reference
Patrick Chan and Rosanna Lee
ISBN 0-201-63458-9

The Java™ FAQ: Frequently Asked Questions
Jonni Kanerva
ISBN 0-201-63456-2

The Java™ Language Specification

Second Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha

ADDISON-WESLEY
An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Credits and permissions for quoted material appear in a separate section on page 519.

Table of Contents

CHAPTER 1

Introduction

If I have seen further it is by standing upon the shoulders of Giants.
—Sir Isaac Newton

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 c*ompile-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 a bytecoded instruction set and binary format defined in *The Java Virtual Machine Specification, Second Edition* (Addison-Wesley, 1999).

This Java Language 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. Java supports the writing of Unicode characters on systems that support only ASCII.

Chapter 4 describes types, values, and variables. These types are the 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 standard class `Object`, which is the (single) root of the class hierarchy. A predefined `String` class supports Unicode character strings. Standard classes exist for wrapping primitive values inside of objects.

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. 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 implementa-

tion of a type from its users and those who extend it. Standard 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 namespace, and the Internet domain name system can 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.

Although the language does not include parameterized classes, the semantics of arrays are those of a parameterized class with some syntactic sugar. Like the programming language Beta, the Java programming language uses a run-time type check when storing references in arrays to ensure complete type safety.

Chapter 9 describes interface types, which declare a set of abstract methods 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.

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 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` expressions in control-flow statements, and does not convert types to `boolean` implicitly, 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.

The book concludes with an index, credits for quotations used in the book, and a colophon describing how the book was created.

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 Sun workstation, 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 book we refer to classes and interfaces drawn from the Java and Java 2 platforms. Whenever we refer to a class or interface which is not defined in an example in this book using a single identifier *N*, the intended reference is to the class or interface `java.lang.`*N*. We use the fully qualified name for classes from packages other than `java.lang`.

1.3 Relationship to Predefined Classes and Interfaces

As noted above, this specification often refers to classes of the Java and Java 2 platforms. In particular, some classes have a special relationship with the language. Examples include classes such as `Object`, `Class`, `ClassLoader`, `String` and `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 platform specification for such detailed API specifications.

Thus this document does not describe reflection in any detail. Many linguistic constructs have analogs 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. See also `http://www.cambridge.apple.com`.

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.

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++ Progamming Language*, 2nd ed. Addison-Wesley, Reading, Massachusetts, 1991, reprinted with corrections January 1994, ISBN 0-201-53992-6.

Unicode Consortium, The. *The Unicode Standard: Worldwide Character Encoding*, Version 1.0, Volume 1, ISBN 0-201-56788-1, and Volume 2, ISBN 0-201-60845-6. Updates and additions necessary to bring the Unicode Standard up to version 1.1 may be found at `http://www.unicode.org.`

Unicode Consortium, The. *The Unicode Standard, Version 2.0*, ISBN 0-201-48345-9. Updates and additions necessary to bring the Unicode Standard up to version 2.1 may be found at `http://www.unicode.org.`

CHAPTER 2

Grammars

Grammar, which knows how to control even kings . . .
—Molière, *Les Femmes Savantes* (1672), Act II, scene vi

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 (§3). 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

The *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 Java 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.

For example, the syntactic definition:

IfThenStatement:
`if (` *Expression* `)` *Statement*

states that the nonterminal *IfThenStatement* represents the token `if`, followed by a left parenthesis token, followed by an *Expression*, followed by a right parenthesis token, followed by a *Statement*. As another example, the syntactic definition:

ArgumentList:
Argument
ArgumentList `,` *Argument*

states that an *ArgumentList* may represent either a single *Argument* or an *ArgumentList*, followed by a comma, followed by an *Argument*. This definition of *ArgumentList* is *recursive*, that is to say, it is defined in terms of itself. The result is that an *ArgumentList* may contain any positive number of arguments. Such recursive definitions of nonterminals are common.

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.

This means that:

BreakStatement:
`break` *Identifier*^{opt} `;`

is a convenient abbreviation for:

BreakStatement:
`break ;`
`break` *Identifier* `;`

and that:

ForStatement:
`for (` *ForInit*^{opt} `;` *Expression*^{opt} `;` *ForUpdate*^{opt} `)` *Statement*

is a convenient abbreviation for:

ForStatement:
`for ( ;` *Expression*^{opt} `;` *ForUpdate*^{opt} `)` *Statement*
`for (` *ForInit* `;` *Expression*^{opt} `;` *ForUpdate*^{opt} `)` *Statement*

which in turn is an abbreviation for:

ForStatement:
`for ( ; ;` *ForUpdate*^{opt} `)` *Statement*
`for ( ;` *Expression* `;` *ForUpdate*^{opt} `)` *Statement*
`for (` *ForInit* `; ;` *ForUpdate*^{opt} `)` *Statement*
`for (` *ForInit* `;` *Expression* `;` *ForUpdate*^{opt} `)` *Statement*

which in turn is an abbreviation for:

ForStatement:
`for ( ; ; )` *Statement*
`for ( ; ;` *ForUpdate* `)` *Statement*
`for ( ;` *Expression* `; )` *Statement*
`for ( ;` *Expression* `;` *ForUpdate* `)` *Statement*
`for (` *ForInit* `; ; )` *Statement*
`for (` *ForInit* `; ;` *ForUpdate* `)` *Statement*
`for (` *ForInit* `;` *Expression* `; )` *Statement*
`for (` *ForInit* `;` *Expression* `;` *ForUpdate* `)` *Statement*

so the nonterminal *ForStatement* actually has eight alternative right-hand sides.

A very long right-hand side may be continued on a second line by substantially indenting this second line, as in:

ConstructorDeclaration:
ConstructorModifiers^{opt} *ConstructorDeclarator*
Throws^{opt} *ConstructorBody*

which defines one right-hand side for the nonterminal *ConstructorDeclaration*.

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.

For example, the lexical grammar contains the production:

ZeroToThree: one of
`0 1 2 3`

which is merely a convenient abbreviation for:

ZeroToThree:
`0`
`1`
`2`
`3`

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.

Thus, the definition:

BooleanLiteral: one of
`true false`

in a lexical grammar production is shorthand for:

BooleanLiteral:
`t r u e`
`f a l s e`

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, as in the productions for *InputCharacter* (§3.4) and *Identifier* (§3.8):

InputCharacter:
UnicodeInputCharacter but not CR or LF

Identifier:
    IdentifierName but not a *Keyword* or *BooleanLiteral* or *NullLiteral*

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:

RawInputCharacter:
    any Unicode character

CHAPTER 3

Lexical Structure

Lexicographer: A writer of dictionaries, a harmless drudge.
—Samuel Johnson, *Dictionary* (1755)

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

Java programs are written using the Unicode character set, version 2.0. Information about this encoding may be found at:

`http://www.unicode.org` and `ftp://unicode.org`

Versions of the Java programming language prior to 1.1 used Unicode version 1.1.5 (see *The Unicode Standard: Worldwide Character Encoding* (§1.4) and updates).

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 Java 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 Java 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 `\u`*xxxx*, where *xxxx* is a hexadecimal value, represents the Unicode character 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 Java 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) for Java.

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

Java implementations first recognize *Unicode escapes* in their input, translating the ASCII characters `\u` followed by four hexadecimal digits to the Unicode character with the indicated hexadecimal value, and passing all other characters unchanged. 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. For example, the raw input `"\\u2297=\u2297"` results in the eleven characters " \ \ u 2 2 9 7 = ⊗ " (`\u2297` is the Unicode encoding of the character "⊗").

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. For example, the raw input `\u005cu005a` results in the six characters \ u 0 0 5 a, because `005c` is the Unicode value for \. It does not result in the character Z, which is Unicode character `005a`, because the \ that resulted from the `\u005c` is not interpreted as the start of a further Unicode escape.

Java specifies a standard way of transforming a Unicode Java program 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, \u*xxxx* becomes \uu*xxxx*—while simultaneously converting non-ASCII characters in the source text to a \u*xxxx* escape containing a single u. 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.

Java systems should use the \u*xxxx* notation as an output format to display Unicode characters when a suitable font is not available.

3.4 Line Terminators

Java implementations next divide the sequence of Unicode input characters into lines by recognizing *line terminators*. This definition of lines determines the line

numbers produced by a Java compiler or other Java system component. It also specifies the termination of the // form of a comment (§3.7).

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. 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 Java syntactic grammar (§2.3).

This process is specified by the following productions:

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*. For example, in this simple piece of code:

```
class Empty {
}
```

we say that the } token is to the right of the { token, even though it appears, in this two-dimensional representation on paper, downward and to the left of the { token. This convention about the use of the words left and right allows us to speak, for example, of the right-hand operand of a binary operator or of the left-hand side of an assignment.

3.6 White Space

White space is defined as the ASCII space, horizontal tab, and form feed characters, as well as line terminators (§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 three 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*	A *end-of-line comment*: all the text from the ASCII characters `//` to the end of the line is ignored (as in C++).
`/**` *documentation* `*/`	A *documentation comment*: the text enclosed by the ASCII characters `/**` and `*/` can be processed by a separate tool to prepare automatically generated documentation of the following class, interface, constructor, or member (method or field) declaration.

These comments are formally specified by the following productions:

Comment:
TraditionalComment
EndOfLineComment
DocumentationComment

TraditionalComment:
`/` `*` *NotStar CommentTail*

EndOfLineComment:
`/` `/` $CharactersInLine_{opt}$ *LineTerminator*

DocumentationComment:
`/` `*` `*` *CommentTailStar*

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 `/**`.

As a result, the text:

```
/* this comment /* // /** ends here: */
```

is a single complete comment.

The lexical grammar implies that comments do not occur within character literals (§3.10.4) or string literals (§3.10.5).

Note that `/**/` is considered to be a documentation comment, while `/* */` (with a space between the asterisks) is a traditional comment.

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` returns `true`. A Java letter-or-digit is a character for which the method `Character.isJavaIdentifierPart` returns `true`.

The Java letters include uppercase and lowercase ASCII Latin letters A–Z (\u0041–\u005a), and a–z (\u0061–\u007a), and, for historical reasons, the ASCII underscore (_, or \u005f) and dollar sign ($, or \u0024). The $ character should be used only in mechanically generated source code or, rarely, to access preexisting names on legacy systems.

The Java digits include the ASCII digits 0-9 (\u0030–\u0039).

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. For example, the identifiers consisting of the single letters LATIN CAPITAL LETTER A (A, \u0041), LATIN SMALL LETTER A (a, \u0061), GREEK CAPITAL LETTER ALPHA (A, \u0391), and CYRILLIC SMALL LETTER A (a, \u0430) are all different.

Unicode composite characters are different from the decomposed characters. For example, a LATIN CAPITAL LETTER A ACUTE (Á, \u00c1) could be considered to be the same as a LATIN CAPITAL LETTER A (A, \u0041) immediately followed by a NON-SPACING ACUTE (´, \u0301) when sorting, but these are different in Java identifiers. See *The Unicode Standard*, Volume 1, pages 412ff for details about decomposition, and see pages 626–627 of that work for details about sorting.

Examples of identifiers are:

```
String    i3    αρετη    MAX_VALUE    isLetterOrDigit
```

3.9 Keywords

The following character sequences, formed from ASCII letters, are reserved for use as *keywords* and cannot be used as identifiers (§3.8):

Keyword: one of

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

The keywords const and goto are reserved, even though they are not currently used. This may allow a Java compiler to produce better error messages if these C++ keywords incorrectly appear in programs.

While `true` and `false` might appear to be keywords, they are technically Boolean literals (§3.10.3). Similarly, while `null` might appear to be a keyword, it is technically the null literal (§3.10.7).

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

See §4.2.1 for a general discussion of the integer types and values.

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). The suffix `L` is preferred, because the letter `l` (ell) is often hard to distinguish from the digit `1` (one).

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 X` *HexDigits*

HexDigits:
HexDigit
HexDigit HexDigits

The following production from §3.3 is repeated here for clarity:

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
OctalDigit OctalDigits

OctalDigit: one of
`0 1 2 3 4 5 6 7`

Note that octal numerals always consist of two or more digits; `0` is always considered to be a decimal numeral—not that it matters much in practice, for the numerals `0`, `00`, and `0x0` all represent exactly the same integer value.

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`.

A compile-time error occurs 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.

Examples of `int` literals:

```
0     2     0372     0xDadaCafe     1996   0x00FF00FF
```

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 `0x7fffffffffffffffL` and `07777777777777777777L`, respectively, which equal `9223372036854775807L` ($2^{63}-1$). The literals `0x8000000000000000L` and `01000000000000000000000L` are the most negative `long` hexadecimal and octal literals, respectively. Each has the decimal value `-9223372036854775808L` (-2^{63}). The hexadecimal and octal literals `0xffffffffffffffffL` and `01777777777777777777777L`, respectively, represent the decimal value `-1L`.

A compile-time error occurs 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.

Examples of `long` literals:

```
0l     0777L     0x100000000L     2147483648L     0xC0B0L
```

3.10.2 Floating-Point Literals

See §4.2.3 for a general discussion of the floating-point types and values.

A *floating-point literal* has the following parts: a whole-number part, a decimal point (represented by an ASCII period character), a fractional part, an exponent, and a type suffix. The exponent, if present, is indicated by the ASCII letter `e` or `E` followed by an optionally signed integer.

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.

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:
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`

The types `float` and `double` are IEEE 754 32-bit single-precision and 64-bit double-precision binary floating-point values, respectively.

The details of proper input conversion from a Unicode string representation of a floating-point number to the internal IEEE 754 binary floating-point representation are described for the methods `valueOf` of class `Float` and class `Double` of the package `java.lang`.

The largest positive finite `float` literal is `3.40282347e+38f`. The smallest positive finite nonzero literal of type `float` is `1.40239846e-45f`. The largest positive finite `double` literal is `1.79769313486231570e+308`. The smallest positive finite nonzero literal of type `double` is `4.94065645841246544e-324`.

A compile-time error occurs if a nonzero floating-point literal is too large, so that on rounded conversion to its internal representation it becomes an IEEE 754 infinity. A program can represent infinities without producing a compile-time error by using constant expressions such as `1f/0f` or `-1d/0d` or by using the predefined constants `POSITIVE_INFINITY` and `NEGATIVE_INFINITY` of the classes `Float` and `Double`.

A compile-time error occurs if a nonzero floating-point literal is too small, so that, on rounded conversion to its internal representation, it becomes a zero. A compile-time error does not occur if a nonzero floating-point literal has a small value that, on rounded conversion to its internal representation, becomes a nonzero denormalized number.

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

Examples of `float` literals:

```
1e1f   2.f   .3f   0f   3.14f   6.022137e+23f
```

Examples of `double` literals:

```
1e1   2.   .3   0.0   3.14   1e-9d   1e137
```

There is no provision for expressing floating-point literals in other than decimal radix. However, method `intBitsToFloat` of class `Float` and method `longBitsToDouble` of class `Double` provide a way to express floating-point values in terms of hexadecimal or octal integer literals. For example, the value of:

```
Double.longBitsToDouble(0x400921FB54442D18L)
```

is equal to the value of `Math.PI`.

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, enclosed in ASCII single quotes. (The single-quote, or apostrophe, character is `\u0027`.)

A character literal is always of type `char`.

CharacterLiteral:
`'` *SingleCharacter* `'`
`'` *EscapeSequence* `'`

SingleCharacter:
InputCharacter but not `'` or `\`

The escape sequences are described in §3.10.6.

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 `'`.

The following are examples of `char` literals:

```
'a'
'%'
'\t'
'\\'
'\''
'\u03a9'
'\uFFFF'
'\177'
'Ω'
'⊗'
```

Because Unicode escapes are processed very early, it is not correct to write `'\u000a'` for a character literal whose value is linefeed (LF); the Unicode escape `\u000a` is transformed into an actual linefeed in translation step 1 (§3.3) and the linefeed becomes a *LineTerminator* in step 2 (§3.4), and so the character literal is not valid in step 3. Instead, one should use the escape sequence `'\n'` (§3.10.6). Similarly, it is not correct to write `'\u000d'` for a character literal whose value is carriage return (CR). Instead, use `'\r'`.

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. Each character may be represented by an escape sequence.

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

The escape sequences are described in §3.10.6.

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).

The following are examples of string literals:

```
""                      // the empty string
"\""                    // a string containing " alone
"This is a string"      // a string containing 16 characters

"This is a " +          // actually a string-valued constant expression,
   "two-line string"    //       formed from two string literals
```

Because Unicode escapes are processed very early, it is not correct to write `"\u000a"` for a string literal containing a single linefeed (LF); the Unicode escape `\u000a` is transformed into an actual linefeed in translation step 1 (§3.3) and the linefeed becomes a *LineTerminator* in step 2 (§3.4), and so the string literal is not valid in step 3. Instead, one should write `"\n"` (§3.10.6). Similarly, it is not correct

to write "\u000d" for a string literal containing a single carriage return (CR). Instead use "\r".

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 .

Thus, the test program consisting of the compilation unit (§7.3):

```
package testPackage;

class Test {
   public static void main(String[] args) {
      String hello = "Hello", lo = "lo";
      System.out.print((hello == "Hello") + " ");
      System.out.print((Other.hello == hello) + " ");
      System.out.print((other.Other.hello == hello) + " ");
      System.out.print((hello == ("Hel"+"lo")) + " ");
      System.out.print((hello == ("Hel"+lo)) + " ");
      System.out.println(hello == ("Hel"+lo).intern());
   }
}

class Other { static String hello = "Hello"; }
```

and the compilation unit:

```
package other;

public class Other { static String hello = "Hello"; }
```

produces the output:

```
true true true true false true
```

This example illustrates six points:

- Literal strings within the same class (§8) in the same package (§7) represent references to the same String object (§4.3.1).
- Literal strings within different classes in the same package represent references to the same String object.
- Literal strings within different classes in different packages likewise represent references to the same String object.
- Strings computed by constant expressions (§15.28) are computed at compile time and then treated as if they were literals.
- Strings computed at run time are newly created and therefore distinct.
- The result of explicitly interning a computed string is the same string as any pre-existing literal string with the same contents.

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

The following nine ASCII characters are the Java *separators* (punctuators):

Separator: one of

```
(    )    {    }    [    ]    ;    ,    .
```

3.12 Operators

The following 37 tokens are the Java *operators*, formed from ASCII characters:

Operator: one of

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

Give her no token but stones; for she's as hard as steel.
—William Shakespeare, *Two Gentlemen of Verona*, Act I, scene i

These lords are visited; you are not free;
For the Lord's tokens on you do I see.
—William Shakespeare, *Love's Labour's Lost*, Act V, scene ii

Thou, thou, Lysander, thou hast given her rhymes,
And interchanged love-tokens with my child.
—William Shakespeare, *A Midsummer Night's Dream*, Act I, scene i

Here is a letter from Queen Hecuba,
A token from her daughter . . .
—William Shakespeare, *Troilus and Cressida*, Act V, scene i

Are there no other tokens . . . ?
—William Shakespeare, *Measure for Measure*, Act IV, scene i

Hush, my darling, don't fear, my darling, the lion sleeps tonight.
—Luigi Creatore, George David Weiss, and Hugo E. Peretti

CHAPTER 4

Types, Values, and Variables

I send no agent or medium,
offer no representative of value,
but offer the value itself.
—Walt Whitman, *Carol of Occupations* (1855),
in *Leaves of Grass*

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 (§4.5) 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).

Names of types are used (§4.4) in declarations, in casts, in class instance creation expressions, in array creation expressions, in class literals, and in `instanceof` operator expressions.

A variable (§4.5) is a storage location. A variable of a primitive type always holds a value of that exact 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 assignable (§5.2) to type *T*. A variable of type `Object` can hold a null reference or a reference to any object, whether class instance or array.

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. In practice, the programmer can ignore the null type and just pretend that `null` is merely a special literal that can be of 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.

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 Unicode characters.

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 – (§15.18.2)
  - The increment operator ++, both prefix (§15.15.1) and postfix (§15.14.1)
  - The decrement operator --, both prefix (§15.15.2) and postfix (§15.14.2)

  - 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 `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 (§5.1.4) 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. The only numeric operators that can throw an exception (§11) 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.

The example:

```
class Test {
   public static void main(String[] args) {
      int i = 1000000;
      System.out.println(i * i);
      long l = i;
      System.out.println(l * l);
      System.out.println(20296 / (l - i));
   }
}
```

produces the output:

```
-727379968
1000000000000
```

and then encounters an `ArithmeticException` in the division by `l - i`, because `l - i` is zero. The first multiplication is performed in 32-bit precision, whereas the

second multiplication is a `long` multiplication. The value `-727379968` is the decimal value of the low 32 bits of the mathematical result, `1000000000000`, which is a value too large for type `int`.

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 a special *Not-a-Number* (hereafter abbreviated NaN). The 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` (§5.1.8, §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.

Parameter	float	float extended-exponent	double	double extended-exponent
N	24	24	53	53
K	8	≥ 11	11	≥ 15
E^{max}	+127	≥ +1023	+1023	≥ +16383
E^{min}	−126	≤ −1022	−1022	≤ −16382

Table 4.1 Floating-point value set parameters

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 the five values positive zero, negative zero, positive infinity, negative infinity, and NaN.

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, except that there is only one NaN value (IEEE 754 specifies $2^{24} - 2$ distinct NaN values). 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, except that there is only one NaN value (IEEE 754 specifies $2^{53} - 2$ distinct NaN values). 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 per-

missible 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.

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 (§15.14.1)
  - The decrement operator `--`, both prefix (§15.15.2) and postfix (§15.14.2)
- 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 at its result the format's value closest to and no greater in magnitude than the infinitely precise result.

Floating-point operators produce no exceptions (§11). 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.

The example program:

```
class Test {
   public static void main(String[] args) {
      // An example of overflow:
      double d = 1e308;
      System.out.print("overflow produces infinity: ");
      System.out.println(d + "*10==" + d*10);
      // An example of gradual underflow:
      d = 1e-305 * Math.PI;
      System.out.print("gradual underflow: " + d + "\n   ");
      for (int i = 0; i < 4; i++)
         System.out.print(" " + (d /= 100000));
      System.out.println();
      // An example of NaN:
      System.out.print("0.0/0.0 is Not-a-Number: ");
      d = 0.0/0.0;
      System.out.println(d);
      // An example of inexact results and rounding:
      System.out.print("inexact results with float:");
      for (int i = 0; i < 100; i++) {
         float z = 1.0f / i;
         if (z * i != 1.0f)
            System.out.print(" " + i);
      }
      System.out.println();
      // Another example of inexact results and rounding:
      System.out.print("inexact results with double:");
      for (int i = 0; i < 100; i++) {
         double z = 1.0 / i;
         if (z * i != 1.0)
            System.out.print(" " + i);
      }
      System.out.println();
      // An example of cast to integer rounding:
      System.out.print("cast to int rounds toward 0: ");
      d = 12345.6;
      System.out.println((int)d + " " + (int)(-d));
   }
}
```

produces the output:

```
overflow produces infinity: 1.0e+308*10==Infinity
gradual underflow: 3.141592653589793E-305
   3.1415926535898E-310 3.141592653E-315 3.142E-320 0.0
0.0/0.0 is Not-a-Number: NaN
inexact results with float: 0 41 47 55 61 82 83 94 97
```

```
inexact results with double: 0 49 98
cast to int rounds toward 0: 12345 -12345
```

This example demonstrates, among other things, that gradual underflow can result in a gradual loss of precision.

The results when `i` is `0` involve division by zero, so that `z` becomes positive infinity, and `z * 0` is NaN, which is not equal to `1.0`.

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 (§14.11)
- The `do` statement (§14.12)
- The `for` statement (§14.13)

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

Only `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` 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 three kinds of *reference types*: class types (§8), interface types (§9), and array types (§10).

ReferenceType:
ClassOrInterfaceType
ArrayType

ClassOrInterfaceType:
ClassType
InterfaceType

ClassType:
TypeName

InterfaceType:
TypeName

ArrayType:
Type `[ ]`

Names are described in §6; type names in §6.5 and, specifically, §6.5.5.

The sample code:

```
class Point { int[] metrics; }
interface Move { void move(int deltax, int deltay); }
```

declares a class type `Point`, an interface type `Move`, and uses an array type `int[]` (an array of `int`) to declare the field `metrics` of the class `Point`.

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 (§15.9).

A new class instance is implicitly created when the string concatenation operator + (§15.18.1) is used in an 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).

Many of these cases are illustrated in the following example:

```
class Point {
   int x, y;
   Point() { System.out.println("default"); }
   Point(int x, int y) { this.x = x; this.y = y; }

   // A Point instance is explicitly created at class initialization time:
   static Point origin = new Point(0,0);

   // A String can be implicitly created by a + operator:
   public String toString() {
      return "(" + x + "," + y + ")";
   }
}

class Test {
   public static void main(String[] args) {
      // A Point is explicitly created using newInstance:
      Point p = null;
      try {
         p = (Point)Class.forName("Point").newInstance();
      } catch (Exception e) {
         System.out.println(e);
      }

      // An array is implicitly created by an array constructor:
      Point a[] = { new Point(0,0), new Point(1,1) };

      // Strings are implicitly created by + operators:
      System.out.println("p: " + p);
      System.out.println("a: { " + a[0] + ", "
                                 + a[1] + " }");

      // An array is explicitly created by an array creation expression:
      String sa[] = new String[2];
      sa[0] = "he"; sa[1] = "llo";
      System.out.println(sa[0] + sa[1]);
   }
}
```

which produces the output:

```
default
p: (0,0)
a: { (0,0), (1,1) }
hello
```

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.

The example program:

```
class Value { int val; }

class Test {
   public static void main(String[] args) {
      int i1 = 3;
      int i2 = i1;
      i2 = 4;
      System.out.print("i1==" + i1);
      System.out.println(" but i2==" + i2);
      Value v1 = new Value();
      v1.val = 5;
      Value v2 = v1;
      v2.val = 6;
      System.out.print("v1.val==" + v1.val);
      System.out.println(" and v2.val==" + v2.val);
   }
}
```

produces the output:

```
i1==3 but i2==4
v1.val==6 and v2.val==6
```

because `v1.val` and `v2.val` reference the same instance variable (§4.5.3) in the one `Value` object created by the only `new` expression, while `i1` and `i2` are different variables.

See §10 and §15.10 for examples of the creation and use of arrays.

Each object has an associated lock (§17.13), which is used by `synchronized` methods (§8.4.3) and the `synchronized` statement (§14.18) to provide control over concurrent access to state by multiple threads (§17.12).

4.3.2 The Class `Object`

The class `Object` is a superclass (§8.1) of all other classes. A variable of type `Object` can hold a reference to any object, whether it is an instance of a class or an array (§10). All class and array types inherit the methods of class `Object`, which are summarized here:

```
package java.lang;

public class Object {
   public final Class getClass() { ... }
   public String toString() { ... }
   public boolean equals(Object obj) { ... }
   public int hashCode() { ... }
   protected Object clone()
      throws CloneNotSupportedException { ... }
   public final void wait()
      throws IllegalMonitorStateException,
         InterruptedException { ... }
   public final void wait(long millis)
      throws IllegalMonitorStateException,
         InterruptedException { ... }
   public final void wait(long millis, int nanos) { ... }
      throws IllegalMonitorStateException,
         InterruptedException { ... }
   public final void notify() { ... }
      throws IllegalMonitorStateException
   public final void notifyAll() { ... }
      throws IllegalMonitorStateException
   protected void finalize()
      throws Throwable { ... }
}
```

The members of `Object` are as follows:

- The method `getClass` returns the `Class` object that represents the class of the object. A `Class` object exists for each reference type. It can be used, for example, to discover the fully qualified name of a class, its members, its immediate superclass, and any interfaces that it implements. 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 `toString` returns a `String` representation of the object.

- The methods `equals` and `hashCode` are very useful in hashtables such as `java.util.Hashtable`. The method `equals` defines a notion of object equality, which is based on value, not reference, comparison.
- The method `clone` is used to make a duplicate of an object.
- The methods `wait`, `notify`, and `notifyAll` are used in concurrent programming using threads, as described in §17.
- The method `finalize` is run just before an object is destroyed and is described in §12.6.

4.3.3 The Class `String`

Instances of class `String` represent sequences of Unicode characters. 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.

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), in which case 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 loaded 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(§10).

4.4 Where Types Are Used

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

The following code fragment contains one or more instances of most kinds of usage of a type:

```
import java.util.Random;

class MiscMath {

   int divisor;

   MiscMath(int divisor) {
      this.divisor = divisor;
   }

   float ratio(long l) {
      try {
         l /= divisor;
      } catch (Exception e) {
         if (e instanceof ArithmeticException)
            l = Long.MAX_VALUE;
         else
            l = 0;
      }
      return (float)l;
   }

   double gausser() {
      Random r = new Random();
      double[] val = new double[2];
      val[0] = r.nextGaussian();
      val[1] = r.nextGaussian();
      return (val[0] + val[1]) / 2;
   }

}
```

In this example, types are used in declarations of the following:

- Imported types (§7.5); here the type `Random`, imported from the type `java.util.Random` of the package `java.util`, is declared
- Fields, which are the class variables and instance variables of classes (§8.3), and constants of interfaces (§9.3); here the field `divisor` in the class `MiscMath` is declared to be of type `int`
- Method parameters (§8.4.1); here the parameter `l` of the method `ratio` is declared to be of type `long`
- Method results (§8.4); here the result of the method `ratio` is declared to be of type `float`, and the result of the method `gausser` is declared to be of type `double`
- Constructor parameters (§8.8.1); here the parameter of the constructor for `MiscMath` is declared to be of type `int`
- Local variables (§14.4, §14.13); the local variables `r` and `val` of the method `gausser` are declared to be of types `Random` and `double[]` (array of `double`)

- Exception handler parameters (§14.19); here the exception handler parameter `e` of the `catch` clause is declared to be of type `Exception`

and in expressions of the following kinds:

- Class instance creations (§15.9); here a local variable `r` of method `gausser` is initialized by a class instance creation expression that uses the type `Random.`
- Array creations (§15.10); here the local variable `val` of method `gausser` is initialized by an array creation expression that creates an array of `double` with size 2
- Casts (§15.16); here the `return` statement of the method `ratio` uses the `float` type in a cast
- The `instanceof` operator (§15.20.2); here the `instanceof` operator tests whether `e` is assignment compatible with the type `ArithmeticException`

4.5 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 always contains a value that is assignment compatible (§5.2) with its type. A variable's value is changed by an assignment (§15.26) or by a prefix or postfix ++ (increment) or -- (decrement) operator (§15.14.1, §15.14.2, §15.15.1, §15.15.2).

Compatibility of the value of a variable with its type is guaranteed by the design of the Java programming language. Default values are compatible (§4.5.5) and all assignments to a variable are checked for assignment compatibility (§5.2), usually at compile time, but, in a single case involving arrays, a run-time check is made (§10.10).

4.5.1 Variables of Primitive Type

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

4.5.2 Variables of Reference Type

A variable of reference type can hold either of the following:

- A null reference
- A reference to any object (§4.3) whose class (§4.5.6) is assignment compatible (§5.2) with the type of the variable

4.5.3 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 (§4.5.5). 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 (§4.5.5) as part of each newly created object of class *T* or of any class that is a subclass of *T* (§8.1.3). 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 (§4.5.5) whenever a new object that is an array is created (§15.10). The array components effectively cease to exist when the array is no longer referenced. See §10 for a description of arrays.

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 (§8.8.5) 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-handler parameter* is created each time an exception is caught by a `catch` clause of a `try` statement (§14.19). The new variable is initialized with the actual object associated with the exception (§11.3, §14.17). The exception-handler 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 (§14.13), 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 (§16) 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 (§14.10), 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 (§16), 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).

The following example contains several different kinds of variables:

```
class Point {
   static int numPoints;     // numPoints is a class variable
   int x, y;                 // x and y are instance variables
   int[] w = new int[10];    // w[0] is an array component
   int setX(int x) {         // x is a method parameter
      int oldx = this.x;     // oldx is a local variable
      this.x = x;
      return oldx;
   }
}
```

4.5.4 `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 (§16) 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. This applies also to arrays, because arrays are objects; if a `final`

variable holds a reference to an array, then the components of the array may be changed by operations on the array, but the variable will always refer to the same array.Declaring a variable `final` can serve as useful documentation that its value will not change and can help avoid programming errors.

In the example:

```
class Point {
    int x, y;
    int useCount;
    Point(int x, int y) { this.x = x; this.y = y; }
    final static Point origin = new Point(0, 0);
}
```

the class `Point` declares a `final` class variable `origin`. The `origin` variable holds a reference to an object that is an instance of class `Point` whose coordinates are (0, 0). The value of the variable `Point.origin` can never change, so it always refers to the same `Point` object, the one created by its initializer. However, an operation on this `Point` object might change its state—for example, modifying its `useCount` or even, misleadingly, its `x` or `y` coordinate.

4.5.5 Initial Values of Variables

Every variable in a The Java programming language 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 (§8.8.5).
- An exception-handler parameter (§14.19) is initialized to the thrown object representing the exception (§11.3, §14.17).
- A local variable (§14.4, §14.13) 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 compiler using the rules for definite assignment (§16).

The example program:

```
class Point {
   static int npoints;
   int x, y;
   Point root;
}

class Test {
   public static void main(String[] args) {
      System.out.println("npoints=" + Point.npoints);
      Point p = new Point();
      System.out.println("p.x=" + p.x + ", p.y=" + p.y);
      System.out.println("p.root=" + p.root);
   }
}
```

prints:

```
npoints=0
p.x=0, p.y=0
p.root=null
```

illustrating the default initialization of `npoints`, which occurs when the class `Point` is prepared (§12.3.2), and the default initialization of `x`, `y`, and `root`, which occurs when a new `Point` is instantiated. See §12 for a full description of all aspects of loading, linking, and initialization of classes and interfaces, plus a description of the instantiation of classes to make new class instances.

4.5.6 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 (§8.1.4) that interface.

Here is an example of creating new objects and of the distinction between the type of a variable and the class of an object:

```
public interface Colorable {
    void setColor(byte r, byte g, byte b);
}

class Point { int x, y; }

class ColoredPoint extends Point implements Colorable {

    byte r, g, b;

    public void setColor(byte rv, byte gv, byte bv) {
        r = rv; g = gv; b = bv;
    }

}

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        p = cp;
        Colorable c = cp;
    }
}
```

In this example:

- The local variable `p` of the method `main` of class `Test` has type `Point` and is initially assigned a reference to a new instance of class `Point`.
- The local variable `cp` similarly has as its type `ColoredPoint`, and is initially assigned a reference to a new instance of class `ColoredPoint`.
- The assignment of the value of `cp` to the variable `p` causes `p` to hold a reference to a `ColoredPoint` object. This is permitted because `ColoredPoint` is a subclass of `Point`, so the class `ColoredPoint` is assignment compatible (§5.2) with the type `Point`. A `ColoredPoint` object includes support for all the methods of a `Point`. In addition to its particular fields `r`, `g`, and `b`, it has the fields of class `Point`, namely `x` and `y`.
- The local variable `c` has as its type the interface type `Colorable`, so it can hold a reference to any object whose class implements `Colorable`; specifically, it can hold a reference to a `ColoredPoint`.
- Note that an expression such as "`new Colorable()`" is not valid because it is not possible to create an instance of an interface, only of a class.

Every array also has a class; the method `getClass`, when invoked for an array object, will return a class object (of class `Class`) that represents the class of the array. The classes for arrays have strange names that are not valid identifiers; for example, the class for an array of `int` components has the name "`[I`" and so the value of the expression:

```
new int[10].getClass().getName()
```

is the string `"[I"`; see the specification of `Class.getName` for details.

Oft on the dappled turf at ease
I sit, and play with similes,
Loose types of things through all degrees.
—William Wordsworth, *To the Same Flower*

CHAPTER 5

Conversions and Promotions

Thou art not for the fashion of these times,
Where none will sweat but for promotion.
—William Shakespeare, *As You Like It*, Act II, scene iii

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; for example, if the expression in an `if` statement (§14.9) has any type other than `boolean`, a compile-time error occurs. 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*.

For example:

- A conversion from type `Object` to type `Thread` requires a run-time check to make sure that the run-time value is actually an instance of class `Thread` or one of its subclasses; if it is not, an exception is thrown.
- A conversion from type `Thread` to type `Object` requires no run-time action; `Thread` is a subclass of `Object`, so any reference produced by an expression of type `Thread` is a valid reference value of type `Object`.

- A conversion from type `int` to type `long` requires run-time sign-extension of a 32-bit integer value to the 64-bit `long` representation. No information is lost.

A conversion from type `double` to type `long` requires a nontrivial translation from a 64-bit floating-point value to the 64-bit integer representation. Depending on the actual run-time value, information may be lost.

- Identity conversions
- Widening primitive conversions
- Narrowing primitive conversions
- Widening reference conversions
- Narrowing reference 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.

- Assignment conversion (§5.2, §15.26) converts the type of an expression to the type of a specified variable. The conversions permitted for assignment are limited in such a way that assignment conversion never causes an exception.
- 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 never causes an exception.
- Casting conversion (§5.6) 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 (§5.7) brings the operands of a numeric operator to a common type so that an operation can be performed.

Here are some examples of the various contexts for conversion:

```
class Test {
   public static void main(String[] args) {
      // Casting conversion (§5.4) of a float literal to
      // type int. Without the cast operator, this would
      // be a compile-time error, because this is a
      // narrowing conversion (§5.1.3):
      int i = (int)12.5f;

      // String conversion (§5.4) of i's int value:
      System.out.println("(int)12.5f==" + i);

      // Assignment conversion (§5.2) of i's value to type
      // float. This is a widening conversion (§5.1.2):
      float f = i;

      // String conversion of f's float value:
      System.out.println("after float widening: " + f);

      // Numeric promotion (§5.7) of i's value to type
      // float. This is a binary numeric promotion.
      // After promotion, the operation is float*float:
      System.out.print(f);
      f = f * i;

      // Two string conversions of i and f:
      System.out.println("*" + i + "==" + f);

      // Method invocation conversion (§5.3) of f's value
      // to type double, needed because the method Math.sin
      // accepts only a double argument:
      double d = Math.sin(f);

      // Two string conversions of f and d:
      System.out.println("Math.sin(" + f + ")==" + d);

   }
}
```

which produces the output:

```
(int)12.5f==12
after float widening: 12.0
12.0*12==144.0
```

Math.sin(144.0)==-0.49102159389846934

5.1 Kinds of Conversion

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

5.1.1 Identity Conversions

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

This may seem trivial, but it has two practical consequences. First, it is always permitted for an expression to have the desired type to begin with, thus allowing the simply stated rule that every expression is subject to conversion, if only a trivial identity conversion. Second, it implies that it is permitted for a program to include redundant cast operators for the sake of clarity.

The only permitted conversion that involves the type `boolean` is the identity conversion from `boolean` to `boolean`.

5.1.2 Widening Primitive Conversion

The following 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`

Widening primitive conversions do not lose information about the overall magnitude of a numeric value. Indeed, conversions widening from an integral type to another integral type and from `float` to `double` do not lose any information at all; the numeric value is preserved exactly. Conversions widening from `float` to `double` in `strictfp` expressions also preserve the numeric value exactly; however, such conversions that are not `strictfp` may lose information about the overall magnitude about the converted value.

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 character to an integral type *T* zero-extends the representation of the character 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 (§11).

Here is an example of a widening conversion that loses precision:

```
class Test {
   public static void main(String[] args) {
      int big = 1234567890;
      float approx = big;
      System.out.println(big - (int)approx);
   }
}
```

which prints:

```
-46
```

thus indicating that information was lost during the conversion from type `int` to type `float` because values of type `float` are not precise to nine significant digits.

5.1.3 Narrowing Primitive Conversions

The following 23 specific conversions on primitive types are called the *narrowing primitive conversions*:

- `byte` to `char`
- `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`

Narrowing conversions may lose information about the overall magnitude of a numeric value and may also lose precision.

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 character 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 characters 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 (§4.2.4). Then there are two cases:
     - 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*.
     - 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:
     - 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`.
     - 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.

The example:

```
class Test {
   public static void main(String[] args) {
      float fmin = Float.NEGATIVE_INFINITY;
      float fmax = Float.POSITIVE_INFINITY;
      System.out.println("long: " + (long)fmin +
                         ".." + (long)fmax);
      System.out.println("int: " + (int)fmin +
                         ".." + (int)fmax);
      System.out.println("short: " + (short)fmin +
                         ".." + (short)fmax);
      System.out.println("char: " + (int)(char)fmin +
                         ".." + (int)(char)fmax);
      System.out.println("byte: " + (byte)fmin +
                         ".." + (byte)fmax);
   }
}
```

produces the output:

```
long: -9223372036854775808..9223372036854775807
int: -2147483648..2147483647
short: 0..-1
char: 0..65535
byte: 0..-1
```

The results for `char`, `int`, and `long` are unsurprising, producing the minimum and maximum representable values of the type.

The results for `byte` and `short` lose information about the sign and magnitude of the numeric values and also lose precision. The results can be understood by examining the low order bits of the minimum and maximum `int`. The minimum `int` is, in hexadecimal, `0x80000000`, and the maximum `int` is `0x7fffffff`. This explains the `short` results, which are the low 16 bits of these values, namely, `0x0000` and `0xffff`; it explains the `char` results, which also are the low 16 bits of these values, namely, `'\u0000'` and `'\uffff'`; and it explains the `byte` results, which are the low 8 bits of these values, namely, `0x00` and `0xff`.

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 (§11).

Here is a small test program that demonstrates a number of narrowing conversions that lose information:

```
class Test {
   public static void main(String[] args) {
      // A narrowing of int to short loses high bits:
      System.out.println("(short)0x12345678==0x" +
                Integer.toHexString((short)0x12345678));
      // A int value not fitting in byte changes sign and magnitude:
      System.out.println("(byte)255==" + (byte)255);
      // A float value too big to fit gives largest int value:
      System.out.println("(int)1e20f==" + (int)1e20f);
      // A NaN converted to int yields zero:
      System.out.println("(int)NaN==" + (int)Float.NaN);
      // A double value too large for float yields infinity:
      System.out.println("(float)-1e100==" + (float)-1e100);
      // A double value too small for float underflows to zero:
      System.out.println("(float)1e-50==" + (float)1e-50);
   }
}
```

This test program produces the following output:

```
(short)0x12345678==0x5678
(byte)255==-1
(int)1e20f==2147483647
(int)NaN==0
(float)-1e100==-Infinity
(float)1e-50==0.0
```

5.1.4 Widening Reference Conversions

The following conversions are called the *widening reference conversions*:

- From any class type *S* to any class type *T*, provided that *S* is a subclass of *T*. (An important special case is that there is a widening conversion to the class type `Object` from any other class type.)
- From any class type *S* to any interface type *K*, provided that *S* implements *K*.
- From the null type to any class type, interface type, or array type.
- From any interface type *J* to any interface type *K*, provided that *J* is a subinterface of *K*.
- From any interface type to type `Object`.
- From any array type to type `Object`.

- From any array type to type `Cloneable`.
- From any array type to type `java.io.Serializable`
- From any array type *SC*`[]` to any array type *TC*`[]`, provided that *SC* and *TC* are reference types and there is a widening conversion from *SC* to *TC*.

Such 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.

See §8 for the detailed specifications for classes, §9 for interfaces, and §10 for arrays.

5.1.5 Narrowing Reference Conversions

The following conversions are called the *narrowing reference conversions*:

- From any class type *S* to any class type *T*, provided that *S* is a superclass of *T*. (An important special case is that there is a narrowing conversion from the class type `Object` to any other class type.)
- From any class type *S* to any interface type *K*, provided that *S* is not final and does not implement *K*. (An important special case is that there is a narrowing conversion from the class type `Object` to any interface type.)
- From type `Object` to any array type.
- From type `Object` to any interface type.
- From any interface type *J* to any class type *T* that is not `final`.
- From any interface type *J* to any class type *T* that is `final`, provided that *T* implements *J*.
- From any interface type *J* to any interface type *K*, provided that *J* is not a subinterface of *K* and there is no method name *m* such that *J* and *K* both contain a method named *m* with the same signature but different return types.
- From any array type *SC*`[]` to any array type *TC*`[]`, provided that *SC* and *TC* are reference types and there is a narrowing 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.

5.1.6 String Conversions

There is a string conversion to type `String` from every other type, including the null type.

5.1.7 Forbidden Conversions

- There is no permitted conversion from any reference type to any primitive type.
- Except for the string conversions, there is no permitted conversion from any primitive type to any reference type.
- There is no permitted conversion from the null type to any primitive type.
- There is no permitted conversion to the null type other than the identity conversion.
- There is no permitted conversion to the type `boolean` other than the identity conversion.
- There is no permitted conversion from the type `boolean` other than the identity conversion and string conversion.
- There is no permitted conversion other than string conversion from class type *S* to a different class type *T* if *S* is not a subclass of *T* and *T* is not a subclass of *S*.
- There is no permitted conversion from class type *S* to interface type *K* if *S* is `final` and does not implement *K*.
- There is no permitted conversion from class type *S* to any array type if *S* is not `Object`.
- There is no permitted conversion other than string conversion from interface type *J* to class type *T* if *T* is `final` and does not implement *J*.
- There is no permitted conversion from interface type *J* to interface type *K* if *J* and *K* contain methods with the same signature but different return types.
- There is no permitted conversion from any array type to any class type other than `Object` or `String`.
- There is no permitted conversion from any array type to any interface type, except to the interface types `java.io.Serializable` and `Cloneable`, which are implemented by all arrays.

- There is no permitted conversion from array type *SC*`[]` to array type *TC*`[]` if there is no permitted conversion other than a string conversion from *SC* to *TC*`.`

5.1.8 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 an identity conversion (§5.1.1), a widening primitive conversion (§5.1.2), or a widening reference conversion (§5.1.4). In addition, a narrowing primitive conversion may be used if all of the following conditions are satisfied:

- The expression is a constant expression of type `byte`, `short`, `char` or `int` .
- The type of the variable is `byte`, `short`, or `char`.
- The value of the expression (which is known at compile time, because it is a constant expression) is representable in the type of the variable.

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 after the type conversion:

- If the value is of type `float` and 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 the value is of type `double` and 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 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.

An assignment conversion never causes an exception. (Note, however, that an assignment may result in an exception in a special case involving array elements —see §10.10 and §15.26.1.)

The compile-time narrowing of constants means that code such as:

```
byte theAnswer = 42;
```

is allowed. Without the narrowing, the fact that the integer literal `42` has type `int` would mean that a cast to `byte` would be required:

byte theAnswer = (byte)42;// *cast is permitted but not required*

A value of primitive type must not be assigned to a variable of reference type; an attempt to do so will result in a compile-time error. A value of type `boolean` can be assigned only to a variable of type `boolean`.

The following test program contains examples of assignment conversion of primitive values:

```
class Test {
   public static void main(String[] args) {
      short s = 12;          // narrow 12 to short
      float f = s;           // widen short to float
      System.out.println("f=" + f);
      char c = '\u0123';
      long l = c;            // widen char to long
      System.out.println("l=0x" + Long.toString(l,16));
      f = 1.23f;
      double d = f;          // widen float to double
      System.out.println("d=" + d);
   }
}
```

It produces the following output:

```
f=12.0
l=0x123
d=1.2300000190734863
```

The following test, however, produces compile-time errors:

```
class Test {
   public static void main(String[] args) {
      short s = 123;
      char c = s;            // error: would require cast
      s = c;                 // error: would require cast
   }
}
```

because not all `short` values are `char` values, and neither are all `char` values `short` values.

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.

Here is a sample program illustrating assignments of references:

```
public class Point { int x, y; }

public class Point3D extends Point { int z; }

public interface Colorable {
   void setColor(int color);
}
```

```
public class ColoredPoint extends Point implements Colorable
{
   int color;
   public void setColor(int color) { this.color = color; }
}




class Test {
   public static void main(String[] args) {
      // Assignments to variables of class type:
      Point p = new Point();
      p = new Point3D();     // ok: because Point3D is a
                             // subclass of Point

      Point3D p3d = p;       // error: will require a cast because a
                             // Point might not be a Point3D
                             // (even though it is, dynamically,
                             // in this example.)
      // Assignments to variables of type Object:
      Object o = p;          // ok: any object to Object
      int[] a = new int[3];
      Object o2 = a;         // ok: an array to Object

      // Assignments to variables of interface type:
      ColoredPoint cp = new ColoredPoint();
      Colorable c = cp;      // ok: ColoredPoint implements
                             // Colorable

      // Assignments to variables of array type:
      byte[] b = new byte[4];
      a = b;                 // error: these are not arrays
                             // of the same primitive type
      Point3D[] p3da = new Point3D[3];
      Point[] pa = p3da;     // ok: since we can assign a
                             // Point3D to a Point
      p3da = pa;             // error: (cast needed) since a Point
                             // can't be assigned to a Point3D
   }
}
```

Assignment of a value of compile-time reference type *S* (source) to a variable of compile-time reference type *T* (target) is checked as follows:

- If *S* is a class type:
  - If *T* is a class type, then *S* must either be the same class as *T*, or *S* must be a subclass of *T*, or a compile-time error occurs.
  - If *T* is an interface type, then *S* must implement interface *T*, or a compile-time error occurs.
  - If *T* is an array type, then a compile-time error occurs.
- If *S* is an interface type:
  - If *T* is a class type, then *T* must be `Object`, or a compile-time error occurs.
  - If *T* is an interface type, then *T* must be either the same interface as *S* or a superinterface of *S*, or a compile-time error occurs.
  - If *T* is an array type, then a compile-time error occurs.
- If *S* is an array type *SC*`[]`, that is, an array of components of type *SC*:
  - If *T* is a class type, then *T* must be `Object`, or a compile-time error occurs.
  - 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 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 both reference types and type *SC* is assignable to *TC*, as determined by a recursive application of these compile-time rules for assignability.

See §8 for the detailed specifications of classes, §9 for interfaces, and §10 for arrays.

The following test program illustrates assignment conversions on reference values, but fails to compile because it violates the preceding rules, as described in its comments. This example should be compared to the preceding one.

```
public class Point { int x, y; }

public interface Colorable { void setColor(int color); }

public class ColoredPoint extends Point implements Colorable
{
   int color;
   public void setColor(int color) { this.color = color; }
}
```

```
class Test {
   public static void main(String[] args) {

      Point p = new Point();

      ColoredPoint cp = new ColoredPoint();

      // Okay because ColoredPoint is a subclass of Point:
      p = cp;

      // Okay because ColoredPoint implements Colorable:
      Colorable c = cp;

      // The following cause compile-time errors because
      // we cannot be sure they will succeed, depending on
      // the run-time type of p; a run-time check will be
      // necessary for the needed narrowing conversion and
      // must be indicated by including a cast:
      cp = p;       // p might be neither a ColoredPoint
                    // nor a subclass of ColoredPoint
      c = p;        // p might not implement Colorable

   }
}
```

Here is another example involving assignment of array objects:

```
class Point { int x, y; }

class ColoredPoint extends Point { int color; }

class Test {
   public static void main(String[] args) {
      long[] veclong = new long[100];
      Object o = veclong;          // okay
      Long l = veclong;            // compile-time error
      short[] vecshort = veclong;  // compile-time error
      Point[] pvec = new Point[100];
      ColoredPoint[] cpvec = new ColoredPoint[100];
      pvec = cpvec;                // okay
      pvec[0] = new Point();       // okay at compile time,
                                   // but would throw an
                                   // exception at run time
      cpvec = pvec;                // compile-time error
   }
}
```

In this example:

- The value of `veclong` cannot be assigned to a `Long` variable, because `Long` is a class type other than `Object`. An array can be assigned only to a variable of a compatible array type, or to a variable of type `Object`.
- The value of `veclong` cannot be assigned to `vecshort`, because they are arrays of primitive type, and `short` and `long` are not the same primitive type.
- The value of `cpvec` can be assigned to `pvec`, because any reference that could be the value of an expression of type `ColoredPoint` can be the value of a variable of type `Point`. The subsequent assignment of the new `Point` to a component of `pvec` then would throw an `ArrayStoreException` (if the program were otherwise corrected so that it could be compiled), because a `ColoredPoint` array can't have an instance of `Point` as the value of a component.
- The value of `pvec` cannot be assigned to `cpvec`, because not every reference that could be the value of an expression of type `ColoredPoint` can correctly be the value of a variable of type `Point`. If the value of `pvec` at run time were a reference to an instance of `Point[]`, and the assignment to `cpvec` were allowed, a simple reference to a component of `cpvec`, say, `cpvec[0]`, could return a `Point`, and a `Point` is not a `ColoredPoint`. Thus to allow such an assignment would allow a violation of the type system. A cast may be used (§5.6, §15.16) to ensure that `pvec` references a `ColoredPoint[]`:

```
cpvec = (ColoredPoint[])pvec;     // okay, but may throw an
        // exception at run time
```

5.3 Method Invocation Conversion

Method invocation conversion is applied to each argument value in a method or constructor invocation (§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 an identity conversion (§5.1.1), a widening primitive conversion (§5.1.2), or a widening reference conversion (§5.1.4).

If the type of an argument expression is either `float` or `double`, then value set conversion (§5.1.8) 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 ele-

ment 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.

Method invocation conversions specifically do not include the implicit narrowing of integer constants which is part of assignment conversion (§5.2). The designers of the Java programming language felt that including these implicit narrowing conversions would add additional complexity to the overloaded method matching resolution process (§15.12.2). Thus, the example:

```
class Test {
   static int m(byte a, int b) { return a+b; }
   static int m(short a, short b) { return a-b; }
   public static void main(String[] args) {
      System.out.println(m(12, 2));   // compile-time error
   }
}
```

causes a compile-time error because the integer literals `12` and `2` have type `int`, so neither method `m` matches under the rules of (§15.12.2). A language that included implicit narrowing of integer constants would need additional rules to resolve cases like this example.

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 + is converted to a `String`, and a new `String` which is the concatenation of the two strings is the result of the +. String conversion is specified in detail within the description of the string concatenation + operator (§15.18.1).

5.5 Casting Conversion

Sing away sorrow, cast away care.
—Miguel de Cervantes (1547–1616),
Don Quixote (Lockhart's translation), Chapter viii

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 an identity conversion (§5.1.1), a widening primitive conversion (§5.1.2), a narrowing primitive conversion (§5.1.3), a widening reference conversion (§5.1.4), or a narrowing reference conversion (§5.1.5). Thus casting conversions are more inclusive than assignment or method invocation conversions: a cast can do any permitted conversion other than a string conversion.

Value set conversion (§5.1.8) is applied after the type conversion.

Some casts can be proven incorrect at compile time; such casts result in a compile-time error.

A value of a primitive type can be cast to another primitive type by identity conversion, if the types are the same, or by a widening primitive conversion or a narrowing primitive conversion.

A value of a primitive type cannot be cast to a reference type by casting conversion, nor can a value of a reference type be cast to a primitive type.

The remaining cases involve conversion between reference types. The detailed rules for compile-time correctness checking of a casting conversion of a value of compile-time reference type *S* (source) to a compile-time reference type *T* (target) are as follows:

- If *S* is a class type:
  - If *T* is a class type, then *S* and *T* must be related classes—that is, *S* and *T* must be the same class, or *S* a subclass of *T,* or *T* a subclass of *S*; otherwise a compile-time error occurs.
  - If *T* is an interface type:
    - If *S* is not a `final` class (§8.1.1), then the cast is always correct 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 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 *T* must implement *S*, or a compile-time error occurs.

  If *T* is a class type that is not `final` (§8.1.1), then the cast is always correct at compile time (because even if *T* does not implement *S*, a subclass of *T* might).

- If *T* is an interface type and if *T* and *S* contain methods with the same signature (§8.4.2) but different return types, then a compile-time error occurs.

- 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 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 be cast to *TC* by a recursive application of these compile-time rules for casting.

See §8 for the detailed specifications of classes, §9 for interfaces, and §10 for arrays.

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

- The cast can be determined to be correct at compile time. A cast from the compile-time type *S* to compile-time type *T* is correct at compile time if and only if *S* can be converted to *T* by assignment conversion (§5.2).

The 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 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*, using the algorithm specified in §5.2 but using the class *R* instead of the compile-time type *S* as specified there.

(Note that *R* cannot be an interface when these rules are first applied for any given cast, but *R* may be an interface if the rules are applied recursively because the run-time reference value may refer to an array whose element type is an interface type.)

- 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 (§8.1.4) 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 (this case could slip past the compile-time checking if, for example, a reference to an array were stored in a variable of type `Object`).
  - 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.

If a run-time exception is thrown, it is a `ClassCastException`.

Here are some examples of casting conversions of reference types, similar to the example in §5.2:

```
public class Point { int x, y; }

public interface Colorable { void setColor(int color); }

public class ColoredPoint extends Point implements Colorable
{
   int color;
   public void setColor(int color) { this.color = color; }
}

final class EndPoint extends Point { }

class Test {
   public static void main(String[] args) {
      Point p = new Point();
```

```
        ColoredPoint cp = new ColoredPoint();
        Colorable c;

        // The following may cause errors at run time because
        // we cannot be sure they will succeed; this possibility
        // is suggested by the casts:
        cp = (ColoredPoint)p;  // p might not reference an
                               // object which is a ColoredPoint
                               // or a subclass of ColoredPoint
        c = (Colorable)p;      // p might not be Colorable

        // The following are incorrect at compile time because
        // they can never succeed as explained in the text:
        Long l = (Long)p;      // compile-time error #1
        EndPoint e = new EndPoint();
        c = (Colorable)e;      // compile-time error #2
    }
}
```

Here the first compile-time error occurs because the class types `Long` and `Point` are unrelated (that is, they are not the same, and neither is a subclass of the other), so a cast between them will always fail.

The second compile-time error occurs because a variable of type `EndPoint` can never reference a value that implements the interface `Colorable`. This is because `EndPoint` is a `final` type, and a variable of a `final` type always holds a value of the same run-time type as its compile-time type. Therefore, the run-time type of variable `e` must be exactly the type `EndPoint`, and type `EndPoint` does not implement `Colorable`.

Here is an example involving arrays (§10):

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    public String toString() { return "("+x+","+y+")"; }
}
public interface Colorable { void setColor(int color); }
public class ColoredPoint extends Point implements Colorable
{
    int color;
    ColoredPoint(int x, int y, int color) {
        super(x, y); setColor(color);
    }
    public void setColor(int color) { this.color = color; }
```

```
    public String toString() {
        return super.toString() + "@" + color;
    }
}
class Test {
    public static void main(String[] args) {
        Point[] pa = new ColoredPoint[4];
        pa[0] = new ColoredPoint(2, 2, 12);
        pa[1] = new ColoredPoint(4, 5, 24);
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.print("cpa: {");
        for (int i = 0; i < cpa.length; i++)
            System.out.print((i == 0 ? " " : ", ") + cpa[i]);
        System.out.println(" }");
    }
}
```

This example compiles without errors and produces the output:

```
cpa: { (2,2)@12, (4,5)@24, null, null }
```

The following example uses casts to compile, but it throws exceptions at run time, because the types are incompatible:

```
public class Point { int x, y; }
public interface Colorable { void setColor(int color); }
public class ColoredPoint extends Point implements Colorable
{
    int color;
    public void setColor(int color) { this.color = color; }
}
class Test {
    public static void main(String[] args) {
        Point[] pa = new Point[100];
        // The following line will throw a ClassCastException:
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.println(cpa[0]);
        int[] shortvec = new int[2];
        Object o = shortvec;
        // The following line will throw a ClassCastException:
        Colorable c = (Colorable)o;
```

```
            c.setColor(0);
        }
    }
```

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) or a widening primitive conversion (§5.1.2).

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 (§5.7.1) and binary numeric promotion (§5.7.2). The analogous conversions in C are called "the usual unary conversions" and "the usual binary conversions."

Numeric promotion is not a general feature of the Java programming language, but rather a property of the specific definitions of the built-in operations.

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`, or `char`, unary numeric promotion promotes it to a value of type `int` by a widening conversion (§5.1.2).
- Otherwise, a unary numeric operand remains as is and is not converted.

In either case, value set conversion (§5.1.8) 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); therefore a `long` shift distance (right operand) does not promote the value being shifted (left operand) to `long`

Here is a test program that includes examples of unary numeric promotion:

```
class Test {
   public static void main(String[] args) {
      byte b = 2;
      int a[] = new int[b];  // dimension expression promotion
      char c = '\u0001';
      a[c] = 1;              // index expression promotion
      a[0] = -c;             // unary - promotion
      System.out.println("a: " + a[0] + "," + a[1]);
      b = -1;
      int i = ~b;            // bitwise complement promotion
      System.out.println("~0x" + Integer.toHexString(b)
                        + "==0x" + Integer.toHexString(i));
      i = b << 4L;           // shift promotion (left operand)
      System.out.println("0x" + Integer.toHexString(b)
                 + "<<4L==0x" + Integer.toHexString(i));
   }
}
```

This test program produces the output:

```
a: -1,1
~0x0==0x-1
0xffffffff<<4L==0xfffffff0
```

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 of a numeric type, the following rules apply, in order, using widening conversion (§5.1.2) to convert operands as necessary:

- 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 (§5.1.8) 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)

An example of binary numeric promotion appears above in §5.1. Here is another:

```
class Test {
   public static void main(String[] args) {
      int i = 0;
      float f = 1.0f;
      double d = 2.0;

      // First int*float is promoted to float*float, then
      // float==double is promoted to double==double:
      if (i * f == d)
         System.out.println("oops");

      // A char&byte is promoted to int&int:
      byte b = 0x1f;
      char c = 'G';
      int control = c & b;
      System.out.println(Integer.toHexString(control));

      // Here int:float is promoted to float:float:
      f = (b==0) ? i : 4.0f;
      System.out.println(1.0/f);

   }
}
```

which produces the output:

```
7
0.25
```

The example converts the ASCII character G to the ASCII control-G (BEL), by masking off all but the low 5 bits of the character. The 7 is the numeric value of this control character.

O suns! O grass of graves! O perpetual transfers and promotions!
—Walt Whitman, *Walt Whitman* (1855),
in *Leaves of Grass*

CHAPTER 6
Names

The Tao that can be told is not the eternal Tao;
The name that can be named is not the eternal name.
The Nameless is the origin of Heaven and Earth;
The Named is the mother of all things.
—Lao-Tsu (c. 6th century BC)

NAMES are used to refer to entities declared in a program. A declared entity (§6.1) is a package, class type, interface type, member (class, interface, field, or method) of a reference type, 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.

Packages and reference types (that is, class types, interface types, and array types) have members (§6.4). A member can be referred to using a qualified name `N.x`, 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) and naming conventions (§6.8) are also discussed in this chapter.

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), or an `instanceof` expression (§15.20.2), 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.

6.1 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 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).
  - A field, one of the following:
    - A field declared in a class type (§8.3)

        - A constant 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 (§14.19)
- A local variable, one of the following:
    - A local variable declared in a block (§14.4)
    - A local variable declared in a `for` statement (§14.13)

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. It distinguishes 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 programs 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
- 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.

- As labels in labeled statements (§14.7) and in `break` (§14.14) and `continue` (§14.15) statements that refer to statement labels

In the example:

```
class Test {
   public static void main(String[] args) {
      Class c = System.out.getClass();
      System.out.println(c.toString().length() +
                         args[0].length() + args.length);
   }
}
```

the identifiers `Test`, `main`, and the first occurrences of `args` and `c` are not names; rather, they are used in declarations to specify the names of the declared entities. The names `String`, `Class`, `System.out.getClass`, `System.out.println`, `c.toString`, `args`, and `args.length` appear in the example. The first occurrence of `length` is not a name, but rather an identifier appearing in a method invocation expression (§15.12). The second occurrence of `length` is not a name, but rather an identifier appearing in a method invocation expression (§15.12).

The identifiers used in labeled statements and their associated `break` and `continue` statements are completely separate from those used in declarations. Thus, the following code is valid:

```
class TestString {
   char[] value;
   int offset, count;
   int indexOf(TestString str, int fromIndex) {
      char[] v1 = value, v2 = str.value;
      int max = offset + (count - str.count);
      int start = offset + ((fromIndex < 0) ? 0 : fromIndex);
   i:
      for (int i = start; i <= max; i++)
      {
         int n = str.count, j = i, k = str.offset;
```

```
            while (n-- != 0) {
               if (v1[j++] != v2[k++])
                  continue i;
            }
            return i - offset;
         }
         return -1;
      }
   }
```

This code was taken from a version of the class `String` and its method `indexOf`, where the label was originally called `test`. Changing the label to have the same name as the local variable `i` does not hide the label in the scope of the declaration of `i`. The identifier `max` could also have been used as the statement label; the label would not hide the local variable `max` within the labeled statement.

6.3 Scope of a Simple Name

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 not hidden (§6.3.1)). The scoping rules for various constructs are given in the sections that describe those constructs. For convenience, the rules are repeated here:
The scope (§6.3, §6.5) of a top level package is determined by conventions of the host system. The package `java` is always in scope. Subpackage names are never in scope.

The scope of a type imported by a single-type-import declaration (§7.5.1) or 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.

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

The scope (§6.3) of a label declared by a a labeled statement is the statement immediately enclosed by the labeled statement

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

The scope of the name of a member *m* declared in or inherited by an interface type *I* is the entire body of *I*, including any nested type declarations.
The declaration of a member needs to appear 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.
- *C* is the innermost class or interface enclosing the usage.

A compile-time error occurs if any of the three requirements above are not met.

This means that a compile-time error results from the test program:

```
class Test {
   int i = j;   // compile-time error: incorrect forward reference
   int j = 1;
}
```

whereas the following example compiles without error:

```
class Test {
   Test() { k = 2; }
   int j = 1;
   int i = j;
   int k;
}
```

even though the constructor (§8.8) for `Test` refers to the field `k` that is declared three lines later. A more elaborate example is:

```
class UseBeforeDeclaration {

  static {

    x = 100; // ok - assignment
    int y = x + 1; // error - read before declaration
    int v = x = 3; // ok - x at left hand side of assignment
int z = useBeforeDeclaration.x * 2; // ok - not accessed via
simple name

    Object o = new Object(){ void foo(){x++;} // ok - occurs
in a different class
                             {x++;} // ok - occurs in a dif-
ferent class
    };
  }

    {
      j = 200; // ok - assignment
      j = j + 1; // error - right hand side reads before dec-
laration
```

```
      int k = j = j + 1;
       int n = j = 300; // ok - j at left hand side of assign-
ment
      int h = j++; // error - read before declaration
     int l = this.j * 3; // ok - not accessed via simple name
    Object o = new Object(){ void foo(){j++;} // ok - occurs
in a different class
                            { j = j + 1;} // ok - occurs in
a different class
      };
    }

int w= x= 3; // ok - x at left hand side of assignment
int p = x; // ok - instance initializers may access static
fields
  static int u =(new Object(){int bar(){return x;}}).bar(); /
/ ok - occurs in a different class
  static int x;
int m = j = 4; // ok - j at left hand side of assignment
int o =(new Object(){int bar(){return j;}}).bar(); // ok -
occurs in a different class
  int j;
}
```

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 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 class declared in a block is the rest of the immediately enclosing block, including its own class declaration.

- The scope of a local variable declared in the *ForInit* part of a `for` statement (§14.13) 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 parameter of an exception handler that is declared in a `catch` clause of a `try` statement (§14.19) is the entire block associated with the `catch`.

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

In the example:

```
package points;

class Point {
   int x, y;
   PointList list;
   Point next;
}

class PointList {
   Point first;
}
```

the use of `PointList` in class `Point` is correct, because the scope of the class type name `PointList` includes both class `Point` and class `PointList`, as well as any other type declarations in other compilation units of package `points`.

6.3.1 Hiding Names

Some declarations may be *hidden* 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* makes the declarations of any other types named *n* that are in scope at the point where *d* occurs hidden throughout the scope of *d*.

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

A declaration *d* of a label named *n* makes the declarations of any other labels named *n* that are in scope at the point where *d* occurs hidden throughout the scope of *d*.

A declaration *d* of a method named *n* makes the declarations of any other methods named *n* that are in scope at the point where *d* occurs hidden throughout the scope of *d*.

A package declaration never causes any other declaration to be hidden.

A single-type-import declaration *d* that imports a type named *n* hides the declarations of any top level type named *n* declared in another compilation unit of the package in which *d* appears, and of any type named *n* imported by a type-import-on-demand declaration.

A type-import-on-demand declaration never causes any other declaration to be hidden. A declaration *d* is said to be *non-hidden at point p in a program* if the scope of *d* includes *p*, and *d* is not hidden 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 *non-hidden*.

The example:

```
class Test {
   static int x = 1;
   public static void main(String[] args) {
      int x = 0;
      System.out.print("x=" + x);
      System.out.println(", Test.x=" + Test.x);
   }
}
```

produces the output:

```
x=0, Test.x=1
```

This example declares:

- a class `Test`
- a class (`static`) variable `x` that is a member of the class `Test`
- a class method `main` that is a member of the class `Test`
- a parameter `args` of the `main` method
- a local variable `x` of the `main` method

Since the scope of a class variable includes the entire body of the class (§8.2) the class variable `x` would normally be available throughout the entire body of the method `main`. In this example, however, the class variable `x` is hidden within the body of the method `main` by the declaration of the local variable `x`.

A local variable has as its scope the rest of the block in which it is declared (§14.4.2); in this case this is the rest of the body of the `main` method, namely its initializer "`0`" and the invocations of `print` and `println`.

This means that:

- The expression "`x`" in the invocation of `print` refers to (denotes) the value of the local variable `x`.
- The invocation of `println` uses a qualified name (§6.6) `Test.x`, which uses the class type name `Test` to access the class variable `x`, because the declaration of `Test.x` is hidden at this point and cannot be referred to by its simple name.

If the standard naming conventions (§6.8) are followed, then hiding that would make the identification of separate naming contexts matter should be rare. The following contrived example involves hiding because it does not follow the standard naming conventions:

```
class Point { int x, y; }
class Test {
   static Point Point(int x, int y) {
      Point p = new Point();
      p.x = x; p.y = y;
      return p;
           }
   public static void main(String[] args) {
      int Point;
      Point[] pa = new Point[2];
      for (Point = 0; Point < 2; Point++) {
         pa[Point] = new Point();
         pa[Point].x = Point;
         pa[Point].y = Point;
      }
      System.out.println(pa[0].x + "," + pa[0].y);
      System.out.println(pa[1].x + "," + pa[1].y);
      Point p = Point(3, 4);
      System.out.println(p.x + "," + p.y);
           }
}
```

This compiles without error and executes to produce the output:

```
0,0
1,1
3,4
```

Within the body of `main`, the lookups of `Point` find different declarations depending on the context of the use:

- In the expression "`new Point[2]`", the two occurrences of the class instance creation expression "`new Point()`", and at the start of three different local variable declaration statements, the `Point` is a *TypeName* (§6.5.5) and denotes the class type `Point` in each case.
- In the method invocation expression "`Point(3, 4)`" the occurrence of `Point` is a *MethodName* (§6.5.7) and denotes the class (`static`) method `Point`.

All other names are *ExpressionName*s (§6.5.6) and refer to the local variable `Point`.

The example:

```
import java.util.*;

class Vector {
   int val[] = { 1 , 2 };
}

class Test {
   public static void main(String[] args) {
      Vector v = new Vector();
      System.out.println(v.val[0]);
   }
}
```

compiles and prints:

```
1
```

using the class `Vector` declared here in preference to class `java.util.Vector` that might be imported on demand.

6.4 Members and Inheritance

Packages and reference types have *members*. The members of a package (§7) are specified in §7.1. For convenience, we repeat these specifications below.

The members of a package are subpackages and all the top level class (§8) and top level interface (§9) types declared in all the compilation units (§7.3) of the package.

The members of a reference type (§4.3) are classes (§8.5, §9.5), interfaces (§8.5, §9.5), fields (§8.3, §9.3, §10.7), and methods (§8.4, §9.4). Members are either declared in the type, or *inherited* because they are accessible members of a superclass or superinterface which are neither private nor hidden nor overridden (§8.4.6).

This section provides an overview of the members of packages and reference types here, as background for the discussion of qualified names and the determination of the meaning of names. For a complete description of membership, see §7.1, §8.2, §9.2, and §10.7.

6.4.1 The Members of a Package

The members of a package are subpackages and all the top level class (§8) and top level interface (§9) types declared in all the compilation units (§7.3) of the package.

In general, the subpackages of a package are determined by the host system (§7.2). However, the package `java` always includes the subpackage `lang` and may

include other subpackages. No two distinct members of the same package may have the same simple name (§7.1), but members of different packages may have the same simple name. For example, it is possible to declare a package:

```
package vector;

public class Vector { Object[] vec; }
```

that has as a member a `public` class named `Vector`, even though the package `java.util` also declares a class named `Vector`. These two class types are different, reflected by the fact that they have different fully qualified names (§6.7). The fully qualified name of this example `Vector` is `vector.Vector`, whereas `java.util.Vector` is the fully qualified name of the standard `Vector` class. Because the package `vector` contains a class named `Vector`, it cannot also have a subpackage named `Vector`.

6.4.2 The Members of a Class Type

The members of a class type (§8.2) are classes, interfaces, fields and methods. The members of a class type are all of the following:

- Members inherited from its direct superclass (§8.1.3), if it has one (the class `Object` has no direct superclass)
- Members inherited from any direct superinterfaces (§8.1.4)

Members declared in the body of the class (§8.1.5)

Constructors (§8.8) are not members.

There is no restriction against a field and a method of a class type having the same simple name. Likewise, there is no restriction against a member class or member interface of a class type having the same simple name as a field or method of that class type.

A class may have two or more fields with the same simple name if they are declared in different interfaces and inherited. An attempt to refer to any of the fields by its simple name results in a compile-time error (§6.5.7.2, §8.2).

In the example:

```
interface Colors {
   int WHITE = 0, BLACK = 1;
}

interface Separates {
   int CYAN = 0, MAGENTA = 1, YELLOW = 2, BLACK = 3;
}

class Test implements Colors, Separates {
   public static void main(String[] args) {
```

```
        System.out.println(BLACK); // compile-time error: ambiguous
    }
}
```

the name `BLACK` in the method `main` is ambiguous, because class `Test` has two members named `BLACK`, one inherited from `Colors` and one from `Separates`.

A class type may have two or more methods with the same simple name if the methods have different signatures (§8.4.2), that is, if they have different numbers of parameters or different parameter types in at least one parameter position. Such a method member name is said to be *overloaded*.

A class type may contain a declaration for a method with the same name and the same signature as a method that would otherwise be inherited from a superclass or superinterface. In this case, the method of the superclass or superinterface is not inherited. If the method not inherited is `abstract`, then the new declaration is said to *implement* it; if the method not inherited is not `abstract`, then the new declaration is said to *override* it.

In the example:

```
class Point {
    float x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
    void move(float dx, float dy) { x += dx; y += dy; }
    public String toString() { return "("+x+","+y+")"; }
}
```

the class `Point` has two members that are methods with the same name, `move`. The overloaded `move` method of class `Point` chosen for any particular method invocation is determined at compile time by the overloading resolution procedure given in §15.12.

In this example, the members of the class `Point` are the `float` instance variables `x` and `y` declared in `Point`, the two declared `move` methods, the declared `toString` method, and the members that `Point` inherits from its implicit direct superclass `Object` (§4.3.2), such as the method `hashCode` . Note that `Point` does not inherit the `toString` method of class `Object` because that method is overridden by the declaration of the `toString` method in class `Point`.

6.4.3 The Members of an Interface Type

The members of an interface type (§9.2) are classes, interfaces, fields and methods. The members of an interface are all of the following:

- Members inherited from any direct superinterfaces (§9.1.2)
- Members declared in the body of the interface (§9.1.3)

An interface may have two or more fields with the same simple name if they are declared in different interfaces and inherited. An attempt to refer to any such field by its simple name results in a compile-time error (§6.5.6.1, §9.2).

In the example:

```
interface Colors {
   int WHITE = 0, BLACK = 1;
}

interface Separates {
   int CYAN = 0, MAGENTA = 1, YELLOW = 2, BLACK = 3;
}

interface ColorsAndSeparates extends Colors, Separates {
   int DEFAULT = BLACK;          // compile-time error: ambiguous
}
```

the members of the interface `ColorsAndSeparates` include those members inherited from `Colors` and those inherited from `Separates`, namely `WHITE`, `BLACK` (first of two), `CYAN`, `MAGENTA`, `YELLOW`, and `BLACK` (second of two). The member name `BLACK` is ambiguous in the interface `ColorsAndSeparates`.

6.4.4 The Members of an Array Type

The members of an array type are specified in §10.7. For convenience, we repeat that specification here.
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

All the members inherited from class `Object`; the only method of `Object` that is not inherited is its `clone` method

The example:

```
class Test {
   public static void main(String[] args) {
      int[] ia = new int[3];
      int[] ib = new int[6];
      System.out.println(ia.getClass() == ib.getClass());
      System.out.println("ia has length=" + ia.length);
   }
```

```
}
```

produces the output:

```
true
ia has length=3
```

This example uses the method `getClass` inherited from class `Object` and the field `length`. The result of the comparison of the `Class` objects in the first `println` demonstrates that all arrays whose components are of type `int` are instances of the same array type, which is `int[]`.

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. First, context causes a name syntactically to fall into one of six categories: *PackageName*, *TypeName*, *ExpressionName*, *MethodName*, *PackageOrTypeName* or *AmbiguousName*. Second, a name that is initially classified by its context as an *AmbiguousName* is then reclassified by certain scoping rules to be a *PackageName*, *TypeName*, or *ExpressionName*. 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 use of context helps to minimize name conflicts between entities of different kinds. Such conflicts will be rare if the naming conventions described in §6.8 are followed. Nevertheless, conflicts may arise unintentionally as types developed by different programmers or different organizations evolve. For example, types, methods, and fields may have the same name. It is always possible to distinguish between a method and a field with the same name, since the context of a use always tells whether a method is intended.

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)
- In an `extends` clause in a class declaration (§8.1.3)
- In an `implements` clause in a class declaration (§8.1.4)
- In an `extends` clause in an interface declaration (§9.1.2)
- 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 (§8.4.4, §8.8.4, §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 (§14.19)
  - As the type in a class literal (§15.8.2)

- As a 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 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 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 this context:

- Before the "`(`" in a method invocation expression (§15.12)

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*

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, §14.19) or field declaration (§8.3) with that name, then the *AmbiguousName* is reclassified as an *ExpressionName*.

- Otherwise, if the *Identifier* appears within the scope (§6.3) of 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 top level class (§8) or interface type declaration (§9), then the *AmbiguousName* is reclassified as a *TypeName*.
- Otherwise, if a type of that name is declared in another compilation unit (§7.3) of the package (§7.1) of the compilation unit containing the *Identifier*, then the *AmbiguousName* is reclassified as a *TypeName*.
- Otherwise, if a type of that name is declared by exactly one type-import-on-demand declaration (§7.5.2) of the compilation unit containing the *Identifier*, then the *AmbiguousName* is reclassified as a *TypeName*.
- Otherwise, if a type of that name is declared by more than one type-import-on-demand declaration of the compilation unit containing the *Identifier*, then a compile-time error results.
- 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 there is a further choice:
    - 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 class or interface denoted by *TypeName*, this *AmbiguousName* is reclassified as an *ExpressionName*. Otherwise, a compile-time error results.
- 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 class or interface 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 class or interface denoted by *T*, then this *AmbiguousName* is reclassified as a *TypeName*. Otherwise, a compile-time error results.

As an example, consider the following contrived "library code":

```
package org.rpgpoet;
import java.util.Random;
interface Music { Random[] wizards = new Random[4]; }
```

and then consider this example code in another package:

```
package bazola;
class Gabriel {
   static int n = org.rpgpoet.Music.wizards.length;
}
```

First of all, the name `org.rpgpoet.Music.wizards.length` is classified as an *ExpressionName* because it functions as a *PostfixExpression*. Therefore, each of the names:

```
org.rpgpoet.Music.wizards
org.rpgpoet.Music
org.rpgpoet
org
```

is initially classified as an *AmbiguousName*. These are then reclassified:

- Assuming that there is no class or interface named `org` in any other compilation unit of package `bazola`, then the simple name `org` is reclassified as a *PackageName*.
- Next, assuming that there is no class or interface named `rpgpoet` in any compilation unit of package `org` (and we know that there is no such class or interface because package `org` has a subpackage named `rpgpoet`), the qualified name `org.rpgpoet` is reclassified as a *PackageName*.
- Next, because package `org.rpgpoet` has an interface type named `Music`, the qualified name `org.rpgpoet.Music` is reclassified as a *TypeName*.

Finally, because the name `org.rpgpoet.Music` is a *TypeName*, the qualified name `org.rpgpoet.Music.wizards` is reclassified as an *ExpressionName*.

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 accessible, as determined by the host system (§7.4.4), 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 accessible package or `Id` does not name an accessible 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 a declaration of a type with this name, or a compile-time error occurs. It is possible that the identifier occurs within the scope of more than one type with that name, in which case the type denoted by the name is determined as follows:

- If the simple type name occurs within the scope of a non-hidden local class declaration (§14.3) with that name, then the simple type name denotes that local class type.
- Otherwise, if the simple type name occurs within the scope of exactly one non-hidden member type (§8.5, §9.5), then the simple type name denotes that member type.
- Otherwise, if the simple type name occurs within the scope of more than one non-hidden member type, then the name is ambiguous as a type name; a compile-time error occurs.
- Otherwise, if a type with that name is declared in the current compilation unit (§7.3), either by a single-type-import declaration (§7.5.1) or by a declaration of a class or interface type (§7.6), then the simple type name denotes that type.
- Otherwise, if a type with that name is declared in another compilation unit (§7.3) of the package (§7.1) containing the identifier, then the identifier denotes that type.
- Otherwise, if a type of that name is declared by exactly one type-import-on-demand declaration (§7.5.2) of the compilation unit containing the identifier, then the simple type name denotes that type.
- Otherwise, if a type of that name is declared by more than one type-import-on-demand declaration of the compilation unit, then the name is ambiguous as a type name; a compile-time error occurs.
- Otherwise, the name is undefined as a type name; a compile-time error occurs.

This order for considering type declarations is designed to choose the most explicit of two or more applicable type declarations.

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 a 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 `QId`, or the member type named `Id` within `Q` is not accessible (§6.6), then a compile-time error occurs.

The example:

```
package wnj.test;
```

```
class Test {
   public static void main(String[] args) {
      java.util.Date date =
         new java.util.Date(System.currentTimeMillis());
      System.out.println(date.toLocaleString());
   }
}
```

produced the following output the first time it was run:

```
Sun Jan 21 22:56:29 1996
```

In this example the name `java.util.Date` must denote a type, so we first use the procedure recursively to determine if `java.util` is an accessible type or package, which it is, and then look to see if the type `Date` is accessible in this package.

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:

- If the *Identifier* appears within the scope (§6.3) of a non-hidden local variable declaration (§14.4) or non-hidden parameter declaration (§8.4.1, §8.8.1, §14.19) with that name, then the expression name denotes a variable, that is, that local variable or parameter.There is necessarily at most one such local variable or parameter. The type of the expression name is the declared type of the local variable or parameter.
- Otherwise, if the *Identifier* appears within a class declaration (§8):
  - If the *Identifier* appears within the scope (§6.3) of a non-hidden field declaration with that name, then there must be an enclosing type declaration *T* of which that field is a member. If there is not exactly one member of *T* that is a field with that name, then a compile-time error results.
  - Otherwise, if the single member field with that name is declared `final` (§8.3.1.2), then the expression name denotes the value of the field. The type of the expression name is the declared type of the field. If the *Identifier* appears in a context that requires a variable and not a value, then a compile-time error occurs.
  - Otherwise, the expression name denotes a variable, the single member field with that name. The type of the expression name is the field's declared type.

If the field is an instance variable (§8.3.1.1), the expression name must appear within the declaration of an instance method (§8.4), constructor (§8.8), 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 (§8.3.1.1, §12.4.2), then a compile-time error occurs.

- Otherwise, the identifier appears within an interface declaration (§9):
  - If the *Identifier* appears within the scope (§6.3) of a non-hidden field declaration with that name, then there must be an enclosing type declaration *T* of which that field is a member. If there is not exactly one member of *T* that is a field with that name, then a compile-time error results.
  - Otherwise, the expression name denotes the value of the single member field of that name. The type of the expression name is the declared type of the field. If the *Identifier* appears in a context that requires a variable and not a value, then a compile-time error occurs.

In the example:

```
class Test {
   static int v;
   static final int f = 3;
   public static void main(String[] args) {
      int i;
      i = 1;
      v = 2;
      f = 33;                          // compile-time error
      System.out.println(i + " " + v + " " + f);
           }

}
```

the names used as the left-hand-sides in the assignments to `i`, `v`, and `f` denote the local variable `i`, the field `v`, and the value of `f` (not the variable `f`, because `f` is a `final` variable). The example therefore produces an error at compile time because the last assignment does not have a variable as its left-hand side. If the erroneous assignment is removed, the modified code can be compiled and it will produce the output:

```
1 2 3
```

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 (§8), 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. 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.
- If *Q* is a type name that names an interface type (§9), 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. 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. The type of the expression *Q*`.`*Id* is the declared type of the field. 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 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 declared type of the field

The example:

```
class Point {
   int x, y;
   static int nPoints;
}

class Test {
   public static void main(String[] args) {
      int i = 0;
      i.x++;                    // compile-time error
      Point p = new Point();
      p.nPoints();              // compile-time error
   }
}
```

encounters two compile-time errors, because the `int` variable `i` has no members, and because `nPoints` is not a method of class `Point`.

6.5.7 Meaning of Method Names

A *MethodName* can appear only in a method invocation expression (§15.12). The meaning of a name classified as a *MethodName* is determined as follows.

6.5.7.1 *Simple Method Names*

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 method of a class or interface within whose declaration the *Identifier* appears. See §15.12 for further discussion of the interpretation of simple method names in method invocation expressions.

6.5.7.2 *Qualified Method Names*

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`*. See §15.12 for further discussion of the interpretation of qualified method names in method invocation expressions.

6.6 Qualified Names and Access Control

Qualified names are a means of access to members of packages and reference types; related means of access include field access expressions (§15.11) and method invocation expressions (§15.12). All three are syntactically similar in that a "`.`" token appears, preceded by some indication of a package, type, or expression having a type and followed by an *Identifier* that names a member of the package or type. These are collectively known as constructs for *qualified access*.

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. Access control applies to qualified access and to the invocation of constructors by class instance creation expressions (§15.9) and explicit constructor invocations (§8.8.5).

If access is permitted, then the accessed entity is said to be *accessible*.

Note that accessibility is a static property that can be determined at compile time; it depends only on types and declaration modifiers.

6.6.1 Determining Accessibility

- Whether a package is accessible is determined by the host system (§7.2).
- 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.
- 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 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 (§8) that encloses the declaration of the member.

- 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 *Qualified Access to a* protected *Member*

Let *C* be the class in which a protected member is declared. There must be a subclass *S* of *C*, which directly or indirectly encloses the use of the protected member. 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 *T*(. . .){. . .} or by a qualified class instance creation expression of the form *E*.new *T*(. . .){. . .}, where *E* is a *Primary* expression, then the access is permitted if and only if *S* is a subclass of *C*, or if *S* is a class nested within a subclass of *C*.
- Otherwise, if the access is by a simple class instance creation expression of the form new *T*(. . .) or by a qualified class instance creation expression of the form *E*.new *T*(. . .), where *E* is a *Primary* expression, then the access is not permitted. (A protected constructor can be accessed by a class instance creation expression only from within the package in which it is defined.)

6.6.3 An Example of Access Control

For examples of access control, consider the two compilation units:

```
package points;
class PointVec { Point[] vec; }
```

and:

```
package points;
public class Point {
   protected int x, y;
   public void move(int dx, int dy) { x += dx; y += dy; }
   public int getX() { return x; }
   public int getY() { return y; }
}
```

which declare two class types in the package `points`:

- The class type `PointVec` is not `public` and not part of the `public` interface of the package `points`, but rather can be used only by other classes in the package.
- The class type `Point` is declared `public` and is available to other packages. It is part of the `public` interface of the package `points`.
- The methods `move`, `getX`, and `getY` of the class `Point` are declared `public` and so are available to any code that uses an object of type `Point`.
- The fields `x` and `y` are declared `protected` and are accessible outside the package `points` only in subclasses of class `Point`, and only when they are fields of objects that are being implemented by the code that is accessing them.

See §6.6.7 for an example of how the `protected` access modifier limits access.

6.6.4 Example: Access to `public` and Non-`public` Classes

If a class lacks the `public` modifier, access to the class declaration is limited to the package in which it is declared (§6.6). In the example:

```
package points;
public class Point {
   public int x, y;
   public void move(int dx, int dy) { x += dx; y += dy; }
```

```
}
class PointList {
   Point next, prev;
}
```

two classes are declared in the compilation unit. The class `Point` is available outside the package `points`, while the class `PointList` is available for access only within the package. Thus a compilation unit in another package can access `points.Point`, either by using its fully qualified name:

```
package pointsUser;
class Test {
   public static void main(String[] args) {
      points.Point p = new points.Point();
      System.out.println(p.x + " " + p.y);
   }
}
```

or by using a single-type-import declaration (§7.5.1) that mentions the fully qualified name, so that the simple name may be used thereafter:

```
package pointsUser;
import points.Point;
class Test {
   public static void main(String[] args) {
      Point p = new Point();
      System.out.println(p.x + " " + p.y);
   }
}
```

However, this compilation unit cannot use or import `points.PointList`, which is not declared `public` and is therefore inaccessible outside package `points`.

6.6.5 Example: Default-Access Fields, Methods, and Constructors

If none of the access modifiers `public`, `protected`, or `private` are specified, a class member or constructor is accessible throughout the package that contains the declaration of the class in which the class member is declared, but the class member or constructor is not accessible in any other package. If a `public` class has a method or constructor with default access, then this method or constructor is not accessible to or inherited by a subclass declared outside this package.

For example, if we have:

```
package points;
public class Point {
   public int x, y;
   void move(int dx, int dy) { x += dx; y += dy; }
   public void moveAlso(int dx, int dy) { move(dx, dy); }
}
```

then a subclass in another package may declare an unrelated `move` method, with the same signature (§8.3.2) and return type. Because the original `move` method is not accessible from package `morepoints`, `super` may not be used:

```
package morepoints;
public class PlusPoint extends points.Point {
   public void move(int dx, int dy) {
      super.move(dx, dy);       // compile-time error
      moveAlso(dx, dy);
   }
}
```

Because move of `Point` is not overridden by `move` in `PlusPoint`, the method `moveAlso` in `Point` never calls the method move in `PlusPoint`.

Thus if you delete the `super.move` call from `PlusPoint` and execute the test program:

```
import points.Point;
import morepoints.PlusPoint;
class Test {
    public static void main(String[] args) {
       PlusPoint pp = new PlusPoint();
       pp.move(1, 1);
  }
}
```

it terminates normally. If move of `Point` were overridden by `move` in `PlusPoint`, then this program would recurse infinitely, until a `StackoverflowError` occurred.

6.6.6 Example: `public` Fields, Methods, and Constructors

A `public` class member or constructor is accessible throughout the package where it is declared and from any other package that has access to the package in which it is declared (§7.4.3). For example, in the compilation unit:

```
package points;
public class Point {
   int x, y;
   public void move(int dx, int dy) {
      x += dx; y += dy;
      moves++;
           }
   public static int moves = 0;
}
```

the public class Point has as public members the move method and the moves field. These public members are accessible to any other package that has access to package points. The fields x and y are not public and therefore are accessible only from within the package points.

6.6.7 Example: protected Fields, Methods, and Constructors

Consider this example, where the points package declares:

```
package points;
public class Point {
   protected int x, y;
   void warp(threePoint.Point3d a) {
      if (a.z > 0)       // compile-time error: cannot access a.z
         a.delta(this);
           }
}
```

and the threePoint package declares:

```
package threePoint;
import points.Point;
public class Point3d extends Point {
   protected int z;
   public void delta(Point p) {
      p.x += this.x;     // compile-time error: cannot access p.x
      p.y += this.y;     // compile-time error: cannot access p.y
           }
   public void delta3d(Point3d q) {
      q.x += this.x;
      q.y += this.y;
      q.z += this.z;
           }
```

```
}
```

which defines a class Point3d. A compile-time error occurs in the method delta here: it cannot access the protected members x and y of its parameter p, because while Point3d (the class in which the references to fields x and y occur) is a subclass of Point (the class in which x and y are declared), it is not involved in the implementation of a Point (the type of the parameter p). The method delta3d can access the protected members of its parameter q, because the class Point3d is a subclass of Point and is involved in the implementation of a Point3d.

The method delta could try to cast (§5.5, §15.16) its parameter to be a Point3d, but this cast would fail, causing an exception, if the class of p at run time were not Point3d.

A compile-time error also occurs in the method warp: it cannot access the protected member z of its parameter a, because while the class Point (the class in which the reference to field z occurs) is involved in the implementation of a Point3d (the type of the parameter a), it is not a subclass of Point3d (the class in which z is declared).

6.6.8 Example: private Fields, Methods, and Constructors

A private class member or constructor is accessible only within the class body in which the member is declared and is not inherited by subclasses. In the example:

```
class Point {
    Point() { setMasterID(); }
    int x, y;
    private int ID;
    private static int masterID = 0;
    private void setMasterID() { ID = masterID++; }
}
```

the private members ID, masterID, and setMasterID may be used only within the body of class Point. They may not be accessed by qualified names, field access expressions, or method invocation expressions outside the body of the declaration of Point.

See §8.8.8 for an example that uses a private constructor.

6.7 Fully Qualified Names and Canonical Names

Every package, top level class, member class, top level interface, member interface, and primitive type has a fully qualified name. An array type has a fully qualified name if and only if its element 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.
- A member class or member interface *M* of another class *C* has a fully qualified name 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*.
- The fully qualified name of an array type consists of the fully qualified name of the component type of the array type followed by "`[]`".

Examples:

- The fully qualified name of the type `long` is "`long`".
- The fully qualified name of the package `java.lang` is "`java.lang`" because it is subpackage `lang` of package `java`.
- The fully qualified name of the class `Object`, which is defined in the package `java.lang`, is "`java.lang.Object`".
- The fully qualified name of the interface `Enumeration`, which is defined in the package `java.util`, is "`java.util.Enumeration`".
- The fully qualified name of the type "array of `double`" is "`double[]`".
- The fully qualified name of the type "array of array of array of array of `String`" is "`java.lang.String[][][][]`".

In the example:

```
package points;

class Point { int x, y; }
class PointVec {
   Point[] vec;
}
```

the fully qualified name of the type `Point` is "`points.Point`"; the fully qualified name of the type `PointVec` is "`points.PointVec`"; and the fully qualified name of the type of the field `vec` of class `PointVec` is "`points.Point[]`".

Every package, top level class, member class, top level interface, member interface, and primitive type has a *canonical name*. An array type has a canonical name if and only if its element type has a canonical name. A member class or member interface *M* declared in another class *C* has a canonical name 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*. For every package, top level class, top level interface and primitive type, the canonical name is the same as the fully qualified name. The canonical name of an array type is defined only when the component type of the array 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 "`[]`".

The difference between a fully qualified name and a canonical name can be seen in examples such as:

```
package p;
class O1 { class I{}}
class O2 extends O1{};
```

`p.O1.I` and `p.O2.I` are both fully qualified names that denote the same class, but only `p.O1.I` is its canonical name.

6.8 Naming Conventions

The class libraries of the Java platform attempt to use, whenever possible, names chosen according to the conventions presented here. These conventions help to make code more readable and avoid certain kinds of name conflicts.

We recommend these conventions for use in all programs written in the Java programming language. However, these conventions should not be followed slavishly if long-held conventional usage dictates otherwise. So, for example, the `sin` and `cos` methods of the class `java.lang.Math` have mathematically conventional names, even though these method names flout the convention suggested here because they are short and are not verbs.

6.8.1 Package Names

Names of packages that are to be made widely available should be formed as described in §7.7. Such names are always qualified names whose first identifier

consists of two or three lowercase letters that name an Internet domain, such as `com`, `edu`, `gov`, `mil`, `net`, `org`, or a two-letter ISO country code such as `uk` or `jp`. Here are examples of hypothetical unique names that might be formed under this convention:

```
com.JavaSoft.jag.Oak
org.npr.pledge.driver

uk.ac.city.rugby.game
```

Names of packages intended only for local use should have a first identifier that begins with a lowercase letter, but that first identifier specifically should not be the identifier `java`; package names that start with the identifier `java` are reserved by Sun for naming Java platform packages.

When package names occur in expressions:

- If a package name is hidden by a field declaration, then `import` declarations (§7.5) can usually be used to make available the type names declared in that package.
- If a package name is hidden by a declaration of a parameter or local variable, then the name of the parameter or local variable can be changed without affecting other code.

The first component of a package name is normally not easily mistaken for a type name, as a type name normally begins with a single uppercase letter. (The Java programming language does not actually rely on case distinctions to determine whether a name is a package name or a type name.)

6.8.2 Class and Interface Type Names

Names of class types should be descriptive nouns or noun phrases, not overly long, in mixed case with the first letter of each word capitalized. For example:

```
ClassLoader
SecurityManager
Thread
Dictionary

BufferedInputStream
```

Likewise, names of interface types should be short and descriptive, not overly long, in mixed case with the first letter of each word capitalized. The name may be a descriptive noun or noun phrase, which is appropriate when an interface is used as if it were an abstract superclass, such as interfaces `java.io.DataInput` and `java.io.DataOutput`; or it may be an adjective describing a behavior, as for the interfaces `java.lang.Runnable` and `java.lang.Cloneable`.

Hiding involving class and interface type names is rare. Names of fields, parameters, and local variables normally do not hide type names because they conventionally begin with a lowercase letter whereas type names conventionally begin with an uppercase letter.

6.8.3 Method Names

Method names should be verbs or verb phrases, in mixed case, with the first letter lowercase and the first letter of any subsequent words capitalized. Here are some additional specific conventions for method names:

- Methods to `get` and `set` an attribute that might be thought of as a variable *V* should be named `get`*V* and `set`*V*. An example is the methods `getPriority` and `setPriority` of class `java.lang.Thread`.
- A method that returns the length of something should be named `length`, as in class `java.lang.String`.
- A method that tests a `boolean` condition *V* about an object should be named `is`*V*. An example is the method `isInterrupted` of class `java.lang.Thread`.
- A method that converts its object to a particular format *F* should be named `to`*F*. Examples are the method `toString` of class `java.lang.Object` and the methods `toLocaleString` and `toGMTString` of class `java.util.Date`.

Whenever possible and appropriate, basing the names of methods in a new class on names in an existing class that is similar, especially a class from the Java Application Programming Interface classes, will make it easier to use.

Method names cannot hide or be hidden by other names (§6.5.7).

6.8.4 Field Names

Names of fields that are not `final` should be in mixed case with a lowercase first letter and the first letters of subsequent words capitalized. Note that well-designed classes have very few `public` or `protected` fields, except for fields that are constants (`final static` fields) (§6.8.5).

Fields should have names that are nouns, noun phrases, or abbreviations for nouns. Examples of this convention are the fields `buf`, `pos`, and `count` of the class `java.io.ByteArrayInputStream` and the field `bytesTransferred` of the class `java.io.InterruptedIOException`.

Hiding involving field names is rare.

- If a field name hides a package name, then an `import` declaration (§7.5) can usually be used to make available the type names declared in that package.
- If a field name hides a type name, then a fully qualified name for the type can be used unless the type name denotes a local class (§14.3).
- Field names cannot hide method names.
- If a field name is hidden by a declaration of a parameter or local variable, then the name of the parameter or local variable can be changed without affecting other code.

6.8.5 Constant Names

The names of constants in interface types should be, and `final` variables of class types may conventionally be, a sequence of one or more words, acronyms, or abbreviations, all uppercase, with components separated by underscore "_" characters. Constant names should be descriptive and not unnecessarily abbreviated. Conventionally they may be any appropriate part of speech. Examples of names for constants include `MIN_VALUE`, `MAX_VALUE`, `MIN_RADIX`, and `MAX_RADIX` of the class `java.lang.Character`.

A group of constants that represent alternative values of a set, or, less frequently, masking bits in an integer value, are sometimes usefully specified with a common acronym as a name prefix, as in:

```
interface ProcessStates {
   int PS_RUNNING = 0;
   int PS_SUSPENDED = 1;
}
```

Hiding involving constant names is rare:

- Constant names should be longer than three letters, so that they do not hide the initial component of a unique package name.
- Constant names normally have no lowercase letters, so they will not normally hide names of packages, types, or fields, whose names normally contain at least one lowercase letter.
- Constant names cannot hide method names, because they are distinguished syntactically.

6.8.6 Local Variable and Parameter Names

Local variable and parameter names should be short, yet meaningful. They are often short sequences of lowercase letters that are not words. For example:

- Acronyms, that is the first letter of a series of words, as in `cp` for a variable holding a reference to a `ColoredPoint`
- Abbreviations, as in `buf` holding a pointer to a `buffer` of some kind
- Mnemonic terms, organized in some way to aid memory and understanding, typically by using a set of local variables with conventional names patterned after the names of parameters to widely used classes. For example:

`in` and `out`, whenever some kind of input and output are involved, patterned after the fields of `java.lang.System`

`off` and `len`, whenever an offset and length are involved, patterned after the parameters to the `read` and `write` methods of the interfaces `DataInput` and `DataOutput` of `java.io`

One-character local variable or parameter names should be avoided, except for temporary and looping variables, or where a variable holds an undistinguished value of a type. Conventional one-character names are:

- `b` for a `byte`
- `c` for a `char`
- `d` for a `double`
- `e` for an `Exception`
- `f` for a `float`
- `i`, `j`, and `k` for integers
- `l` for a `long`
-
- `o` for an `Object`
- `s` for a `String`
- `v` for an arbitrary value of some type

Local variable or parameter names that consist of only two or three lowercase letters should not conflict with the initial country codes and domain names that are the first component of unique package names (§7.7).

What's in a name? That which we call a rose
By any other name would smell as sweet.
—William Shakespeare, *Romeo and Juliet* (c. 1594), Act II, scene ii

Rose is a rose is a rose is a rose.
—Gertrude Stein, *Sacred Emily* (1913), in *Geographies and Plays*

. . . stat rosa pristina nomine, nomina nuda tenemus.
—Bernard of Morlay, *De contemptu mundi* (12th century),
quoted in Umberto Eco, *The Name of the Rose* (1980)

Rose, Rose, bo-Bose,
Banana-fana fo-Fose,
Fee, fie, mo-Mose—
—Rose!
—Lincoln Chase and Shirley Elliston, *The Name Game*
(#3 pop single in the U.S., January 1965),
as applied to the name "Rose"

CHAPTER 7

Packages

Good things come in small packages.
—Traditional proverb

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 (§7.2.1) or in a database (§7.2.2). Packages that are stored in a file system 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 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 (§7.7). 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 subpackages and all the top level class (§8) and top level interface (§9) types declared in all the compilation units (§7.3) of the package.

For example, in the Java Application Programming Interface :

- The package `java` has subpackages `awt`, `applet`, `io`, `lang`, `net`, and `util`, but no compilation units.
- The package `java.awt` has a subpackage named `image`, as well as a number of compilation units containing declarations of class and interface types.

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.

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

Here are some examples:

- Because the package `java.awt` has a subpackage `image`, it cannot (and does not) contain a declaration of a class or interface type named `image`.
- If there is a package named `mouse` and a member type `Button` in that package (which then might be referred to as `mouse.Button`), then there cannot be any package with the fully qualified name `mouse.Button` or `mouse.Button.Click`.
- If `com.Sun.java.jag` is the fully qualified name of a type, then there cannot be any package whose fully qualified name is either `com.Sun.java.jag` or `com.Sun.java.jag.scrabble`.

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`. For example, 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 determines how packages, compilation units, and subpackages are created and stored; which top level packages are in scope in a particular compilation; and which packages are observable.

The packages may be stored in a local file system in simple implementations of the Java platform. Other implementations may use a distributed file system or some form of database to store source and/or binary code.

7.2.1 Storing Packages in a File System

As an extremely simple example, all the packages and source and binary code on a system might be stored in a single directory and its subdirectories. Each immediate subdirectory of this directory would represent a top level package, that is, one whose fully qualified name consists of a single simple name. The directory might contain the following immediate subdirectories:

```
com
gls
jag
java
wnj
```

where directory `java` would contain the Java Application Programming Interface packages; the directories `jag`, `gls`, and `wnj` might contain packages that the three authors of this specification created for their personal use and to share with each other within this small group; and the directory `com` would contain packages procured from companies that used the conventions described in §7.7 to generate unique names for their packages.

Continuing the example, the directory `java` would contain, among others, the following subdirectories:

```
applet
awt
io
lang
net
util
```

corresponding to the packages `java.applet`, `java.awt`, `java.io`, `java.lang`, `java.net`, and `java.util` that are defined as part of the Java Application Programming Interface.

Still continuing the example, if we were to look inside the directory `util`, we might see the following files :

```
BitSet.java                    Observable.java
BitSet.class                   Observable.class
Date.java                      Observer.java
Date.class                     Observer.class
Dictionary.java                Properties.java
Dictionary.class               Properties.class
EmptyStackException.java       Random.java
EmptyStackException.class      Random.class
Enumeration.java               Stack.java
Enumeration.class              Stack.class
Hashtable.java                 StringTokenizer.java
Hashtable.class                StringTokenizer.class
```

```
NoSuchElementException.java     Vector.java
NoSuchElementException.class    Vector.class
```

where each of the `.java` files contains the source for a compilation unit (§7.3) that contains the definition of a class or interface whose binary compiled form is contained in the corresponding `.class` file.

Under this simple organization of packages, an implementation of the Java platform would transform a package name into a pathname by concatenating the components of the package name, placing a file name separator (directory indicator) between adjacent components. For example, if this simple organization were used on a UNIX system, where the file name separator is `/`, the package name:

```
jag.scrabble.board
```

would be transformed into the directory name:

```
jag/scrabble/board
```

and:

```
com.Sun.sunsoft.DOE
```

would be transformed to the directory name:

```
com/Sun/sunsoft/DOE
```

A package name component or class name might contain a character that cannot correctly appear in a host file system's ordinary directory name, such as a Unicode character on a system that allows only ASCII characters in file names. As a convention, the character can be escaped by using, say, the `@` character followed by four hexadecimal digits giving the numeric value of the character, as in the `\u`*xxxx* escape (§3.3), so that the package name:

```
children.activities.crafts.papierM\u00e2ch\u00e9
```

which can also be written using full Unicode as:

```
children.activities.crafts.papierMâché
```

might be mapped to the directory name:

```
children/activities/crafts/papierM@00e2ch@00e9
```

If the `@` character is not a valid character in a file name for some given host file system, then some other character that is not valid in a identifier could be used instead.

7.2.2 Storing Packages in a Database

A host system may store packages and their compilation units and subpackages in a database.

Such a database must not impose the optional restrictions (§7.6) on compilation units in file-based implementations. For example, a system that uses a database to store packages may not enforce a maximum of one `public` class or interface per compilation unit.

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

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.

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 to be referred to using their simple names
- Top level type declarations (§7.6) of class and interface types

Which compilation units are *observable* is determined by the host system. However, all the compilation units of the package `java` and its subpackage `lang`

should always be observable. The observability of a compilation unit influences the observability and accessibility of its package and the accessibility of the type declarations it contains.

Every compilation unit automatically and implicitly imports every `public` type name declared in the predefined package `java.lang`, so that the names of all those types are available as simple names, as described in §7.5.3.

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:
`package` *PackageName* `;`

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

7.4.2 Unnamed Packages

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

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.
As an example, the compilation unit:

```
class FirstCall {
   public static void main(String[] args) {
      System.out.println("Mr. Watson, come here. "
                         + "I want you.");
   }
}
```

defines a very simple compilation unit as part of an unnamed package.

An implementation of the Java 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.

In implementations of the Java platform that use a hierarchical file system for storing packages, one typical strategy is to associate an unnamed package with each directory; only one unnamed package is observable at a time, namely the one that is associated with the "current working directory." The precise meaning of "current working directory" depends on the host system.

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

7.4.3 Accessibility 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.

A package is *accessible* if and only if it is observable.

7.4.4 Scope of a Package Name

The scope (§6.3, §6.5) of a top level package is determined by conventions of the host system. The package `java` is always in scope. Subpackage names are never in scope.

Package names never hide other names.

7.5 Import Declarations

An *import declaration* allows a named type 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 is to use a fully qualified name (§6.7).

ImportDeclaration:
SingleTypeImportDeclaration
TypeImportOnDemandDeclaration

A single-type-import declaration (§7.5.1) imports a single named type, by mentioning its canonical name. A type-import-on-demand declaration (§7.5.2) imports all the `public` types of a named type or package as needed.

The scope of a type imported by a single-type-import declaration (§7.5.1) or 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.

An `import` declaration makes types available by their simple names only within the compilation unit that actually contains the `import` declaration. The scope of the name(s) it introduces specifically does not include the `package` statement, other `import` declarations in the current compilation unit, or other compilation units in the same package. Please see §7.5.4 for an illustrative example.

7.5.1 Single-Type-Import Declaration

A *single-type-import declaration* imports a single type by giving its canonical name, 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 or interface type; a compile-time error occurs if the named type does not exist. The named type must be accessible (§6.6) or a compile-time error occurs.

A single-type-import declaration *d* that imports a type named *n* hides the declarations of any top level type named *n* declared in another compilation unit of the package in which *d* appears, and of any type named *n* imported by a type-import-on-demand declaration.

The example:

```
import java.util.Vector;
```

causes the simple name `Vector` to be available within the class and interface declarations in a compilation unit. Thus, the simple name `Vector` refers to the type `Vector` in the package `java.util` in all places where it is not hidden (§6.3.1) by a declaration of a field, parameter, local variable, or nested type declaration with the same name.

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 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), then a compile-time error occurs.

So the sample program:

```
import java.util.Vector;
class Vector { Object[] vec; }
```

causes a compile-time error because of the duplicate declaration of `Vector`, as does:

```
import java.util.Vector;
import myVector.Vector;
```

where `myVector` is a package containing the compilation unit:

```
package myVector;
public class Vector { Object[] vec; }
```

The compiler keeps track of types by their binary names (§13.1).

Note that an import statement cannot import a subpackage, only a type. For example, it does not work to try to import `java.util` and then use the name `util.Random` to refer to the type `java.util.Random`:

```
import java.util;                   // incorrect: compile-time error
class Test { util.Random generator; }
```

7.5.2 Type-Import-on-Demand Declaration

A *type-import-on-demand declaration* allows all `public` types declared in the type or package named by a canonical name to be imported as needed.

TypeImportOnDemandDeclaration:
`import` *PackageOrTypeName* `. * ;`

It is a compile-time error for a type-import-on-demand declaration to name a type or package that 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; the effect is as if there were exactly one such declaration. 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.

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

The example:

```
import java.util.*;
```

causes the simple names of all `public` types declared in the package `java.util` to be available within the class and interface declarations of the compilation unit. Thus, the simple name `Vector` refers to the type `Vector` in the package

java.util in all places where it is not hidden (§6.3) by a single-type-import declaration of a type whose simple name is Vector; by a type named Vector and declared in the package to which the compilation unit belongs; or by a declaration of a field, parameter, or local variable named Vector or any nested classes or interfaces.(It would be unusual for any of these conditions to occur.)

7.5.3 Automatic Imports

Each compilation unit automatically imports all of the public type names declared in the predefined package java.lang, as if the declaration:

```
import java.lang.*;
```

appeared at the beginning of each compilation unit, immediately following any package statement.

7.5.4 A Strange Example

Package names and type names are usually different under the naming conventions described in §6.8. Nevertheless, in a contrived example where there is an unconventionally-named package Vector, which declares a public class named Mosquito:

```
package Vector;

public class Mosquito { int capacity; }
```

and then the compilation unit:

```
package strange.example;

import java.util.Vector;

import Vector.Mosquito;

class Test {
   public static void main(String[] args) {
      System.out.println(new Vector().getClass());
      System.out.println(new Mosquito().getClass());
   }
}
```

the single-type-import declaration (§7.5.1) importing class Vector from package java.util does not prevent the package name Vector from appearing and being correctly recognized in subsequent import declarations. The example compiles and produces the output:

```
class java.util.Vector
class Vector.Mosquito
```

7.6 Top level Type Declarations

A *top level type declaration* declares a top level class type (§8) or a top level interface type (§9):

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).

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

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*.

Thus in the example:

```
package wnj.points;

class Point { int x, y; }
```

the fully qualified name of class `Point` is `wnj.points.Point`.

An implementation of the Java 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.

For example, if a compilation unit contains the single-type-import declaration (§7.5.1):

```
import java.util.Vector;
```

then within that compilation unit the simple name `Vector` and the fully qualified name `java.util.Vector` refer to the same type.

When packages are stored in a file system (§7.2.1), 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).

This restriction implies that there must be at most one such type per compilation unit. This restriction makes it easy for a compiler for the Java programming language or an implementation of the Java virtual machine to find a named class within a package; for example, the source code for a `public` type `wet.sprocket.Toad` would be found in a file `Toad.java` in the directory `wet/sprocket`, and the corresponding object code would be found in the file `Toad.class` in the same directory.

When packages are stored in a database (§7.2.2), the host system must not impose such restrictions.

In practice, many programmers choose to put each class or interface type in its own compilation unit, whether or not it is `public` or is referred to by code in other compilation units.

A compile-time error occurs 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 (§7.6).

A compile-time error occurs 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.

In the example:

```
class Point { int x, y; }
```

the class `Point` is declared in a compilation unit with no `package` statement, and thus `Point` is its fully qualified name, whereas in the example:

```
package vista;
class Point { int x, y; }
```

the fully qualified name of the class `Point` is `vista.Point`. (The package name `vista` is suitable for local or personal use; if the package were intended to be widely distributed, it would be better to give it a unique package name (§7.7).)

In the example:

```
package test;

import java.util.Vector;

class Point {
   int x, y;
}

interface Point {                    // compile-time error #1
   int getR();
   int getTheta();
}

class Vector { Point[] pts; }        // compile-time error #2
```

the first compile-time error is caused by the duplicate declaration of the name `Point` as both a `class` and an `interface` in the same package. A second error detected at compile time is the attempt to declare the name `Vector` both by a class type declaration and by a single-type-import declaration.

Note, however, that it is not an error for the name of a class to also to name a type that otherwise might be imported by a type-import-on-demand declaration (§7.5.2) in the compilation unit (§7.3) containing the class declaration. In the example:

```
package test;

import java.util.*;

class Vector { Point[] pts; }        // not a compile-time error
```

the declaration of the class `Vector` is permitted even though there is also a class `java.util.Vector`. Within this compilation unit, the simple name `Vector` refers to the class `test.Vector`, not to `java.util.Vector` (which can still be referred to by code within the compilation unit, but only by its fully qualified name).

As another example, the compilation unit:

```
package points;

class Point {
   int x, y;                         // coordinates
   PointColor color;                 // color of this point
   Point next;                       // next point with this color
   static int nPoints;
}

class PointColor {
   Point first;                      // first point with this color
   PointColor(int color) {
      this.color = color;
   }
   private int color;                // color components
}
```

defines two classes that use each other in the declarations of their class members. Because the class types `Point` and `PointColor` have the all the type declarations in package `points`, including all those in the current compilation unit, as their scope, this example compiles correctly—that is, forward reference is not a problem

It is a compile time error if a top level type declaration contains any one of the following access modifiers: `protected`, `private` or `static`.

7.7 Unique Package Names

Did I ever tell you that Mrs. McCave
Had twenty-three sons and she named them all "Dave"?
Well, she did. And that wasn't a smart thing to do. . . .
—Dr. Seuss (Theodore Geisel), *Too Many Daves* (1961)

Developers should take steps to avoid the possibility of two published packages having the same name by choosing *unique package names* for packages that are widely distributed. This allows packages to be easily and automatically installed and catalogued. This section specifies a suggested convention for generating such unique package names. Implementations of the Java platform are encouraged to provide automatic support for converting a set of packages from local and casual package names to the unique name format described here.

If unique package names are not used, then package name conflicts may arise far from the point of creation of either of the conflicting packages. This may create a situation that is difficult or impossible for the user or programmer to resolve. The class `ClassLoader` can be used to isolate packages with the same name from each other in those cases where the packages will have constrained interactions, but not in a way that is transparent to a naïve program.

You form a unique package name by first having (or belonging to an organization that has) an Internet domain name, such as `Sun.com`. You then reverse this name, component by component, to obtain, in this example, `com.Sun`, and use this as a prefix for your package names, using a convention developed within your organization to further administer package names.

In some cases, the internet domain name may not be a valid package name. Here are some suggested conventions for dealing with these situations:

- If the domain name contains a hyphen, or any other special character not allowed in an identifier (§3.8), convert it into an underscore.
- If any of the resulting package name components are keywords (§3.9) then append underscore to them.
- If any of the resulting package name components start with a digit, or any other character that is not allowed as an initial character of an identifier, have an underscore prefixed to the component.

Such a convention might specify that certain directory name components be division, department, project, machine, or login names. Some possible examples:

```
com.Sun.sunsoft.DOE
com.Sun.java.jag.scrabble
com.Apple.quicktime.v2
edu.cmu.cs.bovik.cheese
gov.whitehouse.socks.mousefinder
```

The first component of a unique package name is always written in all-lowercase ASCII letters and should be one of the top level domain names, currently `com`, `edu`, `gov`, `mil`, `net`, `org`, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. For more information, refer to the documents stored at `ftp://rs.internic.net/rfc`, for example, `rfc920.txt` and `rfc1032.txt`.

The name of a package is not meant to imply anything about where the package is stored within the Internet; for example, a package named `edu.cmu.cs.bovik.cheese` is not necessarily obtainable from Internet address `cmu.edu` or from `cs.cmu.edu` or from `bovik.cs.cmu.edu`. The suggested convention for generating unique package names is merely a way to piggyback a package naming convention on top of an existing, widely known unique name registry instead of having to create a separate registry for package names.

Brown paper packages tied up with strings,
These are a few of my favorite things.
—Oscar Hammerstein II, *My Favorite Things* (1959)

CHAPTER 8

Classes

class 1. The noun *class* derives from
Medieval French and French *classe* from Latin *classis*,
probably originally a summons,
hence a summoned collection of persons,
a group liable to be summoned:
perhaps for *callassis* from *calare*,
to call, hence to summon.
—Eric Partridge
Origins: A Short Etymological Dictionary of Modern English

CLASS declarations define new reference types and describe how they are implemented (§8.1).

A *nested class* is any class whose declaration occurs within the body of another class or interface. A *top level class* is a class that is not a nested class.

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` (§8.1.4.1) 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` (§8.1.4.2), 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 (§8.1.6) and may implement interfaces (§8.1.7).

The body of a class declares members (fields and methods and nested classes and interfaces), instance and static initializers, and constructors (§8.1.8). The scope of the name of a member is the entire 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 (§8.1.5).

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 the 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 (§14.18), thus allowing its activities to be synchronized with those of other threads (§17).

Method names may be overloaded (§8.4.8).

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 when it is first loaded (§12.4).

Constructors (§8.9) 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 (§8.9.8).

8.1 Class Declaration

A *class declaration* specifies a new named reference type:

ClassDeclaration:
ClassModifiers^{opt} `class` *Identifier* *Super*^{opt} *Interfaces*^{opt} *ClassBody*

The *Identifier* in a class declaration specifies the name of the class. A compile time error occurs 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
`public` `protected` `private`
`abstract` `static` `final` `strictfp`

Not all modifiers are applicable to all kinds of class declarations. The access modifier `public` pertains only to top level classes (§7.6) and to member classes (§8.5, §9.5), and is discussed in §6.6, §8.5 and §9.5. The access modifiers `protected` and `private` pertain only to member classes within a directly enclosing class declaration (§8.5) and are discussed in §8.5.1. The access modifier `static` pertains only to member classes (§8.5, §9.5). A compile-time error occurs if the same modifier appears more than once in a class declaration.

If two or more class modifiers appear in a class declaration, then it is customary, though not required, that they appear in the order consistent with that shown above in the production for *ClassModifier*.

8.1.1.1 `abstract` *Classes*

An `abstract` class is a class that is incomplete, or to be considered incomplete. Only `abstract` classes may have `abstract` methods (§8.4.3.1, §9.4), that is, methods that are declared but not yet implemented. If a class that is not `abstract` contains an `abstract` method, then a compile-time error occurs. 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 declared an `abstract` method that has not been implemented in C or any of its superclasses.

- A direct superinterface (§8.1.7) of C declares or inherits a method (which is therefore necessarily `abstract`) and C neither declares nor inherits a method that implements it.

In the example:

```
abstract class Point {
   int x = 1, y = 1;
   void move(int dx, int dy) {
      x += dx;
      y += dy;
      alert();
   }
   abstract void alert();
}

abstract class ColoredPoint extends Point {
   int color;
}

class SimplePoint extends Point {
   void alert() { }
}
```

a class `Point` is declared that must be declared `abstract`, because it contains a declaration of an `abstract` method named `alert`. The subclass of `Point` named `ColoredPoint` inherits the `abstract` method `alert`, so it must also be declared `abstract`. On the other hand, the subclass of `Point` named `SimplePoint` provides an implementation of `alert`, so it need not be `abstract`.

A compile-time error occurs if an attempt is made to create an instance of an `abstract` class using a class instance creation expression (§15.9).

Thus, continuing the example just shown, the statement:

```
Point p = new Point();
```

would result in a compile-time error; the class `Point` cannot be instantiated because it is `abstract`. However, a `Point` variable could correctly be initialized with a reference to any subclass of `Point`, and the class `SimplePoint` is not `abstract`, so the statement:

```
Point p = new SimplePoint();
```

would be correct.

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. Thus, in the example just given, instantiation of a `SimplePoint` causes the default constructor and field initializers for `x` and `y` of `Point` to be executed.

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.

This situation can occur if the class would have as members two `abstract` methods that have the same method signature (§8.4.2) but different return types. As an example, the declarations:

```
interface Colorable { void setColor(int color); }
abstract class Colored implements Colorable {
   abstract int setColor(int color);
}
```

result in a compile-time error: it would be impossible for any subclass of class `Colored` to provide an implementation of a method named `setColor`, taking one argument of type `int`, that can satisfy both `abstract` method specifications, because the one in interface `Colorable` requires the same method to return no value, while the one in class `Colored` requires the same method to return a value of type `int` (§8.4).

A class type should be declared `abstract` only if the intent is that subclasses can be created to complete the implementation. If the intent is simply to prevent instantiation of a class, the proper way to express this is to declare a constructor (§8.9.11) of no arguments, make it `private`, never invoke it, and declare no other constructors. A class of this form usually contains class methods and variables. The class `Math` is an example of a class that cannot be instantiated; its declaration looks like this:

```
public final class Math {
   private Math() { }     // never instantiate this class
            . . . declarations of class variables and 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. A compile-time error occurs if the name of a `final` class appears in the `extends` clause (§8.1.6) of another `class` declaration; this implies that a `final` class cannot have any subclasses. A compile-time error occurs if a class is declared both `final` and `abstract`, because the implementation of such a class could never be completed (§8.1.4.1).

Because a `final` class never has any subclasses, the methods of a `final` class are never overridden (§8.4.6.1).

8.1.1.3 `strictfp` *Classes*

The effect of the `strictfp` modifier is to make all `float` or `double` expressions within the class declaration 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`.

Note also that all `float` or `double` expressions within all variable initializers, instance initializers, static initializers and constructors of the class will also be explicitly FP-strict.

8.1.2 Inner Classes and Enclosing Instances

An *inner class* is a nested class that is not explicitly or implicitly declared `static`. Inner classes may not declare static initializers (§8.7) or member interfaces. Inner classes may not declare static members, unless they are compile time constant fields (§15.28).

To illustrate these rules, consider the example below:

```
class HasStatic{
   static int j = 100;
}

class Outer{
   class Inner extends HasStatic{
      static final x = 3; // ok - compile time constant
      static int y = 4; // compile-time error
   } // an inner class
   static class NestedButNotInner{
      static int z = 5; // ok
   }// not an inner class
   interface NeverInner{} // interfaces are never inner
}
```

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 always 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 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 (§8.9.5).

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 con-

text. 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 *nth lexically enclosing class of a class C* if it is the immediately enclosing class of the $n-1$ st 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 *nth lexically enclosing instance of an instance i* if it is the immediately enclosing instance of the $n-1$ st 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 (§4.5.4) 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*.

Furthermore, for every superclass *S* of *C* which is itself a direct inner class of a class *S0*, there is an instance of *S0* 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`, and must be definitely assigned before the body of the inner class.

Inner classes include local (§14.3), anonymous (§15.9.5) and non-static member classes (§8.5). Here are some examples:

```
class Outer {
   int i = 100;

   static void classMethod() {
      final int l = 200;
      class LocalInStaticContext{
         int k = i; // compile-time error
         int m = l; // ok
      }
   }

   void foo() {
      class Local { // a local class
```

```
            int j = i;
        }

    }
}
```

The declaration of class `LocalInStaticContext` occurs in a static context - within the static method `classMethod`. Instance variables of class `Outer` are not available within the body of a static method. In particular, instance variables of `Outer` are not available inside the body of `LocalInStaticContext`. However, local variables from the surrounding method may be referred to without error (provided they are marked `final`).

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 is defined with respect to an enclosing instance of that class So, for example, the class `Local` above has an enclosing instance of class `Outer`. As a further example:

```
class WithDeepNesting{
   boolean toBe;

   WithDeepNesting(boolean b) { toBe = b;}

   class Nested {
      boolean theQuestion;
      class DeeplyNested {
         DeeplyNested(){
            theQuestion = toBe || ~toBe;
         }
      }
   }
}
```

Here, every instance of `WithDeepNesting.Nested.DeeplyNested` has an enclosing instance of class `WithDeepNesting.Nested` (its immediately enclosing instance) and an enclosing instance of class `WithDeepNesting` (its 2nd lexically enclosing instance).

8.1.3 Superclasses and Subclasses

The optional `extends` clause in a class declaration specifies the *direct superclass* of the current class. A class is said to be a *direct subclass* of the class it extends. The direct superclass is the class from whose implementation the implementation of the current class is derived. The `extends` clause must not appear in the defini-

tion of the class `Object` , because it is the primordial class and has no direct superclass. If the class declaration for any other class has no `extends` clause, then the class has the class `Object` as its implicit direct superclass.

Super:
`extends` *ClassType*

The following is repeated from §4.3 to make the presentation here clearer:

ClassType:
TypeName

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` (§8.1.4.2), then a compile-time error occurs; `final` classes are not allowed to have subclasses.

In the example:

```
class Point { int x, y; }
final class ColoredPoint extends Point { int color; }
class Colored3DPoint extends ColoredPoint { int z; } // error
```

the relationships are as follows:

- The class `Point` is a direct subclass of `Object`.
- The class `Object` is the direct superclass of the class `Point`.
- The class `ColoredPoint` is a direct subclass of class `Point`.
- The class `Point` is the direct superclass of class `ColoredPoint`.

The declaration of class `Colored3dPoint` causes a compile-time error because it attempts to extend the `final` class `ColoredPoint`.

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*.

In the example:

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
final class Colored3dPoint extends ColoredPoint { int z; }
```

the relationships are as follows:

- The class `Point` is a superclass of class `ColoredPoint`.
- The class `Point` is a superclass of class `Colored3dPoint`.
- The class `ColoredPoint` is a subclass of class `Point`.
- The class `ColoredPoint` is a superclass of class `Colored3dPoint`.
- The class `Colored3dPoint` is a subclass of class `ColoredPoint`.
- The class `Colored3dPoint` is a subclass of class `Point`.

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 within 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.

For example:

```
class Point extends ColoredPoint { int x, y; }
class ColoredPoint extends Point { int color; }
```

causes a compile-time error.

If circularly declared classes are detected at run time, as classes are loaded (§12.2), then a `ClassCircularityError` is thrown.

8.1.4 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*

The following is repeated from §4.3 to make the presentation here clearer:

InterfaceType:
TypeName

Each *InterfaceType* must name an accessible (§6.6) interface type, or a compile-time error occurs.

A compile-time error occurs if the same interface is mentioned two or more times in a single `implements` clause.

This is true even if the interface is named in different ways; for example, the code:

```
class Redundant implements java.lang.Cloneable, Cloneable {
   int x;
}
```

results in a compile-time error because the names `java.lang.Cloneable` and `Cloneable` refer to the same interface.

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 §9.1.2.
- *I* is a superinterface of the direct superclass of *C*.

A class is said to *implement* all its superinterfaces.

In the example:

```
public interface Colorable {
   void setColor(int color);
   int getColor();
}
public interface Paintable extends Colorable {
   int MATTE = 0, GLOSSY = 1;
   void setFinish(int finish);
   int getFinish();
}
class Point { int x, y; }
class ColoredPoint extends Point implements Colorable {
   int color;
   public void setColor(int color) { this.color = color; }
   public int getColor() { return color; }
}
```

```
class PaintedPoint extends ColoredPoint implements Paintable
{
   int finish;
   public void setFinish(int finish) {
      this.finish = finish;
   }
   public int getFinish() { return finish; }
}
```

the relationships are as follows:

- The interface `Paintable` is a superinterface of class `PaintedPoint`.
- The interface `Colorable` is a superinterface of class `ColoredPoint` and of class `PaintedPoint`.
- The interface `Paintable` is a subinterface of the interface `Colorable`, and `Colorable` is a superinterface of `Paintable`, as defined in §9.1.2.

A class can have a superinterface in more than one way. In this example, the class `PaintedPoint` has `Colorable` as a superinterface both because it is a superinterface of `ColoredPoint` and because it is a superinterface of `Paintable`.

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 (§8.1.4.1).

Thus, the example:

```
interface Colorable {
   void setColor(int color);
   int getColor();
}

class Point { int x, y; };

class ColoredPoint extends Point implements Colorable {
   int color;
}
```

causes a compile-time error, because `ColoredPoint` is not an `abstract` class but it fails to provide an implementation of methods `setColor` and `getColor` of the interface `Colorable`.

It is permitted for a single method declaration in a class to implement methods of more than one superinterface. For example, in the code:

```
interface Fish { int getNumberOfScales(); }

interface Piano { int getNumberOfScales(); }
```

```
class Tuna implements Fish, Piano {
    // You can tune a piano, but can you tuna fish?
    int getNumberOfScales() { return 91; }
}
```

the method `getNumberOfScales` in class `Tuna` has a name, signature, and return type that matches the method declared in interface `Fish` and also matches the method declared in interface `Piano`; it is considered to implement both.

On the other hand, in a situation such as this:

```
interface Fish { int getNumberOfScales(); }

interface StringBass { double getNumberOfScales(); }

class Bass implements Fish, StringBass {
    // This declaration cannot be correct, no matter what type is used.
    public ??? getNumberOfScales() { return 91; }
}
```

it is impossible to declare a method named `getNumberOfScales` with the same signature and return type as those of both the methods declared in interface `Fish` and in interface `StringBass`, because a class can have only one method with a given signature (§8.4). Therefore, it is impossible for a single class to implement both interface `Fish` and interface `StringBass` (§8.4.6).

8.1.5 Class Body and Member Declarations

A *class body* may contain declarations of members of the class, that is, fields (§8.3), classes (§8.5), interfaces (§8.5) and methods (§8.4). A class body may also contain instance initializers (§8.6), static initializers (§8.7), and declarations of constructors (§8.9) for the class.

ClassBody:
`{` *ClassBodyDeclarations*^{opt} `}`

ClassBodyDeclarations:
ClassBodyDeclaration
ClassBodyDeclarations ClassBodyDeclaration

ClassBodyDeclaration:
ClassMemberDeclaration
InstanceInitializer
StaticInitializer
ConstructorDeclaration

ClassMemberDeclaration:
FieldDeclaration
MethodDeclaration
ClassDeclaration
InterfaceDeclaration
;

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

If *C* itself is a nested class, there may be definitions of the same kind (variable, method, or type) for *m* in enclosing scopes. (The scopes may be blocks, classes, or packages.) In all such cases, the member *m* declared or inherited in C hides the other definitions of *m*.

8.2 Class Members

I wouldn't want to belong to any club that would accept me as a member.
—Groucho Marx

The members of a class type are all of the following:

- Members inherited from its direct superclass (§8.1.6), except in class `Object`, which has no direct superclass
- Members inherited from any direct superinterfaces (§8.1.7)
- Members declared in the body of the class (§8.1.8)

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.

The example:

```
class Point {
   int x, y;
   private Point() { reset(); }
   Point(int x, int y) { this.x = x; this.y = y; }
   private void reset() { this.x = 0; this.y = 0; }
}
```

```
class ColoredPoint extends Point {
   int color;
   void clear() { reset(); }                    // error
}
class Test {
   public static void main(String[] args) {
      ColoredPoint c = new ColoredPoint(0, 0);  // error
      c.reset();                                // error
   }
}
```

causes four compile-time errors:

- An error occurs because `ColoredPoint` has no constructor declared with two integer parameters, as requested by the use in `main`. This illustrates the fact that `ColoredPoint` does not inherit the constructors of its superclass `Point`.
- Another error occurs because `ColoredPoint` declares no constructors, and therefore a default constructor for it is automatically created (§8.9.9), and this default constructor is equivalent to:

  ```
  ColoredPoint() { super(); }
  ```

  which invokes the constructor, with no arguments, for the direct superclass of the class `ColoredPoint`. The error is that the constructor for `Point` that takes no arguments is `private`, and therefore is not accessible outside the class `Point`, even through a superclass constructor invocation (§8.9.5).
- Two more errors occur because the method `reset` of class `Point` is `private`, and therefore is not inherited by class `ColoredPoint`. The method invocations in method `clear` of class `ColoredPoint` and in method `main` of class `Test` are therefore not correct.

8.2.1 Examples of Inheritance

This section illustrates inheritance of class members through several examples.

8.2.1.1 *Example: Inheritance with Default Access*

Consider the example where the `points` package declares two compilation units:

```
package points;
public class Point {
   int x, y;
   public void move(int dx, int dy) { x += dx; y += dy; }
}
```

and:

```
package points;

public class Point3d extends Point {
   int z;
   public void move(int dx, int dy, int dz) {
      x += dx; y += dy; z += dz;
   }
}
```

and a third compilation unit, in another package, is:

```
import points.Point3d;

class Point4d extends Point3d {
   int w;
   public void move(int dx, int dy, int dz, int dw) {
      x += dx; y += dy; z += dz; w += dw; // compile-time errors
   }
}
```

Here both classes in the `points` package compile. The class `Point3d` inherits the fields `x` and `y` of class `Point`, because it is in the same package as `Point`. The class `Point4d`, which is in a different package, does not inherit the fields `x` and `y` of class `Point` or the field `z` of class `Point3d`, and so fails to compile.

A better way to write the third compilation unit would be:

```
import points.Point3d;

class Point4d extends Point3d {
   int w;
   public void move(int dx, int dy, int dz, int dw) {
      super.move(dx, dy, dz); w += dw;
   }
}
```

using the `move` method of the superclass `Point3d` to process `dx`, `dy`, and `dz`. If `Point4d` is written in this way it will compile without errors.

8.2.1.2 *Inheritance with* `public` *and* `protected`

Given the class `Point`:

```
package points;

public class Point {

   public int x, y;

   protected int useCount = 0;

   static protected int totalUseCount = 0;
```

```
    public void move(int dx, int dy) {
        x += dx; y += dy; useCount++; totalUseCount++;
    }
}
```

the `public` and `protected` fields `x`, `y`, `useCount` and `totalUseCount` are inherited in all subclasses of `Point`. Therefore, this test program, in another package, can be compiled successfully:

```
class Test extends points.Point {
    public void moveBack(int dx, int dy) {
        x -= dx; y -= dy; useCount++; totalUseCount++;
    }
}
```

8.2.1.3 *Inheritance with* `private`

In the example:

```
class Point {
    int x, y;
    void move(int dx, int dy) {
        x += dx; y += dy; totalMoves++;
    }
    private static int totalMoves;
    void printMoves() { System.out.println(totalMoves); }
}
class Point3d extends Point {
    int z;
    void move(int dx, int dy, int dz) {
        super.move(dx, dy); z += dz; totalMoves++;
    }
}
```

the class variable `totalMoves` can be used only within the class `Point`; it is not inherited by the subclass `Point3d`. A compile-time error occurs because method `move` of class `Point3d` tries to increment `totalMoves`.

8.2.1.4 *Accessing Members of Inaccessible Classes*

Even though a class might not be declared `public`, instances of the class might be available at run time to code outside the package in which it is declared if it has a `public` superclass or superinterface. An instance of the class can be assigned to a

variable of such a `public` type. An invocation of a `public` method of the object referred to by such a variable may invoke a method of the class if it implements or overrides a method of the `public` superclass or superinterface. (In this situation, the method is necessarily declared `public`, even though it is declared in a class that is not `public`.)

Consider the compilation unit:

```
package points;

public class Point {
   public int x, y;
   public void move(int dx, int dy) {
      x += dx; y += dy;
   }
}
```

and another compilation unit of another package:

```
package morePoints;

class Point3d extends points.Point {
   public int z;
   public void move(int dx, int dy, int dz) {
      super.move(dx, dy); z += dz;
   }
   public void move(int dx, int dy) {
      move(dx, dy, 0);
   }
}

public class OnePoint {
   public static points.Point getOne() {
      return new Point3d();
   }
}
```

An invocation `morePoints.OnePoint.getOne()` in yet a third package would return a `Point3d` that can be used as a `Point`, even though the type `Point3d` is not available outside the package `morePoints`. The two argument version of method `move` could then be invoked for that object, which is permissible because method `move` of `Point3d` is `public` (as it must be, for any method that overrides a `public` method must itself be `public`, precisely so that situations such as this will work out correctly). The fields `x` and `y` of that object could also be accessed from such a third package.

While the field `z` of class `Point3d` is `public`, it is not possible to access this field from code outside the package `morePoints`, given only a reference to an instance of class `Point3d` in a variable `p` of type `Point`. This is because the expression `p.z` is not correct, as `p` has type `Point` and class `Point` has no field

named z; also, the expression ((Point3d)p).z is not correct, because the class type Point3d cannot be referred to outside package morePoints. The declaration of the field z as public is not useless, however. If there were to be, in package morePoints, a public subclass Point4d of the class Point3d:

```
package morePoints;

public class Point4d extends Point3d {
   public int w;
   public void move(int dx, int dy, int dz, int dw) {
      super.move(dx, dy, dz); w += dw;
   }
}
```

then class Point4d would inherit the field z, which, being public, could then be accessed by code in packages other than morePoints, through variables and expressions of the public type Point4d.

8.3 Field Declarations

Poetic fields encompass me around,
And still I seem to tread on classic ground.
—Joseph Addison (1672–1719), *A Letter from Italy*

The variables of a class type are introduced by *field declarations*:

FieldDeclaration:
FieldModifiersopt Type VariableDeclarators ;

VariableDeclarators:
VariableDeclarator
VariableDeclarators , *VariableDeclarator*

VariableDeclarator:
VariableDeclaratorId
VariableDeclaratorId = *VariableInitializer*

VariableDeclaratorId:
Identifier
VariableDeclaratorId [ ]

VariableInitializer:
Expression
ArrayInitializer

The *FieldModifiers* are described in §8.3.1. The *Identifier* in a *FieldDeclarator* may be used in a name to refer to the field. Fields are members; the scope (§6.3) of a field name is specified in §8.1.8. 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. Variable declarations involving array types are discussed in §10.2.

It is a compile-time error for the body of a class declaration to contain declarations of two fields with the same name. Methods, types, and fields may have the same name, since they are used in different contexts and are disambiguated by different lookup procedures (§6.5).

If the class declares a field with a certain name, then the declaration of that field is said to *hide* (§6.3.1) any and all accessible declarations of fields with the same name in any enclosing classes, enclosing interfaces, superclasses, and superinterfaces of the class; the field declaration also hides declarations of any local variables, formal method parameters, and exception handler parameters with the same name in any enclosing blocks.

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 (§8.3.3.3). 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 hidden field can be accessed by using a qualified name (if it is `static`) or by using a field access expression (§15.11) that contains the keyword `super` or a cast to a superclass type. See §15.11.2 for discussion and an example.

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
`public protected private`
`static final transient volatile`

The access modifiers `public`, `protected`, and `private` are discussed in §6.6. A compile-time error occurs 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`.

If two or more (distinct) field modifiers appear in a field declaration, it is customary, though not required, that they appear in the order consistent with that shown above in the production for *FieldModifier*.

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, a new variable associated with that instance is created for every instance variable declared in that class or any of its superclasses.

The example program:

```
class Point {
   int x, y, useCount;
   Point(int x, int y) { this.x = x; this.y = y; }
   final static Point origin = new Point(0, 0);
}

class Test {
   public static void main(String[] args) {
      Point p = new Point(1,1);
      Point q = new Point(2,2);
      p.x = 3; p.y = 3; p.useCount++; p.origin.useCount++;
      System.out.println("(" + q.x + "," + q.y + ")");
      System.out.println(q.useCount);
      System.out.println(q.origin == Point.origin);
      System.out.println(q.origin.useCount);
```

```
    }
}
```

prints:

```
(2,2)
0
true
1
```

showing that changing the fields `x`, `y`, and `useCount` of `p` does not affect the fields of `q`, because these fields are instance variables in distinct objects. In this example, the class variable `origin` of the class `Point` is referenced both using the class name as a qualifier, in `Point.origin`, and using variables of the class type in field access expressions (§15.11), as in `p.origin` and `q.origin`. These two ways of accessing the `origin` class variable access the same object, evidenced by the fact that the value of the reference equality expression (§15.21.3):

```
q.origin==Point.origin
```

is `true`. Further evidence is that the incrementation:

```
p.origin.useCount++;
```

causes the value of `q.origin.useCount` to be 1; this is so because `p.origin` and `q.origin` refer to the same variable.

8.3.1.2 `final` *Fields*

A field can be declared `final` (§4.5.4). Both class and instance variables (`static` and non-`static` fields) may be declared `final`.

It is a compile-time error if a blank `final` (§4.5.4) class variable is not definitely assigned (§16.5) by a static initializer (§8.7) of the class in which it is declared.

A blank `final` instance variable must be definitely assigned (§16.6) at the end of every constructor (§8.9) 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.

If an instance of the class `Point`:

```
class Point {
   int x, y;
   transient float rho, theta;
}
```

were saved to persistent storage by a system service, then only the fields `x` and `y` would be saved. This specification does not specify details of such services; see the specification of `java.io.Serializable` for an example of such a service.

8.3.1.4 `volatile` *Fields*

As described in §17, the Java programming language allows threads that access shared variables to keep private working copies of the variables; this allows a more efficient implementation of multiple threads. These working copies need be reconciled with the master copies in the shared main memory only at prescribed synchronization points, namely when objects are locked or unlocked. As a rule, to ensure that shared variables are consistently and reliably updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock that, conventionally, enforces mutual exclusion for those shared variables.

The Java programming language provides a second mechanism that is more convenient for some purposes:

A field may be declared `volatile`, in which case a thread must reconcile its working copy of the field with the master copy every time it accesses the variable. Moreover, operations on the master copies of one or more volatile variables on behalf of a thread are performed by the main memory in exactly the order that the thread requested.

If, in the following example, one thread repeatedly calls the method `one` (but no more than `Integer.MAX_VALUE` times in all), and another thread repeatedly calls the method `two`:

```
class Test {
   static int i = 0, j = 0;
   static void one() { i++; j++; }
   static void two() {
      System.out.println("i=" + i + " j=" + j);
   }
}
```

then method `two` could occasionally print a value for `j` that is greater than the value of `i`, because the example includes no synchronization and, under the rules explained in §17, the shared values of `i` and `j` might be updated out of order.

One way to prevent this out-or-order behavior would be to declare methods `one` and `two` to be `synchronized` (§8.4.3.6):

```
class Test {
   static int i = 0, j = 0;
```

```
    static synchronized void one() { i++; j++; }
    static synchronized void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

This prevents method `one` and method `two` from being executed concurrently, and furthermore guarantees that the shared values of `i` and `j` are both updated before method `one` returns. Therefore method `two` never observes a value for `j` greater than that for `i`; indeed, it always observes the same value for `i` and `j`.

Another approach would be to declare `i` and `j` to be `volatile`:

```
class Test {
    static volatile int i = 0, j = 0;
    static void one() { i++; j++; }
    static void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

This allows method `one` and method `two` to be executed concurrently, but guarantees that accesses to the shared values for `i` and `j` occur exactly as many times, and in exactly the same order, as they appear to occur during execution of the program text by each thread. Therefore, the shared value for `j` is never greater than that for `i`, because each update to `i` must be reflected in the shared value for `i` before the update to `j` occurs. It is possible, however, that any given invocation of method `two` might observe a value for `j` that is much greater than the value observed for `i`, because method `one` might be executed many times between the moment when method `two` fetches the value of `i` and the moment when method `two` fetches the value of `j`.

See §17 for more discussion and examples.

A compile-time error occurs 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).

The example:

```
class Point {
   int x = 1, y = 5;
}
class Test {
   public static void main(String[] args) {
      Point p = new Point();
      System.out.println(p.x + ", " + p.y);
   }
}
```

produces the output:

```
1, 5
```

because the assignments to `x` and `y` occur whenever a new `Point` is created.

Variable initializers are also used in local variable declaration statements (§14.4), where the initializer is evaluated and the assignment performed each time the local variable declaration statement is executed.

It is a compile-time error if the evaluation of a variable initializer for a field of a class (or interface) can complete abruptly with a checked exception (§11.2).

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.

(One subtlety here is that, at run time, `static` variables that are `final` and that are initialized with compile-time constant values are initialized first. This also applies to such fields in interfaces (§9.3.1). These variables are "constants" that will never be observed to have their default initial values (§4.5.5), even by devious programs. See §12.4.2 and §13.4.8 for more discussion.)

Use of class variables whose declarations appear textually after the use is sometimes restricted, even though these class variables are in scope. See §6.3 for the precise scope rules governing class variables.

8.3.2.2 *Initializers for Instance Variables*

A compile-time error occurs if an initialization expression for an instance variable contains a use by a simple name of that instance variable or of another instance variable whose declaration occurs to its right (that is, textually later) in the same class.

Thus:

```
class Test {
   float f = j;
   int j = 1;
   int k = k+1;
}
```

causes two compile-time errors, because `j` is referred to in the initialization of `f` before `j` is declared and because the initialization of `k` refers to `k` itself.

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.

Thus the example:

```
class Test {
   float f = j;
   static int j = 1;
}
```

compiles without error; it initializes `j` to `1` when class `Test` is initialized, and initializes `f` to the current value of `j` every time an instance of class `Test` is created.

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).

Use of instance variables whose declarations appear textually after the use is sometimes restricted, even though these instance variables are in scope. See §6.3 for the precise scope rules governing instance variables.

8.3.3 Examples of Field Declarations

The following examples illustrate some (possibly subtle) points about field declarations.

8.3.3.1 *Example: Hiding of Class Variables*

The example:

```
class Point {
   static int x = 2;
}

class Test extends Point {
   static double x = 4.7;
   public static void main(String[] args) {
      new Test().printX();
   }
   void printX() {
      System.out.println(x + " " + super.x);
   }
}
```

produces the output:

```
4.7 2
```

because the declaration of `x` in class `Test` hides the definition of `x` in class `Point`, so class `Test` does not inherit the field `x` from its superclass `Point`. Within the declaration of class `Test`, the simple name `x` refers to the field declared within class `Test`. Code in class `Test` may refer to the field `x` of class `Point` as `super.x` (or, because `x` is `static`, as `Point.x`). If the declaration of `Test.x` is deleted:

```
class Point {
   static int x = 2;
}

class Test extends Point {
   public static void main(String[] args) {
      new Test().printX();
   }
   void printX() {
      System.out.println(x + " " + super.x);
   }
}
```

then the field `x` of class `Point` is no longer hidden within class `Test`; instead, the simple name `x` now refers to the field `Point.x`. Code in class `Test` may still refer to that same field as `super.x`. Therefore, the output from this variant program is:

```
2 2
```

8.3.3.2 *Example: Hiding of Instance Variables*

This example is similar to that in the previous section, but uses instance variables rather than static variables. The code:

```
class Point {
   int x = 2;
}

class Test extends Point {
   double x = 4.7;
   void printBoth() {
      System.out.println(x + " " + super.x);
   }
   public static void main(String[] args) {
      Test sample = new Test();
      sample.printBoth();
      System.out.println(sample.x + " " +
                                        ((Point)sample).x);
   }
}
```

produces the output:

```
4.7 2
4.7 2
```

because the declaration of `x` in class `Test` hides the definition of `x` in class `Point`, so class `Test` does not inherit the field `x` from its superclass `Point`. It must be noted, however, that while the field `x` of class `Point` is not *inherited* by class `Test`, it is nevertheless *implemented* by instances of class `Test`. In other words, every instance of class `Test` contains two fields, one of type `int` and one of type `float`. Both fields bear the name `x`, but within the declaration of class `Test`, the simple name `x` always refers to the field declared within class `Test`. Code in instance methods of class `Test` may refer to the instance variable `x` of class `Point` as `super.x`.

Code that uses a field access expression to access field `x` will access the field named `x` in the class indicated by the type of reference expression. Thus, the expression `sample.x` accesses a `float` value, the instance variable declared in class `Test`, because the type of the variable sample is `Test`, but the expression `((Point)sample).x` accesses an `int` value, the instance variable declared in class `Point`, because of the cast to type `Point`.

If the declaration of `x` is deleted from class `Test`, as in the program:

```
class Point {
   static int x = 2;
}

class Test extends Point {
   void printBoth() {
      System.out.println(x + " " + super.x);
   }
   public static void main(String[] args) {
```

```
        Test sample = new Test();
        sample.printBoth();
        System.out.println(sample.x + " " +
                                        ((Point)sample).x);
    }
}
```

then the field `x` of class `Point` is no longer hidden within class `Test`. Within instance methods in the declaration of class `Test`, the simple name `x` now refers to the field declared within class `Point`. Code in class `Test` may still refer to that same field as `super.x`. The expression `sample.x` still refers to the field `x` within type `Test`, but that field is now an inherited field, and so refers to the field `x` declared in class `Point`. The output from this variant program is:

```
2 2
2 2
```

8.3.3.3 *Example: Multiply Inherited Fields*

A class may inherit two or more fields with the same name, either from two interfaces or from its superclass and an interface. A compile-time error occurs on any attempt to refer to any ambiguously inherited field by its simple name. A qualified name or a field access expression that contains the keyword `super` (§15.11.2) may be used to access such fields unambiguously. In the example:

```
interface Frob { float v = 2.0f; }

class SuperTest { int v = 3; }

class Test extends SuperTest implements Frob {
    public static void main(String[] args) {
        new Test().printV();
    }
    void printV() { System.out.println(v); }
}
```

the class `Test` inherits two fields named `v`, one from its superclass `SuperTest` and one from its superinterface `Frob`. This in itself is permitted, but a compile-time error occurs because of the use of the simple name `v` in method `printV`: it cannot be determined which `v` is intended.

The following variation uses the field access expression `super.v` to refer to the field named `v` declared in class `SuperTest` and uses the qualified name `Frob.v` to refer to the field named `v` declared in interface `Frob`:

```
interface Frob { float v = 2.0f; }

class SuperTest { int v = 3; }
```

```
class Test extends SuperTest implements Frob {
   public static void main(String[] args) {
      new Test().printV();
   }
   void printV() {
      System.out.println((super.v + Frob.v)/2);
   }
}
```

It compiles and prints:

```
2.5
```

Even if two distinct inherited fields have the same type, the same value, and are both `final`, any reference to either field by simple name is considered ambiguous and results in a compile-time error. In the example:

```
interface Color { int RED=0, GREEN=1, BLUE=2; }

interface TrafficLight { int RED=0, YELLOW=1, GREEN=2; }

class Test implements Color, TrafficLight {
   public static void main(String[] args) {
      System.out.println(GREEN);      // compile-time error
      System.out.println(RED);        // compile-time error
   }
}
```

it is not astonishing that the reference to `GREEN` should be considered ambiguous, because class `Test` inherits two different declarations for `GREEN` with different values. The point of this example is that the reference to `RED` is also considered ambiguous, because two distinct declarations are inherited. The fact that the two fields named `RED` happen to have the same type and the same unchanging value does not affect this judgment.

8.3.3.4 *Example: Re-inheritance of Fields*

If the same field declaration is inherited from an interface by multiple paths, the field is considered to be inherited only once. It may be referred to by its simple name without ambiguity. For example, in the code:

```
public interface Colorable {
   int RED = 0xff0000, GREEN = 0x00ff00, BLUE = 0x0000ff;
}

public interface Paintable extends Colorable {
   int MATTE = 0, GLOSSY = 1;
}

class Point { int x, y; }
```

```
class ColoredPoint extends Point implements Colorable {
   ...
}
class PaintedPoint extends ColoredPoint implements Paintable
{
   ... RED ...
}
```

the fields `RED`, `GREEN`, and `BLUE` are inherited by the class `PaintedPoint` both through its direct superclass `ColoredPoint` and through its direct superinterface `Paintable`. The simple names `RED`, `GREEN`, and `BLUE` may nevertheless be used without ambiguity within the class `PaintedPoint` to refer to the fields declared in interface `Colorable`.

8.4 Method Declarations

The diversity of physical arguments and opinions embraces all sorts of methods.
—Michael de Montaigne (1533–1592), *Of Experience*

A *method* declares executable code that can be invoked, passing a fixed number of values as arguments.

MethodDeclaration:
MethodHeader MethodBody

MethodHeader:
MethodModifiers^{opt} *ResultType MethodDeclarator Throws*^{opt}

ResultType:
Type
`void`

MethodDeclarator:
Identifer (*FormalParameterList*^{opt})

The *MethodModifiers* are described in §8.4.3, the *Throws* clause in §8.4.4, and the *MethodBody* in §8.4.5. A method declaration either specifies the type of value that the method returns or uses the keyword `void` to indicate that the method does not return a value.

The *Identifier* in a *MethodDeclarator* may be used in a name to refer to the method. A class can declare a method with the same name as the class or a field, member class or member interface of the class.

For compatibility with older versions of the Java platform, a declaration form for 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 have as members two methods with the same signature (§8.4.2) (name, number of parameters, and types of any parameters). Methods and fields may have the same name, since they are used in different contexts and are disambiguated by the different lookup procedures (§6.5).

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 an identifier (optionally followed by brackets) that specifies the name of the parameter:

FormalParameterList:
FormalParameter
FormalParameterList `,` *FormalParameter*

FormalParameter:
final^{opt} *Type* *VariableDeclaratorId*

The following is repeated from §8.3 to make the presentation here clearer:

VariableDeclaratorId:
Identifier
VariableDeclaratorId `[ ]`

If a method or constructor has no parameters, only an empty pair of parentheses appears in the declaration of the method or constructor.

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*), then a compile-time error occurs.

It is a compile time error if a method or constructor 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.

The scope of a parameter of a method (§8.4.1) or constructor (§8.9.1) is the entire body of the method or constructor.

These parameter names may not be redeclared as local variables of the method, or as exception parameters of catch clauses in a try statement of the method or constructor. However, a parameter name of a method or constructor may be hidden anywhere inside a class declaration nested within that method or constructor. Such a nested class declaration could declare either a local class (§14.3) or an anonymous class (§15.9).

Formal parameters are referred to only using simple names, never by using qualified names (§6.6).

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

The *signature* of a method consists of the name of the method and the number and types of formal parameters to the method. A class may not declare two methods with the same signature, or a compile-time error occurs.

The example:

```
class Point implements Move {
   int x, y;
   abstract void move(int dx, int dy);
   void move(int dx, int dy) { x += dx; y += dy; }
}
```

causes a compile-time error because it declares two `move` methods with the same signature. This is an error even though one of the declarations is `abstract`.

8.4.3 Method Modifiers

MethodModifiers:
MethodModifier
MethodModifiers MethodModifier

MethodModifier: one of
`public protected private abstract static`
`final synchronized native strictfp`

The access modifiers `public`, `protected`, and `private` are discussed in §6.6. A compile-time error occurs 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`. A compile-time error occurs if a method declaration that contains the keyword `abstract` also contains any one of the keywords `private`, `static`, `final`, `native`, `strictfp`, or `synchronized`. A compile-time error occurs if a method declaration that contains the keyword `native` also contains `strictfp`.

If two or more method modifiers appear in a method declaration, it is customary, though not required, that they appear in the order consistent with that shown above in the production for *MethodModifier*.

8.4.3.1 `abstract` *Methods*

An `abstract` method declaration introduces the method as a member, providing its signature (name and number and type of parameters), 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*); otherwise a compile-time error results. Every subclass of *A* that is not `abstract` must provide an implementation for *m*, or a compile-time error occurs as specified in §8.1.4.1.

It is a compile-time error for a `private` method to be declared `abstract`.

It would be impossible for a subclass to implement a `private abstract` method, because `private` methods are not inherited by subclasses; therefore such a method could never be used.

It is a compile-time error for a `static` method to be declared `abstract`.

It is a compile-time error for a `final` method to be declared `abstract`.

An `abstract` class can override an `abstract` method by providing another `abstract` method declaration.

This can provide a place to put a documentation comment, or to declare that the set of checked exceptions (§11.2) that can be thrown by that method, when it is implemented by its subclasses, is to be more limited. For example, consider this code:

```
class BufferEmpty extends Exception {
   BufferEmpty() { super(); }
   BufferEmpty(String s) { super(s); }
}

class BufferError extends Exception {
   BufferError() { super(); }
   BufferError(String s) { super(s); }
}

public interface Buffer {
   char get() throws BufferEmpty, BufferError;
}

public abstract class InfiniteBuffer implements Buffer {
   abstract char get() throws BufferError;
}
```

The overriding declaration of method `get` in class `InfiniteBuffer` states that method `get` in any subclass of `InfiniteBuffer` never throws a `BufferEmpty` exception, putatively because it generates the data in the buffer, and thus can never run out of data.

An instance method that is not `abstract` can be overridden by an `abstract` method.

For example, we can declare an `abstract` class `Point` that requires its subclasses to implement `toString` if they are to be complete, instantiable classes:

```
abstract class Point {
   int x, y;
   public abstract String toString();
}
```

This `abstract` declaration of `toString` overrides the non-`abstract` `toString` method of class `Object`. (Class `Object` is the implicit direct superclass of class `Point`.) Adding the code:

```
class ColoredPoint extends Point {
   int color;
   public String toString() {
      return super.toString() + ": color " + color; // error
   }
}
```

results in a compile-time error because the invocation `super.toString()` refers to method `toString` in class `Point`, which is `abstract` and therefore cannot be

invoked. Method `toString` of class `Object` can be made available to class `ColoredPoint` only if class `Point` explicitly makes it available through some other method, as in:

```
abstract class Point {
   int x, y;
   public abstract String toString();
   protected String objString() { return super.toString(); }
}

class ColoredPoint extends Point {
   int color;
   public String toString() {
      return objString() + ": color " + color;     // correct
   }
}
```

8.4.3.2 `static` *Methods*

A method that is declared `static` is called a *class method*. A class method is always invoked without reference to a particular object. An attempt to reference the current object using the keyword `this` or the keyword `super` in the body of a class method results in a compile time error. It is a compile-time error for a `static` method to be declared `abstract`.

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 in a `final` class (§8.1.4.2) are implicitly `final`, because it is impossible to override them. It is permitted but not required for the declarations of such methods to redundantly include the `final` keyword.

It is a compile-time error for a `final` method to be declared `abstract`.

At run-time, a machine-code generator or optimizer can easily and safely "inline" the body of a `final` method, replacing an invocation of the method with the code in its body.

Consider the example:

```
final class Point {
   int x, y;
   void move(int dx, int dy) { x += dx; y += dy; }
}
class Test {
   public static void main(String[] args) {
      Point[] p = new Point[100];
      for (int i = 0; i < p.length; i++) {
         p[i] = new Point();
         p[i].move(i, p.length-1-i);
      }
   }
}
```

Here, inlining the method `move` of class `Point` in method `main` would transform the `for` loop to the form:

```
      for (int i = 0; i < p.length; i++) {
         p[i] = new Point();
         Point pi = p[i];
         pi.x += i;
         pi.y += p.length-1-i;
      }
```

The loop might then be subject to further optimizations.

Such inlining cannot be done at compile time unless it can be guaranteed that `Test` and `Point` will always be recompiled together, so that whenever `Point`—and specifically its `move` method—changes, the code for `Test.main` will also be updated.

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.

A compile-time error occurs if a `native` method is declared `abstract`.

For example, the class `RandomAccessFile` of the package `java.io` might declare the following `native` methods:

```
package java.io;

public class RandomAccessFile
   implements DataOutput, DataInput
{  ...
```

```
    public native void open(String name, boolean writeable)
       throws IOException;
    public native int readBytes(byte[] b, int off, int len)
       throws IOException;
    public native void writeBytes(byte[] b, int off, int len)
       throws IOException;
    public native long getFilePointer() throws IOException;
    public native void seek(long pos) throws IOException;
    public native long length() throws IOException;
    public native void close() throws IOException;
}
```

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 lock (§17.1) before it executes. For a class (`static`) method, the lock associated with the `Class` object for the method's class is used. For an instance method, the lock associated with `this` (the object for which the method was invoked) is used.

These are the same locks that can be used by the `synchronized` statement (§14.18); thus, the code:

```
class Test {
   int count;
   synchronized void bump() { count++; }
   static int classCount;
   static synchronized void classBump() {
      classCount++;
   }
}
```

has exactly the same effect as:

```
class BumpTest {
   int count;
   void bump() {
      synchronized (this) {
         count++;
      }
   }
   static int classCount;
   static void classBump() {
      try {
```

```
            synchronized (Class.forName("BumpTest")) {
               classCount++;
            }
         } catch (ClassNotFoundException e) {
               ...
         }
      }
   }
```

The more elaborate example:

```
public class Box {

   private Object boxContents;

   public synchronized Object get() {
      Object contents = boxContents;
      boxContents = null;
      return contents;
   }

   public synchronized boolean put(Object contents) {
      if (boxContents != null)
         return false;
      boxContents = contents;
      return true;
   }

}
```

defines a class which is designed for concurrent use. Each instance of the class `Box` has an instance variable `contents` that can hold a reference to any object. You can put an object in a `Box` by invoking `put`, which returns `false` if the box is already full. You can get something out of a `Box` by invoking `get`, which returns a null reference if the `box` is empty.

If `put` and `get` were not `synchronized`, and two threads were executing methods for the same instance of `Box` at the same time, then the code could misbehave. It might, for example, lose track of an object because two invocations to `put` occurred at the same time.

See §17 for more discussion of threads and locks.

8.4.4 Method Throws

A *throws clause* is used to declare any checked exceptions (§11.2) that can result from the execution of a method or constructor:

Throws:
`throws` *ClassTypeList*

ClassTypeList:
ClassType
ClassTypeList `,` *ClassType*

A compile-time error occurs if any *ClassType* mentioned in a `throws` clause is not the class `Throwable` or a subclass of `Throwable`. It is permitted but not required to mention other (unchecked) exceptions in a `throws` clause.

For each checked exception that can result from execution of the body of a method or constructor, a compile-time error occurs unless that exception type or a superclass of that exception type is mentioned in a `throws` clause in the declaration of the method or constructor.

The requirement to declare checked exceptions allows the compiler to ensure that code for handling such error conditions has been included. Methods or constructors that fail to handle exceptional conditions thrown as checked exceptions will normally result in a compile-time error because of the lack of a proper exception type in a `throws` clause. The Java programming language thus encourages a programming style where rare and otherwise truly exceptional conditions are documented in this way.

The predefined exceptions that are not checked in this way are those for which declaring every possible occurrence would be unimaginably inconvenient:

- Exceptions that are represented by the subclasses of class `Error`, for example `OutOfMemoryError`, are thrown due to a failure in or of the virtual machine. Many of these are the result of linkage failures and can occur at unpredictable points in the execution of a program. Sophisticated programs may yet wish to catch and attempt to recover from some of these conditions.

- The exceptions that are represented by the subclasses of the class `RuntimeException`, for example `NullPointerException`, result from runtime integrity checks and are thrown either directly from the program or in library routines. It is beyond the scope of the Java programming language, and perhaps beyond the state of the art, to include sufficient information in the program to reduce to a manageable number the places where these can be proven not to occur.

A method that overrides or hides another method (§8.4.6), 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 super-

classes must occur in the `throws` clause of *m*; otherwise, a compile-time error occurs.

See §11 for more information about exceptions and a large example.

8.4.5 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
;

A compile-time error occurs if a method declaration is either `abstract` or `native` and has a block for its body. A compile-time error occurs if a method declaration is neither `abstract` nor `native` and has a semicolon for its body.

If an implementation is to be provided for a method but the implementation requires no executable code, the method body should be written as a block that contains no statements: "`{ }`".

If a method is declared `void`, then its body must not contain any `return` statement (§14.16) that has an *Expression*.

If a method is declared to have a return type, then every `return` statement (§14.16) in its body must have an *Expression*. A compile-time error occurs if the body of the method can complete normally (§14.1).

In other words, a method with a return type must return only by using a return statement that provides a value return; it is not allowed to "drop off the end of its body."

Note that it is possible for a method to have a declared return type and yet contain no return statements. Here is one example:

```
class DizzyDean {
                int pitch() { throw new RuntimeException("90 mph?!"); }
}
```

8.4.6 Inheritance, Overriding, and Hiding

A class *inherits* from its direct superclass and direct superinterfaces all the non-private methods (whether `abstract` or not) of the superclass and superinterfaces

that are accessible to code in the class and are neither overridden (§8.4.6.1) nor hidden (§8.4.6.2) by a declaration in the class.

8.4.6.1 *Overriding (by Instance Methods)*

An instance method *m1* declared in a class *C overrides* another method with the same signature, *m2*, declared in class *A* iff both:

1. C is a subclass of *A*.
2. Either
   - *m2* is non-private and accessible from *C*, or
   - *m1* overrides a method *m3*, *m3* distinct from *m1*, *m3* distinct from *m2*, such that *m3* overrides *m2*.

Moreover, if *m1* is not `abstract`, then *m1* is said to *implement* any and all declarations of `abstract` methods that it overrides.

A compile-time error occurs if an instance method overrides a `static` method.

In this respect, overriding of methods differs from hiding of fields (§8.3), for it is permissible for an instance variable to hide a `static` variable.

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 §15.12.4.9 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.

8.4.6.2 *Hiding (by Class Methods)*

If a class declares a `static` method, then the declaration of that method is said to *hide* any and all methods with the same signature in the superclasses and superinterfaces of the class that would otherwise be accessible to code in the class. A compile-time error occurs if a `static` method hides an instance method.

In this respect, hiding of methods differs from hiding of fields (§8.3), for it is permissible for a `static` variable to hide an instance variable.

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.

8.4.6.3 *Requirements in Overriding and Hiding*

If a method declaration overrides or hides the declaration of another method, then a compile-time error occurs if they have different return types or if one has a return type and the other is `void`. Moreover, a method declaration must not have a `throws` clause that conflicts (§8.4.4) with that of any method that it overrides or hides; otherwise, a compile-time error occurs.

In these respects, overriding of methods differs from hiding of fields (§8.3), for it is permissible for a field to hide a field of another type.

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. In more detail:

- 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.

Note that a `private` method cannot be hidden or overridden in the technical sense of those terms. This means that a subclass can declare a method with the same signature as a `private` method in one of its superclasses, and there is no requirement that the return type or `throws` clause of such a method bear any relationship to those of the `private` method in the superclass.

8.4.6.4 *Inheriting Methods with the Same Signature*

It is possible for a class to inherit more than one method with the same signature. Such a situation does not in itself cause a compile-time error. There are then 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. A compile-time error occurs if, comparing the method that is not

`abstract` with each of the other of the inherited methods, for any such pair, either they have different return types or one has a return type and the other is `void`. Moreover, a compile-time error occurs if the inherited method that is not `abstract` has a `throws` clause that conflicts (§8.4.4) 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. A compile-time error occurs if, for any two such inherited methods, either they have different return types or one has a return type and the other is `void`. (The `throws` clauses do not cause errors in this case.)

It is not possible for two or more inherited methods with the same signature not to be `abstract`, because methods that are not `abstract` are inherited only from the direct superclass, not from superinterfaces.

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.

8.4.7 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 different signatures, 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.

Methods are overridden on a signature-by-signature basis.

If, for example, a class declares two `public` methods with the same name, and a subclass overrides one of them, the subclass still inherits the other method. In this respect, the Java programming language differs from C++.

When a method is invoked (§15.12), the number of actual 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).

8.4.8 Examples of Method Declarations

The following examples illustrate some (possibly subtle) points about method declarations.

8.4.8.1 *Example: Overriding*

In the example:

```
class Point {
   int x = 0, y = 0;
   void move(int dx, int dy) { x += dx; y += dy; }
}
class SlowPoint extends Point {
   int xLimit, yLimit;
   void move(int dx, int dy) {
      super.move(limit(dx, xLimit), limit(dy, yLimit));
   }
   static int limit(int d, int limit) {
      return d > limit ? limit : d < -limit ? -limit : d;
   }
}
```

the class `SlowPoint` overrides the declarations of method `move` of class `Point` with its own `move` method, which limits the distance that the point can move on each invocation of the method. When the `move` method is invoked for an instance of class `SlowPoint`, the overriding definition in class `SlowPoint` will always be called, even if the reference to the `SlowPoint` object is taken from a variable whose type is `Point`.

8.4.8.2 *Example: Overloading, Overriding, and Hiding*

In the example:

```
class Point {
   int x = 0, y = 0;
   void move(int dx, int dy) { x += dx; y += dy; }
   int color;
}
class RealPoint extends Point {
   float x = 0.0f, y = 0.0f;
   void move(int dx, int dy) { move((float)dx, (float)dy); }
   void move(float dx, float dy) { x += dx; y += dy; }
}
```

the class `RealPoint` hides the declarations of the `int` instance variables `x` and `y` of class `Point` with its own `float` instance variables `x` and `y`, and overrides the method `move` of class `Point` with its own `move` method. It also overloads the name `move` with another method with a different signature (§8.4.2).

In this example, the members of the class `RealPoint` include the instance variable `color` inherited from the class `Point`, the `float` instance variables `x` and `y` declared in `RealPoint`, and the two `move` methods declared in `RealPoint`.

Which of these overloaded `move` methods of class `RealPoint` will be chosen for any particular method invocation will be determined at compile time by the overloading resolution procedure described in §15.12.

8.4.8.3 *Example: Incorrect Overriding*

This example is an extended variation of that in the preceding section:

```
class Point {
    int x = 0, y = 0, color;
    void move(int dx, int dy) { x += dx; y += dy; }
    int getX() { return x; }
    int getY() { return y; }
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
    float getX() { return x; }
    float getY() { return y; }
}
```

Here the class `Point` provides methods `getX` and `getY` that return the values of its fields `x` and `y`; the class `RealPoint` then overrides these methods by declaring methods with the same signature. The result is two errors at compile time, one for each method, because the return types do not match; the methods in class `Point` return values of type `int`, but the wanna-be overriding methods in class `RealPoint` return values of type `float`.

8.4.8.4 *Example: Overriding versus Hiding*

This example corrects the errors of the example in the preceding section:

```
class Point {
   int x = 0, y = 0;
   void move(int dx, int dy) { x += dx; y += dy; }
   int getX() { return x; }
   int getY() { return y; }
   int color;
}
class RealPoint extends Point {
   float x = 0.0f, y = 0.0f;
   void move(int dx, int dy) { move((float)dx, (float)dy); }
   void move(float dx, float dy) { x += dx; y += dy; }
   int getX() { return (int)Math.floor(x); }
   int getY() { return (int)Math.floor(y); }
}
```

Here the overriding methods `getX` and `getY` in class `RealPoint` have the same return types as the methods of class `Point` that they override, so this code can be successfully compiled.

Consider, then, this test program:

```
class Test {
   public static void main(String[] args) {
      RealPoint rp = new RealPoint();
      Point p = rp;
      rp.move(1.71828f, 4.14159f);
      p.move(1, -1);
      show(p.x, p.y);
      show(rp.x, rp.y);
      show(p.getX(), p.getY());
      show(rp.getX(), rp.getY());
   }
   static void show(int x, int y) {
      System.out.println("(" + x + ", " + y + ")");
   }
   static void show(float x, float y) {
      System.out.println("(" + x + ", " + y + ")");
   }
}
```

The output from this program is:

```
(0, 0)
(2.7182798, 3.14159)
(2, 3)
(2, 3)
```

The first line of output illustrates the fact that an instance of `RealPoint` actually contains the two integer fields declared in class `Point`; it is just that their names are hidden from code that occurs within the declaration of class `RealPoint` (and those of any subclasses it might have). When a reference to an instance of class `RealPoint` in a variable of type `Point` is used to access the field `x`, the integer field `x` declared in class `Point` is accessed. The fact that its value is zero indicates that the method invocation `p.move(1, -1)` did not invoke the method `move` of class `Point`; instead, it invoked the overriding method `move` of class `RealPoint`.

The second line of output shows that the field access `rp.x` refers to the field `x` declared in class `RealPoint`. This field is of type `float`, and this second line of output accordingly displays floating-point values. Incidentally, this also illustrates the fact that the method name `show` is overloaded; the types of the arguments in the method invocation dictate which of the two definitions will be invoked.

The last two lines of output show that the method invocations `p.getX()` and `rp.getX()` each invoke the `getX` method declared in class `RealPoint`. Indeed, there is no way to invoke the `getX` method of class `Point` for an instance of class `RealPoint` from outside the body of `RealPoint`, no matter what the type of the variable we may use to hold the reference to the object. Thus, we see that fields and methods behave differently: hiding is different from overriding.

8.4.8.5 *Example: Invocation of Hidden Class Methods*

A hidden class (`static`) method can be invoked by using a reference whose type is the class that actually contains the declaration of the method. In this respect, hiding of static methods is different from overriding of instance methods. The example:

```
class Super {
   static String greeting() { return "Goodnight"; }
   String name() { return "Richard"; }
}

class Sub extends Super {
   static String greeting() { return "Hello"; }
   String name() { return "Dick"; }
}
```

```
class Test {
   public static void main(String[] args) {
      Super s = new Sub();
      System.out.println(s.greeting() + ", " + s.name());
   }
}
```

produces the output:

```
Goodnight, Dick
```

because the invocation of `greeting` uses the type of `s`, namely `Super`, to figure out, at compile time, which class method to invoke, whereas the invocation of `name` uses the class of `s`, namely `Sub`, to figure out, at run time, which instance method to invoke.

8.4.8.6 *Large Example of Overriding*

Overriding makes it easy for subclasses to extend the behavior of an existing class, as shown in this example:

```
import java.io.OutputStream;
import java.io.IOException;
class BufferOutput {
   private OutputStream o;
   BufferOutput(OutputStream o) { this.o = o; }
   protected byte[] buf = new byte[512];
   protected int pos = 0;
   public void putchar(char c) throws IOException {
      if (pos == buf.length)
         flush();
      buf[pos++] = (byte)c;
   }


   public void putstr(String s) throws IOException {
      for (int i = 0; i < s.length(); i++)
         putchar(s.charAt(i));
   }
   public void flush() throws IOException {
      o.write(buf, 0, pos);
```

```
        pos = 0;
    }
}
class LineBufferOutput extends BufferOutput {
    LineBufferOutput(OutputStream o) { super(o); }
    public void putchar(char c) throws IOException {
        super.putchar(c);
        if (c == '\n')
            flush();
    }
}
class Test {
    public static void main(String[] args)
        throws IOException
    {
        LineBufferOutput lbo =
            new LineBufferOutput(System.out);
        lbo.putstr("lbo\nlbo");
        System.out.print("print\n");
        lbo.putstr("\n");
    }
}
```

This example produces the output:

```
lbo
print
lbo
```

The class `BufferOutput` implements a very simple buffered version of an `OutputStream`, flushing the output when the buffer is full or `flush` is invoked. The subclass `LineBufferOutput` declares only a constructor and a single method `putchar`, which overrides the method `putchar` of `BufferOutput`. It inherits the methods `putstr` and `flush` from class `BufferOutput`.

In the `putchar` method of a `LineBufferOutput` object, if the character argument is a newline, then it invokes the `flush` method. The critical point about overriding in this example is that the method `putstr`, which is declared in class `BufferOutput`, invokes the `putchar` method defined by the current object `this`, which is not necessarily the `putchar` method declared in class `BufferOutput`.

Thus, when `putstr` is invoked in `main` using the `LineBufferOutput` object `lbo`, the invocation of `putchar` in the body of the `putstr` method is an invocation of the `putchar` of the object `lbo`, the overriding declaration of `putchar` that checks for a newline. This allows a subclass of `BufferOutput` to change the behavior of the `putstr` method without redefining it.

Documentation for a class such as `BufferOutput`, which is designed to be extended, should clearly indicate what is the contract between the class and its subclasses, and should clearly indicate that subclasses may override the `putchar` method in this way. The implementor of the `BufferOutput` class would not, therefore, want to change the implementation of `putstr` in a future implementation of `BufferOutput` not to use the method `putchar`, because this would break the preexisting contract with subclasses. See the further discussion of binary compatibility in §13, especially §13.2.

8.4.8.7 *Example: Incorrect Overriding because of Throws*

This example uses the usual and conventional form for declaring a new exception type, in its declaration of the class `BadPointException`:

```
class BadPointException extends Exception {
   BadPointException() { super(); }
   BadPointException(String s) { super(s); }
}

class Point {
   int x, y;
   void move(int dx, int dy) { x += dx; y += dy; }
}

class CheckedPoint extends Point {
   void move(int dx, int dy) throws BadPointException {
      if ((x + dx) < 0 || (y + dy) < 0)
         throw new BadPointException();
      x += dx; y += dy;
   }
}
```

This example results in a compile-time error, because the override of method `move` in class `CheckedPoint` declares that it will throw a checked exception that the `move` in class `Point` has not declared. If this were not considered an error, an invoker of the method `move` on a reference of type `Point` could find the contract between it and `Point` broken if this exception were thrown.

Removing the `throws` clause does not help:

```
class CheckedPoint extends Point {
   void move(int dx, int dy) {
      if ((x + dx) < 0 || (y + dy) < 0)
         throw new BadPointException();
      x += dx; y += dy;
   }
}
```

A different compile-time error now occurs, because the body of the method `move` cannot throw a checked exception, namely `BadPointException`, that does not appear in the `throws` clause for `move`.

8.5 Member Type Declarations

A *member class* is a class whose declaration is directly enclosed in another class or interface declaration. Similarly, a *member interface* is an interface whose declaration is directly enclosed in another class or interface declaration. The scope (§6.3) of the name of a member class or interface is specified in §8.1.8.

Within a class *C*, a declaration *d* of a member type named *n* hides the declarations of any other types named *n* that are in scope at the point where *d* occurs.

If a member class or interface declared with simple name *C* is directly enclosed within the declaration of a class with fully qualified name *N*, then the member class or interface has the fully qualified name *N*`.`*C*.

A class may inherit two or more type declarations with the same name, either from two interfaces or from its superclass and an interface. A compile-time error occurs on any 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 Modifers

The access modifiers `public`, `protected`, and `private` are discussed in §6.6. A compile-time error occurs if a member type declaration has more than one of the access modifiers `public`, `protected`, and `private`.

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 *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), as specified in §8.9.5.1.

InstanceInitializer:
Block

An instance initializer of a named class may not throw a checked exception unless that exception is explicitly declared in the `throws` clause of each constructor of its class and the class has at least one explicitly declared constructor. It follows that an instance initializer in an anonymous class can throw any exceptions. It is a compile time error if an instance initializer cannot complete normally (§14.20). If a `return` statement (§14.16) appears anywhere within an instance initializer, then a compile-time error occurs.

Use of instance variables whose declarations appear textually after the use is sometimes restricted, even though these instance variables are in scope. See §6.3 for the precise scope rules governing instance variables.

Instance initializers are permitted to refer to the current object `this` (§15.8.3) and to use the keyword `super` (§15.11.2, §15.12).

8.7 Static Initializers

Any *static initializers* declared in a class are executed when the class is initialized and, together with any field initializers (§8.3.2) for class variables, may be used to initialize the class variables of the class (§12.4).

StaticInitializer:
`static` *Block*

It is a compile-time error for a static initializer to be able to complete abruptly (§14.1, §15.6) with a checked exception (§11.2). It is a compile time error if a static initializer cannot complete normally (§14.20).

The static initializers and class variable initializers are executed in textual order. Use of class variables whose declarations appear textually after the use is sometimes restricted, even though these class variables are in scope. See §6.3 for the precise scope rules governing class variables.

This restriction is designed to catch, at compile time, circular or otherwise malformed initializations. Thus, both:

```
class Z {
   static int i = j + 2;
   static int j = 4;
}
```

and:

```
class Z {
   static { i = j + 2; }
   static int i, j;
   static { j = 4; }
}
```

result in compile-time errors. Accesses to class variables by methods are not checked in this way, so:

```
class Z {
   static int peek() { return j; }
   static int i = peek();
   static int j = 1;
}
class Test {
   public static void main(String[] args) {
      System.out.println(Z.i);
   }
}
```

produces the output:

```
0
```

because the variable initializer for `i` uses the class method `peek` to access the value of the variable `j` before `j` has been initialized by its variable initializer, at which point it still has its default value (§4.5.5).

If a `return` statement (§14.16) appears anywhere within a static initializer, then a compile-time error occurs.

If the keyword `this` (§15.8.3) or the keyword `super` (§15.11, §15.12) appears anywhere within a static initializer, then a compile-time error occurs.

8.8 Constructor Declarations

The constructor of wharves, bridges, piers, bulk-heads,
floats, stays against the sea . . .

—Walt Whitman, *Song of the Broad-Axe* (1856)

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:
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.

Here is a simple example:

```
class Point {
   int x, y;
   Point(int x, int y) { this.x = x; this.y = y; }
}
```

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 (§8.9.5). Constructors are never invoked by method invocation expressions (§15.12).

Access to constructors is governed by access modifiers (§6.6).

This is useful, for example, in preventing instantiation by declaring an inaccessible constructor (§8.9.11).

Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding.

8.8.1 Formal Parameters

The formal parameters of a constructor are identical in structure and behavior to the formal parameters of a method (§8.4.1).

8.8.2 Constructor Signature

The *signature* of a constructor consists of the number and types of formal parameters to the constructor. A class may not declare two constructors with the same signature, or a compile-time error occurs.

8.8.3 Constructor Modifiers

ConstructorModifiers:
ConstructorModifier
ConstructorModifiers *ConstructorModifier*

ConstructorModifier: one of
`public` `protected` `private`

The access modifiers `public`, `protected`, and `private` are discussed in §6.6. A compile-time error occurs 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`.

Unlike methods, a constructor cannot be `abstract`, `static`, `final`, `native`, or `synchronized`. A constructor is not inherited, so there is no need to declare it `final` and an `abstract` constructor could never be implemented. A constructor is always invoked with respect to an object, so it makes no sense for a constructor to be `static`. There is no practical need for a constructor to be `synchronized`, because it would lock the object under construction, which is normally not made available to other threads until all constructors for the object have completed their work. The lack of `native` constructors is an arbitrary language design choice that makes it easy for an implementation of the Java virtual machine to verify that superclass constructors are always properly invoked during object creation.

Note that a *ConstructorModifier* cannot be declared `strictfp`. This difference in the definitions for *ConstructorModifier* and *MethodModifier* (§8.4.3) is an intentional language design choice; it effectively ensures that a constructor is FP-strict if and only if its class is FP-strict, so to speak.

8.8.4 Constructor Throws

The `throws` clause for a constructor is identical in structure and behavior to the `throws` clause for a method (§8.4.4).

8.8.5 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 (§8.9.5.1).

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 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 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 (§8.4.5). A `return` statement (§14.16) may be used in the body of a constructor if it does not include an expression.

In the example:

```
class Point {
   int x, y;
   Point(int x, int y) { this.x = x; this.y = y; }
}
class ColoredPoint extends Point {
   static final int WHITE = 0, BLACK = 1;
   int color;
   ColoredPoint(int x, int y) {
      this(x, y, WHITE);
   }
   ColoredPoint(int x, int y, int color) {
      super(x, y);
      this.color = color;
   }
}
```

the first constructor of `ColoredPoint` invokes the second, providing an additional argument; the second constructor of `ColoredPoint` invokes the constructor of its superclass `Point`, passing along the coordinates.

§12.5 and §15.9 describe the creation and initialization of new class instances.

8.8.5.1 *Explicit Constructor Invocations*

ExplicitConstructorInvocation:

`this` `(` *ArgumentList*^{opt} `)` `;`

`super` `(` *ArgumentList*^{opt} `)` `;`

Primary`.super` `(` *ArgumentList*^{opt} `)` `;`

Explicit constructor invocation statements can be divided into two kinds:

- *Alternate constructor invocations* begin with the keyword `this`. They are used to invoke an alternative constructor of the same class.
- *Superclass constructor invocation*s begin with either the keyword `super` or a *Primary* expression. They are used to invoke a constructor of the direct superclass. Superclass constructor invocations may be further subdivided:
  - *Unqualified superclass constructor invocations* begin with the keyword `super`.
  - *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 (§8.1.5). 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 declared in this class or any superclass, or use `this` or `super` in any expression; otherwise, a compile-time error occurs.

For example, if the first constructor of `ColoredPoint` in the example above were changed to:

```
ColoredPoint(int x, int y) {
   this(x, y, color);
}
```

then a compile-time error would occur, because an instance variable cannot be used within a superclass constructor invocation.

If an anonymous class instance creation expression appears within an explicit constructor invocation statement, then the anonymous class may not refer to any of the enclosing instances of the class whose constructor is being invoked.

For example:

```
class Top {
  int x;

  class Dummy {
     Dummy(Object o) {}
  }
  class Inside extends Dummy {
     Inside() {
```

```
            super(new Object() { int r = x; }); // compile-time
    error
          }

        Inside(final int y) {
            super(new Object() { int r = y; }); // correct
        }
    }
}
```

Let *C* be the class being instantiated, let *S* be the direct superclass of *C*, and let *i* be the instance being created. The evaluation of an explicit constructor invocation proceeds as follows:

- First, if the constructor invocation statement is a superclass constructor invocation, then the immediately enclosing instance of *i* with respect to *S* (if any) must be determined. Whether or not *i* has an immediately enclosing instance with respect to *S* is determined by the superclass constructor invocation 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. A compile time error occurs if the superclass constructor invocation is a qualified superclass constructor invocation.
  - Otherwise:
    - If the superclass constructor invocation is qualified, then the the *Primary* expression *p* immediately preceding "`.super`" is evaluated. If the primary expression 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*. Let *O* be the immediately lexically enclosing class of *S*; it is a compile time error if the type of *p* is not *O* or a subclass of *O*.
    - Otherwise:
      - If *S* is a local class (§14.3), then *S* must be declared in a method declared in a lexically enclosing class *O*. Let *O* be 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). If *S* is a member of an enclosing class *O*, then let *O* be 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`.

- Second, the arguments to the constructor are evaluated, left-to-right, as in an ordinary method invocation.
- Next, the constructor is invoked.
- Finally, if the constructor invocation statement is a superclass constructor invocation and the 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`. This action is performed regardless of whether the superclass constructor invocation actually appears as an explicit constructor invocation statement or is provided automatically (§8.9.9). An alternate constructor invocation does not perform this additional implicit action.

8.8.6 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).

8.8.7 Default Constructor

If a class contains no constructor declarations, then a *default constructor* that takes 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 takes no parameters and simply invokes the superclass constructor with no arguments.

A compile-time error occurs if a default constructor is provided by the compiler but the superclass does not have an accessible constructor that takes no arguments.

A default constructor has no `throws` clause.

It follows that if the nullary constructor of the superclass has a `throws` clause, then a compile-time error will occur.

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.

Thus, the example:

```
public class Point {
   int x, y;
}
```

is equivalent to the declaration:

```
public class Point {
   int x, y;
   public Point() { super(); }
}
```

where the default constructor is `public` because the class `Point` is `public`.

The rule that the default constructor of a class has the same access modifier as the class itself is simple and intuitive. Note, however, that this does not imply that the constructor is accessible whenever the class is accessible. Consider

```
package p1;

class Outer {

  protected class Inner{}
}

package p2;

class SonOfOuter {
   void foo() {
      new Inner(); // access error
   }
}
```

The constructor for `Inner` is protected. However, the constructor is protected relative to `Inner`, while `Inner` is protected relative to `Outer`. So, `Inner` is accessible in `SonOfOuter`, since it is a subclass of `Outer`. `Inner`'s constructor is not accessible in `SonOfOuter`, because the class `SonOfOuter` is not a subclass of `Inner`! Hence, even though `Inner` is accessible, it's default constructor is not.

8.8.8 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 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 declaring no constructor that is `public`.

Thus, in the example:

```
class ClassOnly {
   private ClassOnly() { }
   static String just = "only the lonely";
}
```

the class `ClassOnly` cannot be instantiated, while in the example:

```
package just;

public class PackageOnly {
   PackageOnly() { }
   String[] justDesserts = { "cheesecake", "ice cream" };
}
```

the class `PackageOnly` can be instantiated only within the package `just`, in which it is declared.

Bow, bow, ye lower middle classes!
Bow, bow, ye tradesmen, bow, ye masses!
Blow the trumpets, bang the brasses!
Tantantara! Tzing! Boom!

—W. S. Gilbert, *Iolanthe*

CHAPTER 9

Interfaces

My apple trees will never get across
And eat the cones under his pines, I tell him.
He only says "Good Fences Make Good Neighbors."
—Robert Frost, *Mending Wall* (1914)

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.

This chapter discusses the common semantics of all interfaces - top-level (§7.6) and nested (§8.5, §9.5). 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 inherit-

ance 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 object that is an instance of a class declared to implement 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:

InterfaceDeclaration:
InterfaceModifiers^{opt} `interface` *Identifier*
ExtendsInterfaces^{opt} *InterfaceBody*

The *Identifier* in an interface declaration specifies the name of the interface.

9.1.1 Interface Modifiers

An interface declaration may include *interface modifiers*:

InterfaceModifiers:
InterfaceModifier
InterfaceModifiers InterfaceModifier

InterfaceModifier: one of
`public protected private`
`abstract static strictfp`

The access modifier `public` is discussed in §6.6. Not all modifiers are applicable to all kinds of interface declarations. The access modifiers `protected` and `private` pertain only to member interfaces within a directly enclosing class declaration (§8.5) and are discussed in §8.5.1. The access modifier `static` pertains only to member interfaces (§8.5, §9.5). A compile-time error occurs if the same modifier appears more than once in an interface declaration.

9.1.1.1 `abstract` *Interfaces*

Every interface is implicitly `abstract`. This modifier is obsolete and should not be used in new programs.

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).

9.1.2 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` *InterfaceType*
ExtendsInterfaces `,` *InterfaceType*

The following is repeated from §4.2 to make the presentation here clearer:

InterfaceType:
TypeName

Each *InterfaceType* in the `extends` clause of an interface declaration must name an accessible 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 (§8.1.3) on *T*.
- *I* directly depends on an interface *J* that depends on *T* (using this definition recursively).

A compile-time error occurs 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`.

9.1.3 Interface Body and Member Declarations

The body of an interface may declare members of the interface:

InterfaceBody:
`{` *InterfaceMemberDeclarations*^{opt} `}`

InterfaceMemberDeclarations:
InterfaceMemberDeclaration
InterfaceMemberDeclarations InterfaceMemberDeclaration

InterfaceMemberDeclaration:
ConstantDeclaration
AbstractMethodDeclaration
ClassDeclaration
InterfaceDeclaration
`;`

The scope of the name of a member `m` declared in or inherited by an interface type `I` is the entire body of `I`, including any nested type declarations.

9.1.4 Access to Interface Member Names

All interface members are implicitly `public`. They are accessible outside the package where the interface is declared if the interface is also declared `public` and the package containing the interface is accessible as described in §7.4.3.

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 has a public abstract member method *m* with signature *s* and `throws` clause *t* corresponding to each public instance method *m* with signature *s* and `throws` clause *t* declared in `Object`, unless a method with the same signature and a compatible `throws` clause is declared by the interface or one of its superinterfaces.

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.

9.3 Field (Constant) Declarations

The materials of action are variable,
but the use we make of them should be constant.
—Epictetus (circa 60 A.D.),
translated by Thomas Wentworth Higginson

ConstantDeclaration:
ConstantModifiers^{opt} *Type VariableDeclarators*

ConstantModifiers:
ConstantModifier
ConstantModifier ConstantModifers

ConstantModifier: one of
`public static final`

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 possible for an interface to inherit more than one field with the same name (§8.3.3.3). Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the interface to refer to either 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).

A compile-time error occurs 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.

Thus:

```
interface Test {
   float f = j;
   int j = 1;
   int k = k+1;
}
```

causes two compile-time errors, because `j` is referred to in the initialization of `f` before `j` is declared and because the initialization of `k` refers to `k` itself.

(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 (§4.5.5), even by devious programs. See §12.4.2 and §13.4.8 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 a compile-time error occurs.

9.3.2 Examples of Field Declarations

The following example illustrates some (possibly subtle) points about field declarations.

9.3.2.1 *Ambiguous Inherited Fields*

If two fields with the same name are inherited by an interface because, for example, two of its direct superinterfaces declare fields with that name, then a single *ambiguous member* results. Any use of this ambiguous member will result in a compile-time error. Thus in the example:

```
interface BaseColors {
   int RED = 1, GREEN = 2, BLUE = 4;
}
interface RainbowColors extends BaseColors {
   int YELLOW = 3, ORANGE = 5, INDIGO = 6, VIOLET = 7;
}
```

```
interface PrintColors extends BaseColors {
   int YELLOW = 8, CYAN = 16, MAGENTA = 32;
}
interface LotsOfColors extends RainbowColors, PrintColors {
   int FUCHSIA = 17, VERMILION = 43, CHARTREUSE = RED+90;
}
```

the interface `LotsOfColors` inherits two fields named `YELLOW`. This is all right as long as the interface does not contain any reference by simple name to the field `YELLOW`. (Such a reference could occur within a variable initializer for a field.)

Even if interface `PrintColors` were to give the value `3` to `YELLOW` rather than the value `8`, a reference to field `YELLOW` within interface `LotsOfColors` would still be considered ambiguous.

9.3.2.2 *Multiply Inherited Fields*

If a single field is inherited multiple times from the same interface because, for example, both this interface and one of this interface's direct superinterfaces extend the interface that declares the field, then only a single member results. This situation does not in itself cause a compile-time error.

In the example in the previous section, the fields `RED`, `GREEN`, and `BLUE` are inherited by interface `LotsOfColors` in more than one way, through interface `RainbowColors` and also through interface `PrintColors`, but the reference to field `RED` in interface `LotsOfColors` is not considered ambiguous because only one actual declaration of the field `RED` is involved.

9.4 Abstract Method Declarations

AbstractMethodDeclaration:
AbstractMethodModifiers^{opt} *ResultType MethodDeclarator Throws*^{opt} `;`

AbstractMethodModifiers:
AbstractMethodModifier
AbstractMethodModifiers AbstractMethodModifier

AbstractMethodModifier: one of
`public abstract`

The access modifier `public` is discussed in §6.6. A compile-time error occurs if the same modifier appears more than once in an abstract method declaration.

Every method declaration in the body of an interface is implicitly `abstract`, so its body is always represented by a semicolon, not a block.

For compatibility with older versions of the Java platform, it is permitted but discouraged, as a matter of style, to redundantly specify the `abstract` modifier for methods declared in interfaces.

Every method declaration in the body of an interface is implicitly `public`.

It is permitted, but strongly discouraged as a matter of style, to redundantly specify the `public` modifier for interface methods.

Note that a method declared in an interface must not be declared `static`, or a compile-time error occurs, because `static` methods cannot be `abstract`.

Note that a method declared in an interface must not be declared `strictfp` or `native` or `synchronized`, or a compile-time error occurs, 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.

Note that a method declared in an interface must not be declared `final` or a compile-time error occurs. However, a method declared in an interface may be implemented by a method that is declared `final` in a class that implements the interface.

9.4.1 Inheritance and Overriding

If the interface declares a method, then the declaration of that method is said to *override* any and all methods with the same signature in the superinterfaces of the interface.

If a method declaration in an interface overrides the declaration of a method in another interface, a compile-time error occurs if the methods have different return types or if one has a return type and the other is `void`. Moreover, a method declaration must not have a `throws` clause that conflicts (§8.4.4) with that of any method that it overrides; otherwise, a compile-time error occurs.

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 more than one method with the same signature (§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, a compile-time error occurs if, for any two such inherited methods, either they have different return types or one has a return type and the other is `void`. (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 a interface, or one declared and one inherited) have the same name but different signatures, 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.

9.4.3 Examples of Abstract Method Declarations

The following examples illustrate some (possibly subtle) points about abstract method declarations.

9.4.3.1 *Example: Overriding*

Methods declared in interfaces are `abstract` and thus contain no implementation. About all that can be accomplished by an overriding method declaration, other than to affirm a method signature, is to restrict the exceptions that might be thrown by an implementation of the method. Here is a variation of the example shown in §8.4.3.1:

```
class BufferEmpty extends Exception {
   BufferEmpty() { super(); }
   BufferEmpty(String s) { super(s); }
}
class BufferError extends Exception {
   BufferError() { super(); }
   BufferError(String s) { super(s); }
}
public interface Buffer {
   char get() throws BufferEmpty, BufferError;
}
public interface InfiniteBuffer extends Buffer {
    char get() throws BufferError;       // override
}
```

9.4.3.2 *Example: Overloading*

In the example code:

```
interface PointInterface {
   void move(int dx, int dy);
}
interface RealPointInterface extends PointInterface {
   void move(float dx, float dy);
   void move(double dx, double dy);
}
```

the method name `move` is overloaded in interface `RealPointInterface` with three different signatures, two of them declared and one inherited. Any class that implements interface `RealPointInterface` must provide implementations of all three method signatures.

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`.

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*.

An interface may inherit two or more type declarations with the same name. A compile-time error occurs on any 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.

Death, life, and sleep, reality and thought,
Assist me, God, their boundaries to know . . .
—William Wordsworth, *Maternal Grief*

CHAPTER 10

Arrays

Even Solomon in all his glory was not arrayed like one of these.
—Matthew 6:29

JAVA *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 nonnegative 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

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 component type are allowed. The elements of such an array may have as their value a null reference or instances of any type that implements the interface.
- Arrays with an `abstract` class type as the component type are allowed. The elements of such an array may have as their value a null reference or instances of any subclass of the `abstract` class that is not itself `abstract`.

Array types are used in declarations and in cast expressions (§15.16).

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) may create an array, a reference to which then becomes the initial value of the variable.

Because an array's length is not part of its type, a single variable of array type may contain references to arrays of different lengths.

Here are examples of declarations of array variables that do not create arrays:

```
int[] ai;               // array of int
short[][] as;           // array of array of short
Object[]  ao,           // array of Object
          otherAo;      // array of Object
short  s,               // scalar short
       aas[][];         // array of array of short
```

Here are some examples of declarations of array variables that create array objects:

```
Exception ae[] = new Exception[3];
Object aao[][] = new Exception[2][3];
int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };
char ac[] = { 'n', 'o', 't', ' ', 'a', ' ',
              'S', 't', 'r', 'i', 'n', 'g' };
String[] aas = { "array", "of", "String", };
```

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, as in this example:

```
byte[] rowvector, colvector, matrix[];
```

This declaration is equivalent to:

```
byte rowvector[], colvector[], matrix[][];
```

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.

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 §10.11 for a discussion.

10.3 Array Creation

An array is created by an array creation expression (§15.10) or an array initializer (§10.7).

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`.

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 Arrays: A Simple Example

The example:

```
class Gauss {
   public static void main(String[] args) {
      int[] ia = new int[101];
      for (int i = 0; i < ia.length; i++)
         ia[i] = i;
      int sum = 0;
      for (int i = 0; i < ia.length; i++)
         sum += ia[i];
      System.out.println(sum);
   }
}
```

that produces output:

```
5050
```

declares a variable `ia` that has type array of `int`, that is, `int[]`. The variable `ia` is initialized to reference a newly created array object, created by an array creation expression (§15.10). The array creation expression specifies that the array should have `101` components. The length of the array is available using the field `length`, as shown.

The example program fills the array with the integers from `0` to `100`, sums these integers, and prints the result.

10.6 Array Initializers

An *array initializer* may be specified in a declaration, or as part of an array creation expression (§15.10), creating an array and providing some initial values:

ArrayInitializer:
`{` *VariableInitializers*^{opt} `,`^{opt} `}`

VariableInitializers:
VariableInitializer
VariableInitializers , *VariableInitializer*

The following is repeated from §8.3 to make the presentation here clearer:

VariableInitializer:
Expression
ArrayInitializer

An array initializer is written as a comma-separated list of expressions, enclosed by braces "{" and "}".

The length of the constructed array will equal the number of expressions.

The expressions in an array initializer are 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*st array component. Each expression must be assignment-compatible (§5.2) with the array's component type, or a compile-time error results.

If the component type is itself an array type, then the expression specifying a component may itself be an array initializer; that is, array initializers may be nested.

A trailing comma may appear after the last expression in an array initializer and is ignored.

As an example:

```
class Test {
   public static void main(String[] args) {
      int ia[][] = { {1, 2}, null };
      for (int i = 0; i < 2; i++)
         for (int j = 0; j < 2; j++)
            System.out.println(ia[i][j]);
   }
}
```

prints:

```
1
2
```

before causing a `NullPointerException` in trying to index the second component of the array `ia`, which is a null reference.

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

All the members inherited from class `Object`; the only method of `Object` that is not inherited is its `clone` method

An array thus has the same public fields and methods as the following class:

```
class A implements Cloneable, java.io.Serializable {
   public final int length = x;
   public Object clone() {
      try {
         return super.clone();
      } catch (CloneNotSupportedException e) {
         throw new InternalError(e.getMessage());
      }
   }
}
```

Every array implements the interfaces `Cloneable` and `java.io.Serializable`.

That arrays are cloneable is shown by the test program:

```
class Test {
   public static void main(String[] args) {
      int ia1[] = { 1, 2 };
      int ia2[] = (int[])ia1.clone();
      System.out.print((ia1 == ia2) + " ");
      ia1[1]++;
      System.out.println(ia2[1]);
   }
}
```

which prints:

```
false 2
```

showing that the components of the arrays referenced by `ia1` and `ia2` are different variables. (In some early implementations of Java this example failed to compile because the compiler incorrectly believed that the clone method for an array could throw a `CloneNotSupportedException`.)

A `clone` of a multidimensional array is shallow, which is to say that it creates only a single new array. Subarrays are shared.

This is shown by the example program:

```
class Test {
   public static void main(String[] args) throws Throwable {
      int ia[][] = { { 1 , 2}, null };
      int ja[][] = (int[][])ia.clone();
      System.out.print((ia == ja) + " ");
      System.out.println(ia[0] == ja[0] && ia[1] == ja[1]);
   }
}
```

which prints:

```
false true
```

showing that the `int[]` array that is `ia[0]` and the `int[]` array that is `ja[0]` are the same array.

10.8 `Class` Objects for Arrays

Every array has an associated `Class` object, shared with all other arrays with the same component type. The superclass of an array type is considered to be `Object`, as shown by the following example code:

```
class Test {
   public static void main(String[] args) {
      int[] ia = new int[3];
      System.out.println(ia.getClass());
      System.out.println(ia.getClass().getSuperclass());
   }
}
```

which prints:

```
class [I
class java.lang.Object
```

where the string "`[I`" is the run-time type signature for the class object "array with component type `int`".

10.9 An Array of Characters is Not a `String`

In Java, 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 Java `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.

10.10 Array Store Exception

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*.

Thus, the example:

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }


class Test {
   public static void main(String[] args) {
      ColoredPoint[] cpa = new ColoredPoint[10];
      Point[] pa = cpa;
      System.out.println(pa[1] == null);
      try {
         pa[0] = new Point();
      } catch (ArrayStoreException e) {
         System.out.println(e);
      }
   }
}
```

produces the output:

```
true
java.lang.ArrayStoreException
```

Here the variable `pa` has type `Point[]` and the variable `cpa` has as its value a reference to an object of type `ColoredPoint[]`. A `ColoredPoint` can be assigned to a `Point`; therefore, the value of `cpa` can be assigned to `pa`.

A reference to this array `pa`, for example, testing whether `pa[1]` is `null`, will not result in a run-time type error. This is because the element of the array of type `ColoredPoint[]` is a `ColoredPoint`, and every `ColoredPoint` can stand in for a `Point`, since `Point` is the superclass of `ColoredPoint`.

On the other hand, an assignment to the array `pa` can result in a run-time error. At compile time, an assignment to an element of `pa` is checked to make sure that the value assigned is a `Point`. But since `pa` holds a reference to an array of `ColoredPoint`, the assignment is valid only if the type of the value assigned at run-time is, more specifically, a `ColoredPoint`.

Java checks for such a situation at run-time to ensure that the assignment is valid; if not, an `ArrayStoreException` is thrown. More formally: 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`.

At length burst in the argent revelry,
With plume, tiara, and all rich array . . .
—John Keats, *The Eve of St. Agnes* (1819)

CHAPTER 11

Exceptions

If anything can go wrong, it will.
—Finagle's Law
(often incorrectly attributed to Murphy, whose law is rather different—which only goes to show that Finagle was right)

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 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 (§14.17).

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 (§14.19). During the process of throwing an exception, the Java virtual machine abruptly completes, one by one, any expres-

sions, 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 method `uncaughtException` is invoked for the `ThreadGroup` that is the parent of the current thread—thus every effort is made to avoid letting an exception go unhandled.

The exception mechanism of the Java platform is integrated with its synchronization model (§17), so that locks are released as `synchronized` statements (§14.18) and invocations of `synchronized` methods (§8.4.3.6, §15.12) complete abruptly.

This chapter describes the different causes of exceptions (§11.1). It details how exceptions are checked at compile time (§11.2) and processed at run time (§11.4). A detailed example (§11.5) is then followed by an explanation of the exception hierarchy(§11.6).

11.1 The Causes of Exceptions

If we do not succeed, then we run the risk of failure.
—J. Danforth Quayle (1990)

An exception is thrown for one of three *reasons*:

- 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, such as an integer divide by zero, as summarized in §15.6
  - an error occurs in loading or linking part of the program (§12.2, §12.3)
  - some limitation on a resource is exceeded, such as using too much memory

  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.
- A `throw` statement (§14.17) was executed.
- An asynchronous exception occurred either because:
  - the method `stop` of class `Thread` was invoked
  - an internal error has occurred in the virtual machine (§11.6.4.3)

Exceptions are represented by instances of the class `Throwable` and instances of its subclasses. These classes are, collectively, the *exception classes*.

11.2 Compile-Time Checking of Exceptions

A compiler for the Java programming language checks, at compile time, that a program contains handlers for *checked exceptions*, by analyzing which checked exceptions can result from execution of a method or constructor. For each checked exception which is a possible result, the `throws` clause for the method (§8.4.4) or constructor (§8.8.4) 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 *unchecked exceptions classes* are the class `RuntimeException` and its subclasses, and the class `Error` and its subclasses. All other exception classes are *checked exception classes*. The Java API defines a number of exception classes, both checked and unchecked. Additional exception classes, both checked and unchecked, may be declared by programmers. See §11.6 for a description of the exception class hierarchy and some of the exception classes defined by the Java API and Java virtual machine.

The checked exception classes 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. 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).

Variable initializers for fields (§8.3.2) and static initializers (§8.7) must not result in a checked exception; if one does, a compile-time error occurs.

11.2.1 Why Errors are Not Checked

Those unchecked exception classes which are the *error classes* (`Error` and its subclasses) are exempted from compile-time checking because they can occur at many points in the program and recovery from them is difficult or impossible. A program declaring such exceptions would be cluttered, pointlessly.

11.2.2 Why Runtime Exceptions are Not Checked

The *runtime exception classes* (`RuntimeException` and its subclasses) are exempted from compile-time checking because, in the judgment of the designers of the Java programming language, having to declare such exceptions would not aid significantly in establishing the correctness of programs. Many of the operations and constructs of the Java programming language can result in runtime exceptions. The information available to a compiler, and the level of analysis the compiler performs, are usually not sufficient to establish that such runtime exceptions cannot occur, even though this may be obvious to the programmer. Requiring such exception classes to be declared would simply be an irritation to programmers.

For example, certain code might implement a circular data structure that, by construction, can never involve `null` references; the programmer can then be certain that a `NullPointerException` cannot occur, but it would be difficult for a compiler to prove it. The theorem-proving technology that is needed to establish such global properties of data structures is beyond the scope of this specification.

11.3 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 of a `try` statement (§14.19) that handles 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.

Whether a particular `catch` clause *handles* 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 handles the exception if the type of its parameter is the class of the exception or a superclass of the class of the exception. Equivalently, a `catch` clause will catch any exception object that is an `instanceof` (§15.20.2) the declared parameter type.

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.

If no `catch` clause handling an exception can be found, then the current thread (the thread that encountered the exception) is terminated, but only after all `finally` clauses have been executed and the method `uncaughtException` has been invoked for the `ThreadGroup` that is the parent of the current thread.

In situations where it is desirable to ensure that one block of code is always executed after another, even if that other block of code completes abruptly, a `try` statement with a `finally` clause (§14.19.2) may be used.

If a `try` or `catch` block in a `try`–`finally` or `try`–`catch`–`finally` statement completes abruptly, then the `finally` clause is executed during propagation of the exception, even if no matching `catch` clause is ultimately found. If a `finally` clause is executed because of abrupt completion of a `try` block and the `finally` clause itself completes abruptly, then the reason for the abrupt completion of the `try` block is discarded and the new reason for abrupt completion is propagated from there.

The exact rules for abrupt completion and for the catching of exceptions are specified in detail with the specification of each statement in §14 and for expressions in §15 (especially §15.6).

11.3.1 Exceptions are Precise

Exceptions 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 optimized code has speculatively executed some of the expressions or statements which follow the point at which the exception occurs, such code must be prepared to hide this speculative execution from the user-visible state of the program.

11.3.2 Handling 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.

Proper understanding of the semantics of asynchronous exceptions is necessary if high-quality machine code is to be generated.

Asynchronous exceptions are rare. They occur only as a result of:

- An invocation of the `stop` methods of class `Thread` or `ThreadGroup`
- An internal error (§11.6.4.3) in the Java virtual machine

The `stop` methods may be invoked by one thread to affect another thread or all the threads in a specified thread group. They are asynchronous because they may occur at any point in the execution of the other thread or threads. An `InternalError` is considered asynchronous.

The Java platform permits a small but bounded amount of execution to occur before an asynchronous exception is thrown. This delay is permitted to allow optimized code to detect and throw these exceptions at points where it is practical to handle them while obeying the semantics of the Java programming language.

A simple implementation might poll for asynchronous exceptions at the point of each control transfer instruction. Since a program has a finite size, this provides a bound on the total delay in detecting an asynchronous exception. Since no asynchronous exception will occur between control transfers, the code generator has some flexibility to reorder computation between control transfers for greater performance.

The paper *Polling Efficiently on Stock Hardware* by Marc Feeley, *Proc. 1993 Conference on Functional Programming and Computer Architecture*, Copenhagen, Denmark, pp. 179–187, is recommended as further reading.

Like all exceptions, asynchronous exceptions are precise (§11.4.1).

11.4 An Example of Exceptions

Consider the following example:

```
class TestException extends Exception {
   TestException() { super(); }
   TestException(String s) { super(s); }
}
```

```
class Test {
   public static void main(String[] args) {
      for (int i = 0; i < args.length; i++) {


         try {
            thrower(args[i]);
            System.out.println("Test \"" + args[i] +
               "\" didn't throw an exception");
         } catch (Exception e) {
            System.out.println("Test \"" + args[i] +
               "\" threw a " + e.getClass() +
               "\n    with message: " + e.getMessage());
         }
      }
   }
   static int thrower(String s) throws TestException {
      try {
         if (s.equals("divide")) {
            int i = 0;
            return i/i;
         }
         if (s.equals("null")) {
            s = null;
            return s.length();
         }
         if (s.equals("test"))
            throw new TestException("Test message");
         return 0;
      } finally {
         System.out.println("[thrower(\"" + s +
               "\") done]");
      }
   }

}
```

If we execute the test program, passing it the arguments:

```
divide null not test
```

it produces the output:

```
[thrower("divide") done]
Test "divide" threw a class java.lang.ArithmeticException
    with message: / by zero
[thrower("null") done]
Test "null" threw a class java.lang.NullPointerException
    with message: null
```

```
[thrower("not") done]
Test "not" didn't throw an exception
[thrower("test") done]
Test "test" threw a class TestException
    with message: Test message
```

This example declares an exception class `TestException`. The `main` method of class `Test` invokes the `thrower` method four times, causing exceptions to be thrown three of the four times. The `try` statement in method `main` catches each exception that the `thrower` throws. Whether the invocation of `thrower` completes normally or abruptly, a message is printed describing what happened.

The declaration of the method `thrower` must have a `throws` clause because it can throw instances of `TestException`, which is a checked exception class (§11.2). A compile-time error would occur if the `throws` clause were omitted.

Notice that the `finally` clause is executed on every invocation of `thrower`, whether or not an exception occurs, as shown by the "`[thrower(...) done]`" output that occurs for each invocation

11.5 The Exception Hierarchy

The possible exceptions in a program are organized in a hierarchy of classes, rooted at class `Throwable` (§11.6), a direct subclass of `Object`. The classes `Exception` and `Error` are direct subclasses of `Throwable`. The class `RuntimeException` is a direct subclass of `Exception`.

Programs can use the pre-existing exception classes in `throw` statements, or define additional exception classes, as subclasses of `Throwable` or of any of its subclasses, as appropriate. To take advantage of the Java platform's compile-time checking for exception handlers, it is typical to define most new exception classes as checked exception classes, specifically as subclasses of `Exception` that are not subclasses of `RuntimeException`.

The class `Exception` is the superclass of all the exceptions that ordinary programs may wish to recover from. The class `RuntimeException` is a subclass of class `Exception`. The subclasses of `RuntimeException` are unchecked exception classes. The subclasses of `Exception` other than `RuntimeException` are all checked exception classes.

The class `Error` and its subclasses are exceptions from which ordinary programs are not ordinarily expected to recover. See the Java API specification for a detailed description of the exception hierarchy.

The class `Error` is a separate subclass of `Throwable`, distinct from `Exception` in the class hierarchy, to allow programs to use the idiom:

```
} catch (Exception e) {
```

to catch all exceptions from which recovery may be possible without catching errors from which recovery is typically not possible.

11.5.0.1 *Loading and Linkage Errors*

The Java virtual machine throws an object that is an instance of a subclass of `LinkageError` when a loading, linkage, preparation, verification or initialization error occurs:

- The loading process is described in §12.2. .
- The linking process is described in §12.3.
- The class verification process is described in §12.3.1.
- The class preparation process is described in §12.3.2.
- The class initialization process is described in §12.4.

11.5.0.2 *Virtual Machine Errors*

The Java virtual machine throws an object that is an instance of a subclass of the class `VirtualMachineError` when an internal error or resource limitation prevents it from implementing the semantics of the Java programming language. See *The Java Virtual Machine Specification Second Edition* for the definitive discussion of these errors.

I

I never forget a face—but in your case I'll be glad to make an exception.
—Groucho Marx

CHAPTER 12

Execution

We must all hang together, or assuredly we shall all hang separately.
—Benjamin Franklin (July 4, 1776)

THIS chapter specifies activities that occur during execution of a program. It is organized around the life cycle of a Java Virtual Machine and of the classes, interfaces, and objects that form a program.

A 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 (§12.4), and initialization (§12.5).

The chapter continues with a specification of the procedures for creation of new class instances (§12.6); and finalization of class instances (§12.7). It concludes by describing the unloading of classes (§12.11) and the procedure followed when a program exits (§12.12).

12.1 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 virtual machine start-up are given in chapter 5 of *The Java Virtual Machine Specification, Second Edition*. 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. For example, in a UNIX implementation, the command line:

```
java Test reboot Bob Dot Enzo
```

will typically start a Java Virtual Machine by invoking method main of class Test (a class in an unnamed package), passing it an array containing the four strings "reboot", "Bob", "Dot", and "Enzo".

We now outline the steps the 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 virtual machine does not currently contain a binary representation for this class. The 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 §12.4.

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 §12.4.1.

Preparation involves allocation of static storage and any data structures that are used internally by the virtual machine, such as method tables. Preparation is described further in §12.4.2.

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 §12.4.3.

12.1.3 Initialize Test: Execute Initializers

In our continuing example, the virtual machine is still trying to execute the method main of class Test. This is permitted only if the class has been initialized (§12.5.1).

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 §12.5.

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 strings.

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 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, Second Edition*. 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 classloader 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. For further discussion of these issues, see *The Java Virtual*

Machine Specification, Second Edition and the paper *Dynamic Class Loading in the Java Virual 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. A basic principle of the design of the Java programming language is that the type system cannot be subverted by code written in the language, not even by implementations of such otherwise sensitive system classes as `ClassLoader` and `SecurityManager`.

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 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, Second Edition*. 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.

For the specification of the verification process, see the separate volume of this series, *The Java Virtual Machine Specification*.

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, §13.4.8, and §13.4.15 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 (§4.5.5). This does not require the execution of any source code; explicit initializers for `static` fields are executed as part of initialization (§12.5), not preparation.

Implementations of the Java Virtual Machine may precompute additional data structures at preparation time in order to make later operations on a class or interface more efficient. One particularly useful data structure is a "method table" or other data structure that allows any method to be invoked on instances of a class without requiring a search of superclasses at invocation time.

12.3.3 Resolution of Symbolic References

A Java binary file references other classes and interfaces and their fields, methods, and constructors symbolically, using the fully-qualified names of the other classes and interfaces (§13.1). For fields and methods, these symbolic references include the name of the class or interface type that declares the field or method 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 `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 (§13.4.6).

- `InstantiationError`: A symbolic reference has been encountered that is used in a 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 (§13.4.7).

- `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 (§13.4.11).

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 the virtual machine (§12.4).

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 there.

Before a class is initialized, its 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

Initialization of a class consists of executing its static initializers and the initializers for static fields declared in the class. *Initialization* of an interface consists of executing the initializers for fields declared in the interface.

Before a class is initialized, its direct superclass must be initialized, but interfaces implemented by the class need not be initialized. Similarly, the superinterfaces of an interface need not be initialized before the interface is initialized.

A class or interface type *T* will be initialized immediately before the first occurence 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 reference to the field is not a compile-time constant (§15.28). Such a reference must be resolved at compile time to a copy of the compile-time constant value, so uses of such a field never cause initialization.

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 (§8.7). This restriction is designed to detect, at compile time, most circular or otherwise malformed initializations.

As shown in an example in §8.7, the fact that initialization code is unrestricted allows examples to be constructed 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 §12.6). 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 (§12.5.3).

Before a class is initialized, its superclasses are initialized, if they have not previously been initialized.

Thus, the test program:

```
class Super {
   static { System.out.print("Super "); }
}

class One {
   static { System.out.print("One "); }
}

class Two extends Super {
   static { System.out.print("Two "); }
}
```

```
class Test {
   public static void main(String[] args) {
      One o = null;
      Two t = new Two();
      System.out.println((Object)o == (Object)t);
   }
}
```

prints :

```
Super Two false
```

The class One is never initialized, because it not used actively and therefore is never linked to. The class Two is initialized only after its superclass Super has 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.

The test program:

```
class Super { static int taxi = 1729; }
class Sub extends Super {
   static { System.out.print("Sub "); }
}

class Test {
   public static void main(String[] args) {
      System.out.println(Sub.taxi);
   }
}
```

prints only:

```
1729
```

because the class Sub is never initialized; the reference to Sub.taxi is a reference to a field actually declared in class Super and does not trigger initialization of the class Sub.

Initialization of an interface does not, of itself, cause initialization of any of its superinterfaces.

Thus, the test program:

```
interface I {
   int i = 1, ii = Test.out("ii", 2);
}

interface J extends I {
   int j = Test.out("j", 3), jj = Test.out("jj", 4);
}
```

```
interface K extends J {
   int k = Test.out("k", 5);
}

class Test {
   public static void main(String[] args) {
      System.out.println(J.i);
      System.out.println(K.j);
   }

   static int out(String s, int i) {
      System.out.println(s + "=" + i);
      return i;
   }

}
```

produces the output:

```
1
j=3
jj=4
3
```

The reference to `J.i` is to a field that is a compile-time constant; therefore, it does not cause `I` to be initialized. The reference to `K.j` is a reference to a field actually declared in interface `J` that is not a compile-time constant; this causes initialization of the fields of interface `J`, but not those of its superinterface `I`, nor those of interface `K`. Despite the fact that the name `K` is used to refer to field `j` of interface `J`, interface `K` is not initialized.

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 the verification or preparation step failed, or because initialization was attempted and failed.

The procedure for initializing a class or interface is then as follows:

1. Synchronize (§14.18) on the `Class` object that represents the class or interface to be initialized. This involves waiting until the current thread can obtain the lock for that object (§17.13).
2. If initialization is in progress for the class or interface by some other thread, then `wait` on this `Class` object (which temporarily releases the lock). When the current thread awakens from the `wait`, repeat this step.
3. If initialization is in progress for the class or interface by the current thread, then this must be a recursive request for initialization. Release the lock on the `Class` object and complete normally.
4. If the class or interface has already been initialized, then no further action is required. Release the lock on the `Class` object and complete normally.
5. If the `Class` object is in an erroneous state, then initialization is not possible. Release the lock on the `Class` object and throw a `NoClassDefFoundError`.
6. Otherwise, record the fact that initialization of the `Class` object is now in progress by the current thread and release the lock on the `Class` object.
7. Next, if the `Class` object represents a class rather than an interface, and the superclass of this class has not yet been initialized, then recursively perform this entire procedure for the superclass. If necessary, verify and prepare the superclass first. If the initialization of the superclass completes abruptly because of a thrown exception, then lock this `Class` object, label it erroneous, notify all waiting threads , release the lock, and complete abruptly, throwing the same exception that resulted from initializing the superclass.
8. 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, except that `final` class variables and fields of interfaces whose values are compile-time constants are initialized first (§8.3.2.1, §9.3.1, §13.4.8).
9. If the execution of the initializers completes normally, then lock this `Class` object, label it fully initialized, notify all waiting threads, release the lock, and complete this procedure normally.

10. 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.

11. Lock the `Class` object, label it erroneous, notify all waiting threads, release the lock, and complete this procedure abruptly with reason *E* or its replacement as determined in the previous step.

(Due to a flaw in some early implementations, a exception during class initialization was ignored, rather than causing an `ExceptionInInitializerError` as described here.)

12.4.3 Initialization: Implications for Code Generation

Code generators need to preserve the points of possible initialization of a class or interface, inserting an invocation of the initialization procedure just described. If this initialization procedure completes normally and the `Class` object is fully initialized and ready for use, then the invocation of the initialization procedure is no longer necessary and it may be eliminated from the code—for example, by patching it out or otherwise regenerating the code.

Compile-time analysis may, in some cases, be able to eliminate many of the checks that a type has been initialized from the generated code, if an initialization order for a group of related types can be determined. Such analysis must, however, fully account for concurrency and for the fact that initialization code is unrestricted.

12.5 Creation of New Class Instances

A new class instance is explicitly created when one of the following situations occurs:

- 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 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 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. 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 (§4.5.5). 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 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. (In some early implementations, the compiler incorrectly omitted the code to initialize a field if the field initializer expression was a constant expression whose value was equal to the default initialization value for its type.)

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.

In the example:

```
class Point {
   int x, y;
   Point() { x = 1; y = 1; }
}

class ColoredPoint extends Point {
   int color = 0xFF00FF;
}

class Test {
   public static void main(String[] args) {
      ColoredPoint cp = new ColoredPoint();
      System.out.println(cp.color);
   }
}
```

a new instance of `ColoredPoint` is created. First, space is allocated for the new `ColoredPoint`, to hold the fields `x`, `y`, and `color`. All these fields are then initialized to their default values (in this case, `0` for each field). Next, the `ColoredPoint` constructor with no arguments is first invoked. Since `ColoredPoint` declares no constructors, a default constructor of the form:

```
ColoredPoint() { super(); }
```

is provided for it automatically by the Java compiler.

This constructor then invokes the `Point` constructor with no arguments. The `Point` constructor does not begin with an invocation of a constructor, so the compiler provides an implicit invocation of its superclass constructor of no arguments, as though it had been written:

```
Point() { super(); x = 1; y = 1; }
```

Therefore, the constructor for `Object` which takes no arguments is invoked.

The class `Object` has no superclass, so the recursion terminates here. Next, any instance initializers, instance variable initializers of `Object` are invoked.

Next, the body of the constructor of `Object` that takes no arguments is executed. No such constructor is declared in `Object`, so the compiler supplies a default one, which in this special case is:

```
Object() { }
```

This constructor executes without effect and returns.

Next, all initializers for the instance variables of class `Point` are executed. As it happens, the declarations of `x` and `y` do not provide any initialization expressions, so no action is required for this step of the example. Then the body of the `Point` constructor is executed, setting `x` to `1` and `y` to `1`.

Next, the initializers for the instance variables of class `ColoredPoint` are executed. This step assigns the value `0xFF00FF` to `color`. Finally, the rest of the body of the `ColoredPoint` constructor is executed (the part after the invocation of `super`); there happen to be no statements in the rest of the body, so no further action is required and initialization is complete.

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. Thus, compiling and running the example:

```
class Super {
   Super() { printThree(); }
   void printThree() { System.out.println("three"); }
}


class Test extends Super {
   int three = (int)Math.PI;                  // That is, 3
   public static void main(String[] args) {
      Test t = new Test();
      t.printThree();
   }
   void printThree() { System.out.println(three); }
}
```

produces the output:

```
0
3
```

This shows that the invocation of `printThree` in the constructor for class `Super` does not invoke the definition of `printThree` in class `Super`, but rather invokes the overriding definition of `printThree` in class `Test`. This method therefore runs before the field initializers of `Test` have been executed, which is why the first value output is `0`, the default value to which the field `three` of `Test` is initialized. The later invocation of `printThree` in method `main` invokes the same definition of `printThree`, but by that point the initializer for instance variable `three` has been executed, and so the value `3` is printed.

See §8.8 for more details on constructor declarations.

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 (such as file descriptors or operating system graphics contexts) 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. If an uncaught exception is thrown during the finalization, the exception is ignored and finalization of that object terminates.

The `finalize` method declared in class `Object` takes no action.

However, 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, which is usually good practice. (Unlike constructors, finalizers do not automatically invoke the finalizer for the superclass; such an invocation must be coded explicitly.)

For efficiency, an implementation may keep track of classes that do not override the `finalize` method of class `Object`, or override it in a trivial way, such as:

```
protected void finalize() throws Throwable {
    super.finalize();
}
```

We encourage implementations to treat such objects as having a finalizer that is not overridden, and to finalize them more efficiently, as described in §12.7.1.

A finalizer may be invoked explicitly, just like any other method.

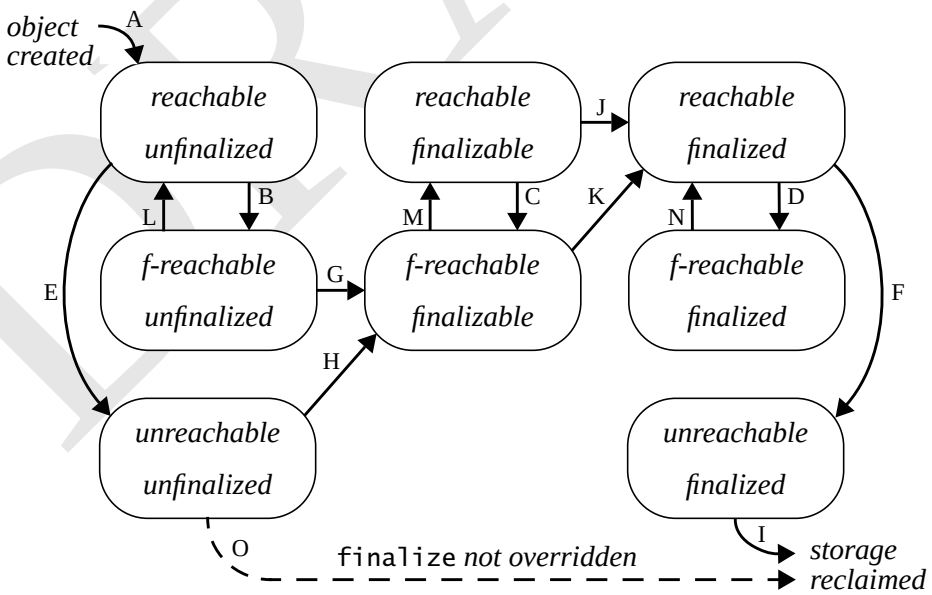
12.6.1 Implementing Finalization

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. Optimizing transformations of a program can be designed that reduce the number of objects that are reachable to be less than those which would naively be considered reachable. For example, a compiler or code generator may choose, explicitly or implicitly, to set a variable or parameter that will no longer be used to `null` to cause the storage for such an object to be potentially reclaimable sooner. 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.

The life cycle of an object obeys the following transition diagram, where we abbreviate "finalizer-reachable" as "f-reachable":



When an object is first created (A), it is reachable and unfinalized.

As references to an object are discarded during program execution, an object that was reachable may become finalizer-reachable (B, C, D) or unreachable (E, F). (Note that a finalizer-reachable object never becomes unreachable directly; it becomes reachable when the finalizer from which it can be reached is invoked, as explained below.)

If the Java Virtual Machine detects that an unfinalized object has become finalizer-reachable or unreachable, it may label the object finalizable (G, H); moreover, if the object was unreachable, it becomes finalizer-reachable (H).

If the Java Virtual Machine detects that a finalized object has become unreachable, it may reclaim the storage occupied by the object because the object will never again become reachable (I).

At any time, a Java Virtual Machine may take any finalizable object, label it finalized, and then invoke its `finalize` method in some thread. This causes the object to become finalized and reachable (J, K), and it also may cause other objects that were finalizer-reachable to become reachable again (L, M, N).

A finalizable object cannot also be unreachable; it can be reached because its finalizer may eventually be invoked, whereupon the thread running the finalizer will have access to the object, as `this` (§15.8.3). Thus, there are actually only eight possible states for an object.

After an object has been finalized, no further action is taken until the automatic storage management determines that it is unreachable. Because of the way that an object progresses from the *unfinalized* state through the *finalizable* state to the *finalized* state, the `finalize` method is never automatically invoked more than once by a Java Virtual Machine for each object, even if the object is again made reachable after it has been finalized.

Explicit invocation of a finalizer ignores the current state of the object and does not change the state of the object from unfinalized or finalizable to finalized.

If a class does not override method `finalize` of class `Object` (or overrides it in only a trivial way, as described above), then if instances of such a class become unreachable, they may be discarded immediately rather than made to await a second determination that they have become unreachable. This strategy is indicated by the dashed arrow (O) in the transition diagram.

Programmers should also be aware that a finalizer can be automatically invoked, even though it is reachable, during finalization-on-exit (§12.12); moreover, a finalizer can also be invoked explicitly as an ordinary method.

Therefore, we recommend that the design of `finalize` methods be kept simple and that they be programmed defensively, so that they will work in all cases.

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.

As an example, if a circularly linked group of unfinalized objects becomes unreachable (or finalizer-reachable), then all the objects may become finalizable together. Eventually, the finalizers for these objects may be invoked, in any order, or even concurrently using multiple threads. If the automatic storage manager later finds that the objects are unreachable, then their storage can be reclaimed.

It is straightforward to implement a class that will cause a set of finalizer-like methods to be invoked in a specified order for a set of objects when all the objects become unreachable. Defining such a class is left as an exercise for the reader.

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 class loader is unreachable (§12.7.1). Classes 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 .

.

. . . Farewell!
The day frowns more and more. Thou'rt like to have
A lullaby too rough: I never saw
The heavens so dim by day: A savage clamour!
Well may I get aboard! This is the chase.
I am gone for ever!

[Exit, pursued by a bear]

—William Shakespeare, *The Winter's Tale*, Act III, scene iii

CHAPTER 13

Binary Compatibility

Despite all of its promise, software reuse in object-oriented programming has yet to reach its full potential. A major impediment to reuse is the inability to evolve a compiled class library without abandoning the support for already compiled applications. . . . [A]n object-oriented model must be carefully designed so that class-library transformations that should not break already compiled applications, indeed, do not break such applications.

—Ira Forman, Michael Conner, Scott Danforth, and Larry Raper, Release-to-Release Binary Compatibility in SOM (1995)

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 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 or interface.
- 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 platform 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 (§13.6), 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 proprerties are:

- 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 the binary name of its immediately enclosing type, followed by $ followed by the simple name of the member.
  - The binary name of any nested type must have, as a prefix, the binary name of its enclosing top-level class.
- A reference to another class or interface type must be symbolic, using the binary name of the type.
- Given a legal expression denoting a field access in a class *C*, referencing a 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 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 lexically enclosing class of which *f* is a member. *T* is the qualifying type of the reference.

  The reference to *f* must be compiled into a symbolic reference to the qualifying type of the reference, plus the simple name of the field, *f*. The reference must also include a symbolic reference to the declared type of the field so that the verifier can check that the type is as expected. References to fields that are

`final` and initialized with compile-time constant expressions are resolved at compile time to the constant value that is denoted. No reference to such a constant field should be present in the code in a binary file (except in the class or interface containing the constant field, which will have code to initialize it), and such constant fields must always appear to have been initialized; the default initial value for the type of such a field must never be observed. See §13.4.8 for a discussion.

- Given a method invocation expression in a class or interface *C* referencing a method named *m* declared 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 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*, let *T* be *C*. Otherwise, let *T* be the lexically enclosing class of which *m* is a member. *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 qualifying type of the invocation, plus the signature of the method (§8.4.2). A reference to a method must also include either a symbolic reference to 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 parameters to the method
- A symbolic reference to the type of each parameter

- Given a class instance creation expression (§15.9) or a constructor invocation statement (§8.8.5.1) 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 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 to the constructor
  - A symbolic reference to the type of each 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).
- Any constructs introduced by the compiler that do not have a corresponding construct in the a source code must be marked as synthetic, except for default constructors and the class initialization method.

A binary representation for a class or interface must also contain all of the following:

- If it is a class and is not class `Object`, then a symbolic reference to the direct superclass of this class
- A symbolic reference to each direct superinterface, if any
- 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 type of the field
- If it is a class, then the signature of each constructor, as described above

- For each method declared in the class or interface, its signature and return type, as described above
- 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
- Every type must contain sufficient information to recover its canonical name (§6.7).
- Every member type must have sufficient information to recover its source level access modifier.
- Every nested class must have a symbolic reference to its immediately enclosing class.
- 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) preexisting binaries if preexisting 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 preexisting binaries. The method signature that the

preexisting binary will use for method lookup is chosen by the method overload resolution algorithm at compile time (§15.12.2). (If the language had been designed so that the particular method to be executed was chosen at run time, then such an ambiguity might be detected at run time. Such a rule would imply that adding an additional overloaded method so as to make ambiguity possible at a call site could break compatibility with an unknown number of preexisting binaries. See §13.4.24 for more discussion.)

Binary compatibility is not the same as source compatibility. In particular, the example in §13.4.5 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 preexisting 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 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 was 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 was 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 was 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 link-time 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.

For example, suppose that the following test program:

```
class Hyper { char h = 'h'; }

class Super extends Hyper { char s = 's'; }

class Test extends Super {
    public static void main(String[] args) {
        Hyper h = new Super();
        System.out.println(h.h);
    }
}
```

is compiled and executed, producing the output:

```
h
```

Suppose that a new version of class `Super` is then compiled:

```
class Super { char s = 's'; }
```

This version of class `Super` is not a subclass of `Hyper`. If we then run the existing binaries of `Hyper` and `Test` with the new version of `Super`, then a `VerifyError` is thrown at link time. The verifier objects because the result of `new Super()` cannot be assigned to a variable of type `Hyper`, because `Super` is not a subclass of `Hyper`.

It is instructive to consider what might happen without the verification step: the program might run and print:

```
s
```

This demonstrates that without the verifier the type system could be defeated by linking inconsistent binary files, even though each was produced by a correct Java compiler.

The lesson is that an implementation that lacks a verifier or fails to use it will not maintain type safety and is, therefore, not a valid implementation.

13.4.5 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, accessibility, (for fields) or same name, accessibility, 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. If the program:

```
class Hyper {
   void hello() { System.out.println("hello from Hyper"); }
}

class Super extends Hyper {
   void hello() { System.out.println("hello from Super"); }
}

class Test {
   public static void main(String[] args) {
      new Super().hello();
   }
}
```

is compiled and executed, it produces the output:

```
hello from Super
```

Suppose that a new version of class `Super` is produced:

```
class Super extends Hyper { }
```

then recompiling `Super` and executing this new binary with the original binaries for `Test` and `Hyper` produces the output:

```
hello from Hyper
```

as expected.

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`. Thus, if the program:

```
class Hyper {
   void hello() { System.out.println("hello from Hyper"); }
}

class Super extends Hyper { }

class Test extends Super {
```

```
    public static void main(String[] args) {
        new Test().hello();
    }

    void hello() {
        super.hello();
    }

}
```

is compiled and executed, it produces the output:

```
hello from Hyper
```

Suppose that a new version of class `Super` is produced:

```
class Super extends Hyper {
    void hello() { System.out.println("hello from Super"); }
}
```

If `Super` and `Hyper` are recompiled but not `Test`, then running the new binaries with the existing binary of `Test` produces the output:

```
hello from Super
```

as you might expect. (A flaw in some early implementations caused them to print:

```
hello from Hyper
```

incorrectly.)

13.4.6 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. So, for example, if the package `points` defines the class `Point`:

```
package points;

public class Point {
    public int x, y;
    protected void print() {
```

```
        System.out.println("(" + x + "," + y + ")");
    }
}
```

used by the `Test` program:

```
class Test extends points.Point {
    protected void print() { System.out.println("Test"); }

    public static void main(String[] args) {
        Test t = new Test();
        t.print();
    }
}
```

then these classes compile and `Test` executes to produce the output:

```
Test
```

If the method `print` in class `Point` is changed to be `public`, and then only the `Point` class is recompiled, and then executed with the previously existing binary for `Test` then no linkage error occurs, even though it is improper, at compile time, for a `public` method to be overridden by a `protected` method (as shown by the fact that the class `Test` could not be recompiled using this new `Point` class unless print were changed to be `public`.)

Allowing superclasses to change `protected` methods to be `public` without breaking binaries of preexisting subclasses helps make binaries less fragile. The alternative, where such a change would cause a linkage error, would create additional binary incompatibilities.

13.4.7 Field Declarations

Widely distributed programs should not expose any fields to their clients. Adding a field to a class may break compatibility with pre-existing binaries that are not recompiled. Apart from the binary compatibility issues discussed below, this is generally good software engineering practice.

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. Thus compiling and executing the code:

```
class Hyper { String h = "hyper"; }

class Super extends Hyper { String s = "super"; }

class Test {
   public static void main(String[] args) {
      System.out.println(new Super().h);
   }
}
```

produces the output:

```
hyper
```

Changing `Super` to be defined as:

```
class Super extends Hyper {
   String s = "super";
   int h = 0;
}
```

recompiling `Hyper` and `Super`, and executing the resulting new binaries with the old binary of `Test` produces the output:

```
hyper
```

The field `h` of `Hyper` is output by the original binary of `main` . While this may seem surprising at first, it serves to reduce the number of incompatibilities that occur at run time. (In an ideal world, all source files that needed recompilation would be recompiled whenever any one of them changed, eliminating such surprises. But such a mass recompilation is often impractical or impossible, especially in the Internet. And, as was previously noted, such recompilation would sometimes require further changes to the source code.)

As an example, if the program:

```
class Hyper { String h = "Hyper"; }

class Super extends Hyper { }

class Test extends Super {
   public static void main(String[] args) {
```

```
        String s = new Test().h;
        System.out.println(s);
    }
}
```

is compiled and executed, it produces the output:

```
Hyper
```

Suppose that a new version of class `Super` is then compiled:

```
class Super extends Hyper { char h = 'h'; }
```

If the resulting binary is used with the existing binaries for `Hyper` and `Test`, then the output is still:

```
Hyper
```

even though compiling the source for these binaries:

```
class Hyper { String h = "Hyper"; }

class Super extends Hyper { char h = 'h'; }

class Test extends Super {
    public static void main(String[] args) {
        String s = new Test().h;
        System.out.println(s);
    }
}
```

would result in a compile-time error, because the `h` in the source code for `main` would now be construed as referring to the `char` field declared in `Super`, and a `char` value can't be assigned to a `String`.

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.

13.4.8 `final` Fields and Constants

If a field that was not `final` is changed to be `final`, then it can break compatibility with pre-existing binaries that attempt to assign new values to the field. For example, if the program:

```
class Super { static char s; }

class Test extends Super {
    public static void main(String[] args) {
        s = 'a';
```

```
        System.out.println(s);
    }
}
```

is compiled and executed, it produces the output:

```
a
```

Suppose that a new version of class `Super` is produced:

```
class Super { final static char s = 'b'; }
```

If `Super` is recompiled but not `Test`, then running the new binary with the existing binary of `Test` results in a `IllegalAccessError`.

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 compile-time constant, 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 constant unless they are recompiled. If the example:

```
class Flags { final static boolean debug = true; }

class Test {
    public static void main(String[] args) {
        if (Flags.debug)
            System.out.println("debug is true");
    }
}
```

is compiled and executed, it produces the output:

```
debug is true
```

Suppose that a new version of class `Flags` is produced:

```
class Flags { final static boolean debug = false; }
```

If `Flags` is recompiled but not `Test`, then running the new binary with the existing binary of `Test` produces the output:

```
debug is true
```

because the value of `debug` was a compile-time primitive constant, and could have been used in compiling `Test` without making a reference to the class `Flags`.

This result is a side-effect of the decision to support conditional compilation, as discussed at the end of §14.20.

This behavior would not change if Flags were changed to be an interface, as in the modified example:

```
interface Flags { boolean debug = true; }

class Test {
   public static void main(String[] args) {
      if (Flags.debug)
         System.out.println("debug is true");
   }
}
```

(One reason for requiring inlining of primitive constants is that switch statements require constants on each case, and no two such constant values may be the same. The compiler checks for duplicate constant values in a switch statement at compile time; the class file format does not do symbolic linkage of case values.)

The best way to avoid problems with "inconstant constants" in widely-distributed code is to declare as primitive constants only values which truly are unlikely ever to change. Many primitive constants in interfaces are small integer values replacing enumerated types, which the language does not support; these small values can be chosen arbitrarily, and should not need to be changed. 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();
}
```

insuring 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 (§4.5.5). This means that all such fields appear to be initialized first during class initialization (§8.3.2.1, §9.3.1, §12.4.2).

13.4.9 `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 time error, specifically an `IncompatibleClassChangeError`, will result if the field is used by a preexisting binary which expected a field of the other kind. Such changes are not recommended in code that has been widely distributed.

13.4.10 `transient` Fields

Adding or deleting a `transient` modifier of a field does not break compatibility with pre-existing binaries.

13.4.11 Method and Constructor Declarations

Adding a method or constructor declaration to a class will not break compatibility with any pre-existing binaries, 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.

If the source code for a class contains no declared constructors, the Java compiler automatically supplies a constructor with no parameters. Adding one or more constructor declarations to the source code of such a class will prevent this default constructor from being supplied automatically, effectively deleting a constructor, unless one of the new constructors also has no parameters, thus replacing the default constructor. The automatically supplied constructor with no parameters is given the same access modifier as the class of its declaration, so any replacement should have as much or more access if compatibility with pre-existing binaries is to be preserved.

13.4.12 Method and Constructor 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, 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 (see §13.4.14).

13.4.13 Method Result Type

Changing the result type of a method, 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 §13.4.14).

13.4.14 `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`. If the example program:

```
class Super { void out() { System.out.println("Out"); } }

class Test extends Super {
   public static void main(String[] args) {
      Test t = new Test();
      System.out.println("Way ");
      t.out();
   }
}
```

is compiled and executed, it produces the output:

```
Way
Out
```

Suppose that a new version of class `Super` is produced:

```
abstract class Super {
   abstract void out();
}
```

If `Super` is recompiled but not `Test`, then running the new binary with the existing binary of `Test` results in a `AbstractMethodError`, because class `Test` has no implementation of the method `out`, and is therefore is (or should be) abstract.

13.4.15 `final` Methods

Changing an instance method that is not `final` to be `final` may break compatibility with existing binaries that depend on the ability to override the method. If the test program:

```
class Super { void out() { System.out.println("out"); } }

class Test extends Super {

   public static void main(String[] args) {
      Test t = new Test();
      t.out();
   }

   void out() { super.out(); }

}
```

is compiled and executed, it produces the output:

```
out
```

Suppose that a new version of class `Super` is produced:

```
class Super { final void out() { System.out.println("!"); } }
```

If Super is recompiled but not Test, then running the new binary with the existing binary of Test results in a VerifyError because the class Test improperly tries to override the instance method out.

Changing a class (static) method that is not final to be final does not break compatibility with existing binaries, because the method could not have been overridden.

Removing the final modifier from a method does not break compatibility with pre-existing binaries.

13.4.16 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 preexisting 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.

13.4.17 static Methods

If a method that is not declared private was 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.

13.4.18 synchronized Methods

Adding or deleting a synchronized modifier of a method does not break compatibility with existing binaries.

13.4.19 Method and Constructor Throws

Changes to the throws clause of methods or constructors do not break compatibility with existing binaries; these clauses are checked only at compile time.

13.4.20 Method and Constructor Body

Changes to the body of a method or constructor do not break compatibility with pre-existing binaries.

- We note that a compiler cannot expand a method inline at compile time.

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.

In general we suggest that implementations use late-bound (run-time) code generation and optimization.

13.4.21 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 (§15.12.2.2), no such error occurs when a program is executed, because no overload resolution is done at execution time.

If the example program:

```
class Super {
   static void out(float f) { System.out.println("float"); }
}

class Test {
   public static void main(String[] args) {
      Super.out(2);
   }
}
```

is compiled and executed, it produces the output:

```
float
```

Suppose that a new version of class `Super` is produced:

```
class Super {
   static void out(float f) { System.out.println("float"); }
```

```
    static void out(int i) { System.out.println("int"); }
}
```

If `Super` is recompiled but not `Test`, then running the new binary with the existing binary of `Test` still produces the output:

```
float
```

However, if `Test` is then recompiled, using this new `Super`, the output is then:

```
int
```

as might have been naively expected in the previous case.

13.4.22 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.

13.4.23 . Static Initializers

Adding, deleting, or changing a static initializer (§8.7) of a class does not impact pre-existing binaries.

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 obscure 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. If the example program:

```
interface I { void hello(); }

class Test implements I {

   public static void main(String[] args) {
      I anI = new Test();
      anI.hello();
   }

   public void hello() { System.out.println("hello"); }

}
```

is compiled and executed, it produces the output:

```
hello
```

Suppose that a new version of interface `I` is compiled:

```
interface I { }
```

If `I` is recompiled but not `Test`, then running the new binary with the existing binary for `Test` will result in a `NoSuchMethodError`. (In some early implementations this program still executed; the fact that the method `hello` no longer exists in interface `I` was not correctly detected.)

13.5.4 Field Declarations

The considerations for changing field declarations in interfaces are the same as those for `static final` fields in classes, as described in §13.4.7 and §13.4.8.

13.5.5 Abstract Method Declarations

The considerations for changing abstract method declarations in interfaces are the same as those for `abstract` methods in classes, as described in §13.4.15, §13.4.16, §13.4.22, and §13.4.24.

Lo! keen-eyed, towering Science! . . .
Yet again, lo! the Soul—above all science . . .
For it, the partial to the permanent flowing,
For it, the Real to the Ideal tends.
For it, the mystic evolution . . .
—Walt Whitman, *Song of the Universal* (1874)

CHAPTER 14

Blocks and Statements

He was not merely a chip of the old block, but the old block itself.
—Edmund Burke, *On Pitt's First Speech*

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 (§15) 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), because they can appear in certain places where other kinds of statements are not allowed, and because one other kind of statement, a local variable declaration statement (§14.4), must be immediately contained within a block.

Next a grammatical maneuver that sidesteps the familiar "dangling `else`" problem (§14.5) is explained.

Statements that will be familiar to C and C++ programmers are the empty (§14.6), labeled (§14.7), expression (§14.8), `if` (§14.9), `switch` (§14.10), `while` (§14.11), `do` (§14.12), `for` (§14.13), `break` (§14.14), `continue` (§14.15), and `return` (§14.16) statements.

Unlike C and C++, the Java programming language has no `goto` statement. However, the `break` and `continue` statements are allowed to mention statement labels.

The Java programming language statements that are not in the C language are the `throw` (§14.17), `synchronized` (§14.18), and `try` (§14.19) statements.

The last section (§14.20) of this chapter addresses the requirement that every statement be *reachable* in a certain technical sense.

14.1 Normal and Abrupt Completion of Statements

Poirot's abrupt departure had intrigued us all greatly.
—Agatha Christie, *The Mysterious Affair at Styles* (1920), Chapter 12

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` (§14.14), `continue` (§14.15), and `return` (§14.16) 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` (§14.17) 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 (§14.17) or a run-time exception or error (§11, §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

He wears his faith but as the fashion of his hat;
it ever changes with the next block.
—William Shakespeare, *Much Ado about Nothing* (1623), Act I, scene i

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 (§8) that is not a member of any class and that has a name. All local classes are inner classes (§8.1.2). 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 scope of a local class declared in a block is the rest of the immediately enclosing block, including its own class declaration.

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. However, a local class name may be hidden anywhere inside a class declaration nested within the local class name's scope. Local classes do not have a fully qualified name.

It is a compile time error if a local class declaration contains any one of the following access modifiers: `public`, `protected`, `private` or `static`.

Here is an example that illustrates several aspects of the rules given above:

```
class Global {
   class Cyclic {}
   void foo() {
      new Cyclic(); // create a Global.Cyclic
      class Cyclic extends Cyclic{}; // circular definition
      {
         class Local{};
         {
            class Local{}; // compile-time error
         }
         class Local{};
         class AnotherLocal {
            void bar() {
               class Local {}; // ok
            }
         }
      }
      class Local{}; // ok, not in scope of prior Local
}
```

The first statement of method `foo` creates an instance of the member class `Global.Cyclic` rather than an instance of the local class `Cyclic`, because the local class name is not yet in scope.

The fact that the scope of a local class encompasses its own declaration (not only its body) means that the definition of the local class `Cyclic` is indeed cyclic because it extends itself rather than `Global.Cyclic`. Consequently, the declaration of the local class `Cyclic` will be rejected at compile-time.

Since local class names cannot be redeclared within the same method (or constructor or initializer, as the case may be), the second and third declarations of `Local` result in compile-time errors. However, `Local` can be redeclared in the context of another, more deeply nested, class such as `AnotherLocal`.

The fourth and last declaration of `Local` is legal, since it occurs outside the scope of any prior declaration of `Local`.

14.4 Local Variable Declaration Statements

A *local variable declaration statement* declares one or more local variable names.

LocalVariableDeclarationStatement:
LocalVariableDeclaration `;`

LocalVariableDeclaration:
`final`^{opt} *Type* *VariableDeclarators*

The following are repeated from §8.3 to make the presentation here clearer:

VariableDeclarators:
VariableDeclarator
VariableDeclarators `,` *VariableDeclarator*

VariableDeclarator:
VariableDeclaratorId
VariableDeclaratorId `=` *VariableInitializer*

VariableDeclaratorId:
Identifier
VariableDeclaratorId `[ ]`

VariableInitializer:
Expression
ArrayInitializer

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 (§14.13). 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.

The type of the variable is denoted by the *Type* that appears at the start of the local variable declaration, followed by any bracket pairs that follow the *Identifier* in the declarator. Thus, the local variable declaration:

```
int a, b[], c[][];
```

is equivalent to the series of declarations:

```
int a;
int[] b;
int[][] c;
```

Brackets are allowed in declarators as a nod to the tradition of C and C++. The general rule, however, also means that the local variable declaration:

```
float[][] f[][], g[][][], h[];          // Yechh!
```

is equivalent to the series of declarations:

```
float[][][][] f;
float[][][][][] g;
float[][][] h;
```

We do not recommend such "mixed notation" for array declarations.

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 Scope of Local Variable Declarations

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 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. However, a local variable of a method or initializer block may be hidden anywhere inside a class declaration nested within the scope of the local variable.

A local variable cannot be referred to using a qualified name (§6.6), only a simple name.

The example:

```
class Test {
   static int x;

   public static void main(String[] args) {
      int x = x;
   }
}
```

causes a compile-time error because the initialization of `x` is within the scope of the declaration of `x` as a local variable, and the local `x` does not yet have a value and cannot be used.

The following program does compile:

```
class Test {
   static int x;
   public static void main(String[] args) {
      int x = (x=2)*2;
      System.out.println(x);
   }
}
```

because the local variable `x` is definitely assigned (§16) before it is used. It prints:

```
4
```

Here is another example:

```
class Test {
   public static void main(String[] args) {
      System.out.print("2+1=");
      int two = 2, three = two + 1;
      System.out.println(three);
   }
}
```

which compiles correctly and produces the output:

```
2+1=3
```

The initializer for `three` can correctly refer to the variable `two` declared in an earlier declarator, and the method invocation in the next line can correctly refer to the variable `three` declared earlier in the block.

The scope of a local variable declared in a `for` statement is the rest of the `for` statement, including its own initializer.

If a declaration of an identifier as a local variable of the same method, constructor, or initializer block appears within the scope of a parameter or local variable of the same name, a compile-time error occurs. Thus the following example does not compile:

```
class Test {
   public static void main(String[] args) {
      int i;
      for (int i = 0; i < 10; i++)
         System.out.println(i);
   }
}
```

This restriction helps to detect some otherwise very obscure bugs. A similar restriction on hiding of members by local variables was judged impractical, because the addition of a member in a superclass could cause subclasses to have to rename local variables. Related considerations make restrictions on hiding of local variables by members of nested classes, or on hiding of local variables by local variables declared within nested classes unattractive as well. Hence, the following example compiles without error:

```
class Test {
   public static void main(String[] args) {
      int i;
      class Local {
         {
            for (int i = 0; i < 10; i++)
            System.out.println(i);
         }
      }
      new Local();
   }
}
```

On the other hand, local variables with the same name may be declared in two separate blocks or `for` statements neither of which contains the other. Thus:

```
class Test {
   public static void main(String[] args) {
      for (int i = 0; i < 10; i++)
         System.out.print(i + " ");
      for (int i = 10; i > 0; i--)
         System.out.print(i + " ");
      System.out.println();
   }
}
```

compiles without error and, when executed, produces the output:

```
0 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1
```

14.4.3 Hiding of Names by Local Variables

If a name declared as a local variable is already declared as a field name, then that outer declaration is hidden throughout the scope of the local variable. Similarly, if a name is already declared as a variable or parameter name, then that outer declaration is hidden throughout the scope of the local variable (provided that the hiding does not cause a compile-time error under the rules of §14.4.2). The hidden name can sometimes be accessed using an appropriately qualified name. For example, the keyword `this` can be used to access a hidden field `x`, using the form `this.x`. Indeed, this idiom typically appears in constructors (§8.8):

```
class Pair {
   Object first, second;
   public Pair(Object first, Object second) {
      this.first = first;
      this.second = second;
   }
}
```

In this example, the constructor takes parameters having the same names as the fields to be initialized. This is simpler than having to invent different names for the parameters and is not too confusing in this stylized context. In general, however, it is considered poor style to have local variables with the same names as fields.

14.4.4 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 §16, 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 the 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 language. Most correspond to statements in the C and C++ languages, but some are unique to Java.

As in C and C++, the Java `if` statement 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 language, like C and C++ and many 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

StatementNoShortIf:
StatementWithoutTrailingSubstatement
LabeledStatementNoShortIf
IfThenElseStatementNoShortIf
WhileStatementNoShortIf
ForStatementNoShortIf

StatementWithoutTrailingSubstatement:
Block
EmptyStatement
ExpressionStatement
SwitchStatement
DoStatement
BreakStatement

ContinueStatement
ReturnStatement
SynchronizedStatement
ThrowStatement
TryStatement

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

I did never know so full a voice issue from so empty a heart:
but the saying is true 'The empty vessel makes the greatest sound.'
—William Shakespeare, *Henry V* (1623), Act IV, scene iv

An *empty statement* does nothing.

EmptyStatement:
`;`

Execution of an empty statement always completes normally.

14.7 Labeled Statements

Inside of five minutes I was mounted, and perfectly satisfied with my outfit. I had no time to label him "This is a horse," and so if the public took him for a sheep I cannot help it.
—Mark Twain, *Roughing It* (1871)

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` (§14.14) or `continue` (§14.15) statements appearing anywhere within the labeled statement.

The scope (§6.3) of a label declared by a a labeled statement is the statement immediately enclosed by 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* hides 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 hide 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 (§14.19) does not hide 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 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 a compile-time error occurs.

14.9.1 The `if–then` Statement

I took an early opportunity of testing that statement . . .
—Agatha Christie, *The Mysterious Affair at Styles* (1920), Chapter 12

An `if–then` statement is executed by first evaluating the *Expression*. If evaluation of the *Expression* 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

Did you ever have to finally decide—
To say yes to one, and let the other one ride?
—John Sebastian, *Did You Ever Have to Make Up Your Min*
d?

An `if–then–else` statement is executed by first evaluating the *Expression*. If evaluation of the *Expression* 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 `switch` Statement

Fetch me a dozen crab-tree staves, and strong ones: these are but switches . . .
—William Shakespeare, *Henry VIII* (1623), Act V, scene iv

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* `:`
`default :`

The type of the *Expression* must be `char`, `byte`, `short`, or `int`, 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 `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) 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 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.

In C and C++ the body of a `switch` statement can be a statement and statements with `case` labels do not have to be immediately contained by that statement. Consider the simple loop:

```
for (i = 0; i < n; ++i) foo();
```

where n is known to be positive. A trick known as *Duff's device* can be used in C or C++ to unroll the loop, but this is not valid code in the Java programming language:

```
int q = (n+7)/8;
switch (n%8) {
case 0:   do {   foo();        // Great C hack, Tom,
case 7:          foo();        // but it's not valid here.
case 6:          foo();
case 5:          foo();
case 4:          foo();
case 3:          foo();
case 2:          foo();
case 1:          foo();
         } while (--q >= 0);
}
```

Fortunately, this trick does not seem to be widely known or used. Moreover, it is less needed nowadays; this sort of code transformation is properly in the province of state-of-the-art optimizing compilers.

When the switch statement is executed, first the *Expression* is evaluated. If evaluation of the *Expression* 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. Then there is a choice:

- 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". For example, the program:

```
class Toomany {

   static void howMany(int k) {
      switch (k) {
      case 1:   System.out.print("one ");
      case 2:   System.out.print("too ");
      case 3:   System.out.println("many");
      }
   }

   public static void main(String[] args) {
      howMany(3);
      howMany(2);
      howMany(1);
   }

}
```

contains a switch block in which the code for each case falls through into the code for the next case. As a result, the program prints:

```
many
too many
one too many
```

If code is not to fall through case to case in this manner, then `break` statements should be used, as in this example:

```
class Twomany {

   static void howMany(int k) {
      switch (k) {
      case 1:   System.out.println("one");
                break;           // exit the switch
      case 2:   System.out.println("two");
                break;           // exit the switch
      case 3:   System.out.println("many");
                break;           // not needed, but good style
      }
   }
```

```
    public static void main(String[] args) {
        howMany(1);
        howMany(2);
        howMany(3);
    }
}
```

This program prints:

```
one
two
many
```

14.11 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 a compile-time error occurs.

A `while` statement is executed by first evaluating the *Expression*. If evaluation of the *Expression* 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 §14.11.1 below.
- If the value of the *Expression* is `false`, no further action is taken and the `while` statement completes normally.

If the value of the *Expression* is `false` the first time it is evaluated, then the *Statement* is not executed.

14.11.1 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. Note that the case of abrupt completion because of a `break` with a label is handled by the general rule for labeled statements (§14.7).

14.12 The do Statement

"She would not see it," he said at last, curtly,
feeling at first that this statement must do without explanation.
—George Eliot, *Middlemarch* (1871), Chapter 76

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 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 evaluation of the *Expression* 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 §14.12.1 below.

Executing a do statement always executes the contained *Statement* at least once.

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, 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 (§14.7).

14.12.2 Example of do statement

The following code is one possible implementation of the `toHexString` method of class `Integer`:

```
public static String toHexString(int i) {
   StringBuffer buf = new StringBuffer(8);
   do {
      buf.append(Character.forDigit(i & 0xF, 16));
```

```
        i >>>= 4;
    } while (i != 0);
    return buf.reverse().toString();
}
```

Because at least one digit must be generated, the do statement is an appropriate control structure.

14.13 The for Statement

The 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.

ForStatement:
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 a compile-time error occurs.

14.13.1 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 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. In this case, the scope of a declared local variable is its own initializer and any further declarators in the *ForInit* part, plus the *Expression*, *ForUpdate*, and contained *Statement* of the `for` statement. The scope of a local variable declared in the *ForInit* part of a `for` statement (§14.13) 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*
- 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.

14.13.2 Iteration of `for` statement

Next, a `for` iteration step is performed, as follows:

- If the *Expression* is present, it is evaluated, and if evaluation of the *Expression* 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:
  - If the *Expression* is not present, or it is present and the value resulting from its evaluation 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:
      - 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.

        - Second, another `for` iteration step is performed.
      - If execution of the *Statement* completes abruptly, see §14.13.3 below.
    - If the *Expression* is present and the value resulting from its evaluation is `false`, no further action is taken and the `for` statement completes normally.

If the 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.

14.13.3 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:
  - 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.
  - 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:
    - 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.
    - 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. Note that 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 break Statement

A break statement transfers control out of an enclosing statement.

BreakStatement:
`break` *Identifier*^{opt} `;`

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 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 `while`, `do`, `for`, or `switch` statement. A `break` statement must refer to a label within the immediately enclosing method or initializer block. There are no non-local jumps.

To be precise, a `break` statement with label *Identifier* always completes abruptly, the reason being a `break` with label *Identifier*. If no labeled statement with *Identifier* as its label encloses the `break` statement, a compile-time error occurs.

It can be seen, then, that a `break` statement always completes abruptly.

The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any `try` statements (§14.19) within the break target whose `try` blocks contain the `break` statement, then any `finally` clauses of those `try` statements are executed, in order, innermost to outermost, before control is transferred to the break target. Abrupt completion of a `finally` clause can disrupt the transfer of control initiated by a `break` statement.

In the following example, a mathematical graph is represented by an array of arrays. A graph consists of a set of nodes and a set of edges; each edge is an arrow that points from some node to some other node, or from a node to itself. In this example it is assumed that there are no redundant edges; that is, for any two nodes *P* and *Q*, where *Q* may be the same as *P*, there is at most one edge from *P* to *Q*. Nodes are represented by integers, and there is an edge from node *i* to node `edges[i][j]` for every *i* and *j* for which the array reference `edges[i][j]` does not throw an `IndexOutOfBoundsException`.

The task of the method `loseEdges`, given integers *i* and *j*, is to construct a new graph by copying a given graph but omitting the edge from node *i* to node *j*, if any, and the edge from node *j* to node *i*, if any:

```
class Graph {
   int edges[][];

   public Graph(int[][] edges) { this.edges = edges; }

   public Graph loseEdges(int i, int j) {
      int n = edges.length;
      int[][] newedges = new int[n][];
      for (int k = 0; k < n; ++k) {

         edgelist: {
            int z;

            search: {
               if (k == i) {
                  for (z = 0; z < edges[k].length; ++z)
                     if (edges[k][z] == j)
                        break search;
               } else if (k == j) {
                  for (z = 0; z < edges[k].length; ++z)
                     if (edges[k][z] == i)
                        break search;
               }
               // No edge to be deleted; share this list.
               newedges[k] = edges[k];
               break edgelist;
            }//search

            // Copy the list, omitting the edge at position z.
            int m = edges[k].length - 1;
            int ne[] = new int[m];
            System.arraycopy(edges[k], 0, ne, 0, z);
            System.arraycopy(edges[k], z+1, ne, z, m-z);
            newedges[k] = ne;
         }//edgelist

      }
      return new Graph(newedges);
   }

}
```

Note the use of two statement labels, `edgelist` and `search`, and the use of `break` statements. This allows the code that copies a list, omitting one edge, to be shared between two separate tests, the test for an edge from node *i* to node *j*, and the test for an edge from node *j* to node *i*.

14.15 The continue Statement

"Your experience has been a most entertaining one," remarked Holmes as his client paused and refreshed his memory with a huge pinch of snuff. "Pray continue your very interesting statement."

—Sir Arthur Conan Doyle, *The Red-headed League* (1891)

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` *Identifier*^{opt} `;`

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.

More precisely, a `continue` statement with label *Identifier* always completes abruptly, the reason being a `continue` with label *Identifier*. If no labeled statement with *Identifier* as its label contains the `continue` statement, a compile-time error occurs.

It can be seen, then, that a `continue` statement always completes abruptly.

See the descriptions of the `while` statement (§14.11), `do` statement (§14.12), and `for` statement (§14.13) for a discussion of the handling of abrupt termination because of `continue`.

The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any `try` statements (§14.19) within the continue target whose `try` blocks contain the `continue` statement, then any `finally` clauses of those `try` statements are executed, in order, innermost to outermost, before control is transferred to the continue target. Abrupt completion of a

finally clause can disrupt the transfer of control initiated by a continue statement.

In the Graph example in the preceding section, one of the break statements is used to finish execution of the entire body of the outermost for loop. This break can be replaced by a continue if the for loop itself is labeled:

```
class Graph {
   ...
   public Graph loseEdges(int i, int j) {
      int n = edges.length;
      int[][] newedges = new int[n][];

      edgelists: for (int k = 0; k < n; ++k) {
         int z;

         search: {
            if (k == i) {
               ...
            } else if (k == j) {
               ...
            }
            newedges[k] = edges[k];
            continue edgelists;
         }//search
         ...
      }//edgelists

      return new Graph(newedges);
   }
}
```

Which to use, if either, is largely a matter of programming style.

14.16 The return Statement

"Know you, O judges and people of Helium," he said, "that John Carter, one time Prince of Helium, has returned by his own statement from the Valley Dor . . ."

—Edgar Rice Burroughs, *The Gods of Mars* (1913)

A return statement returns control to the invoker of a method (§8.4, §15.12) or constructor (§8.8, §15.9).

ReturnStatement:
return *Expression*^{opt} ;

A return statement with no *Expression* must be contained in the body of a method that is declared, using the keyword void, not to return any value (§8.4), or in the body of a constructor (§8.8). A compile-time error occurs if a return statement appears within an instance initializer or a static initializer (§8.7). 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.

The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any try statements (§14.19) within the method or constructor whose try blocks contain the return statement, then any finally clauses of those try statements will be executed, in order, innermost to outermost, before control is transferred to the invoker of the method or constructor. Abrupt completion of a finally clause can disrupt the transfer of control initiated by a return statement.

14.17 The throw Statement

A throw statement causes an exception (§11) 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 (§14.19) is found that catches the thrown value. If no such try statement is found, then execution of the thread

(§17) 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 a variable or value of a reference type which is assignable (§5.2) to the type Throwable, or a compile-time error occurs. Moreover, at least one of the following three conditions must be true, or a compile-time error occurs:

- The exception is not a checked exception (§11.2)—specifically, one of the following situations is true:
  - The type of the *Expression* is the class RuntimeException or a subclass of RuntimeException.
  - The type of the *Expression* is the class Error or a subclass of Error.
- The throw statement is contained in the try block of a try statement (§14.19) and the type of the *Expression* is assignable (§5.2) to the type of the parameter of at least one catch clause of the try statement. (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 (§8.4.4, §8.8.4) of the declaration.

A throw statement first evaluates the *Expression*. If the 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 value *V*, then the throw statement completes abruptly, the reason being a throw with value *V*.

It can be seen, then, that a throw statement always completes abruptly.

If there are any enclosing try statements (§14.19) 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 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 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.

By convention, user-declared throwable types should usually be declared to be subclasses of class `Exception`, which is a subclass of class `Throwable` (§11.5,).

14.18 The `synchronized` Statement

A `synchronized` statement acquires a mutual-exclusion lock (§17.13) 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*.

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 then completes abruptly for the same reason.

Acquiring the lock associated with an object does not of itself prevent other threads from accessing fields of the object or invoking unsynchronized methods

on the object. Other threads can also use synchronized methods or the synchronized statement in a conventional manner to achieve mutual exclusion.

The locks acquired by synchronized statements are the same as the locks that are acquired implicitly by synchronized methods; see §8.4.3.6. A single thread may hold a lock more than once. The example:

```
class Test {
   public static void main(String[] args) {
      Test t = new Test();
      synchronized(t) {
         synchronized(t) {
            System.out.println("made it!");
         }
      }
   }
}
```

prints:

```
made it!
```

This example would deadlock if a single thread were not permitted to lock a lock more than once.

14.19 The try statement

These are the times that try men's souls.
—Thomas Paine, *The American Crisis* (1780)

. . . and they all fell to playing the game of catch as catch can, till the gunpowder ran out at the heels of their boots.
—Samuel Foote

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* $Catches^{opt}$ *Finally*

Catches:
CatchClause
Catches CatchClause

CatchClause:
`catch (` *FormalParameter* `)` *Block*

Finally:
`finally` *Block*

The following is repeated from §8.4.1 to make the presentation here clearer:

```
FormalParameter:
   finalopt Type VariableDeclaratorId
```

The following is repeated from §8.3 to make the presentation here clearer:

```
VariableDeclaratorId:
   Identifier
   VariableDeclaratorId [ ]
```

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 of `Throwable`, or a compile-time error occurs. The scope of the parameter variable is the *Block* of the `catch` clause.

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.

The scope of a parameter of an exception handler that is declared in a `catch` clause of a `try` statement (§14.19) is the entire block associated with the `catch`. 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 . However, an exception parameter may be hidden anywhere inside a class declaration nested within the *Block* of the `catch` clause.

It is a compile time error if an exception parameter that is declared `final` is assigned to within the body of the catch clause.

Exception parameters cannot be referred to using qualified names (§6.6), only by simple names.

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.

14.19.1 Execution of `try-catch`

Our supreme task is the resumption of our onward, normal way.
—Warren G. Harding, Inaugural Address (1921)

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.

In the example:

```
class BlewIt extends Exception {

   BlewIt() { }

   BlewIt(String s) { super(s); }
```

```
}

class Test {

   static void blowUp() throws BlewIt { throw new BlewIt(); }

   public static void main(String[] args) {

      try {
         blowUp();
      } catch (RuntimeException r) {
         System.out.println("RuntimeException:" + r);
      } catch (BlewIt b) {
         System.out.println("BlewIt");
      }
   }

}
```

the exception `BlewIt` is thrown by the method `blowUp`. The `try-catch` statement in the body of `main` has two `catch` clauses. The run-time type of the exception is `BlewIt` which is not assignable to a variable of type `RuntimeException`, but is assignable to a variable of type `BlewIt`, so the output of the example is:

```
BlewIt
```

14.19.2 Execution of `try-catch-finally`

After the great captains and engineers have accomplish'd their work,
After the noble inventors—after the scientists, the chemist,
the geologist, ethnologist,
Finally shall come the Poet . . .
—Walt Whitman, *Passage to India* (1870)

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).

The example:

```
class BlewIt extends Exception {

   BlewIt() { }
```

```
    BlewIt(String s) { super(s); }
}

class Test {
    static void blowUp() throws BlewIt {
        throw new NullPointerException();
    }
    public static void main(String[] args) {
        try {
            blowUp();
        } catch (BlewIt b) {
            System.out.println("BlewIt");
        } finally {
            System.out.println("Uncaught Exception");
        }
    }
}
```

produces the output:

```
Uncaught Exception
java.lang.NullPointerException
    at Test.blowUp(Test.java:7)
    at Test.main(Test.java:11)
```

The `NullPointerException` (which is a kind of `RuntimeException`) that is thrown by method `blowUp` is not caught by the `try` statement in `main`, because a `NullPointerException` is not assignable to a variable of type `BlewIt`. This causes the `finally` clause to execute, after which the thread executing `main`, which is the only thread of the test program, terminates because of an uncaught exception , which typically results in printing the exception name and a simple backtrace.

14.20 Unreachable Statements

That looks like a path.
Is that the way to reach the top from here?
—Robert Frost, *The Mountain* (1915)

It is a compile-time error if a statement cannot be executed because it is *unreachable*. Every Java compiler must carry out the conservative flow analysis specified here to make sure all statements are reachable.

This section is devoted to a precise explanation of the word "reachable." The idea is that there must be some possible execution path from the beginning of the constructor, method, instance initializer or static initializer that contains the statement to the statement itself. The analysis takes into account the structure of statements. Except for the special treatment of `while`, `do`, and `for` statements whose condition expression has the constant value `true`, the values of expressions are not taken into account in the flow analysis. For example, a Java compiler will accept the code:

```
{
   int n = 5;
   while (n > 7) k = 2;
}
```

even though the value of `n` is known at compile time and in principle it can be known at compile time that the assignment to `k` can never be executed. A Java compiler must operate according to the rules laid out in this section.

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.

To shorten the description of the rules, the customary abbreviation "iff" is used to mean "if and only if."

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 nonempty block that is not a switch block can complete normally iff the last statement in it can complete normally. The first statement in a nonempty block that is not a switch block is reachable iff the block is reachable. Every other statement *S* in a nonempty 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.
- A `switch` statement can complete normally iff at least one of the following is true:
  - The last statement in the switch block can complete normally.
  - The switch block is empty or contains only switch labels.
  - 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 is the innermost while, do, or for statement that contains that continue 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 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 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.
- 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:
  - Some expression or throw statement in the try block is reachable and can throw an exception whose type is assignable to the parameter of the catch clause *C*. (An expression is considered 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.

- If a `finally` block is present, it is reachable iff the `try` statement is reachable.

One might expect the `if` statement to be handled in the following manner, but these are not the rules that the Java programming language actually uses:

- HYPOTHETICAL: An `if–then` statement can complete normally iff at least one of the following is `true`:
  - The `if–then` statement is reachable and the condition expression is not a constant expression whose value is `true`.
  - The `then`–statement can complete normally.

  The `then`–statement is reachable iff the `if–then` statement is reachable and the condition expression is not a constant expression whose value is `false`.
- HYPOTHETICAL: 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 and the condition expression is not a constant expression whose value is `false`. The `else` statement is reachable iff the `if–then–else` statement is reachable and the condition expression is not a constant expression whose value is `true`.

This approach would be consistent with the treatment of other control structures. However, in order to allow the if statement to be used conveniently for "conditional compilation" purposes, the actual rules are as follows:

- ACTUAL: An `if–then` statement can complete normally iff it is reachable. The `then`–statement is reachable iff the `if–then` statement is reachable.
- ACTUAL: 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.

As an example, the following statement results in a compile-time error:

```
while (false) { x=3; }
```

because the statement `x=3;` is not reachable; but the superficially similar case:

```
if (false) { x=3; }
```

does not result in a compile-time error. An optimizing compiler may realize that the statement `x=3;` will never be executed and may choose to omit the code for that statement from the generated `class` file, but the statement `x=3;` is not regarded as "unreachable" in the technical sense specified here.

The rationale for this differing treatment is to allow programmers to define "flag variables" such as:

```
static final boolean DEBUG = false;
```

and then write code such as:

```
if (DEBUG) { x=3; }
```

The idea is that it should be possible to change the value of `DEBUG` from `false` to `true` or from `true` to `false` and then compile the code correctly with no other changes to the program text.

This ability to "conditionally compile" has a significant impact on, and relationship to, binary compatibility (§13). If a set of classes that use such a "flag" variable are compiled and conditional code is omitted, it does not suffice later to distribute just a new version of the class or interface that contains the definition of the flag. A change to the value of a flag is, therefore, not binary compatible with preexisting binaries (§13.4.8). (There are other reasons for such incompatibility as well, such as the use of constants in `case` labels in `switch` statements; see §13.4.8.)

One ought not to be thrown into confusion
By a plain statement of relationship . . .
—Robert Frost, *The Generations of Men* (1914)

CHAPTER 15

Expressions

When you can measure what you are speaking about,
and express it in numbers, you know something about it;
but when you cannot measure it, when you cannot express it in numbers,
your knowledge of it is of a meager and unsatisfactory kind:
it may be the beginning of knowledge, but you have scarcely,
in your thoughts, advanced to the stage of science.
—William Thompson, Lord Kelvin

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 (§4.5) (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 (§15.13) 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 (§5.1.8) is applied to the result of every expression that produces a value.

Each expression occurs in the declaration of some (class or interface) type that is being declared: in a field initializer, in a static initializer, in a constructor declaration, or in the code for a method.

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 (§5.1.8) 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 always assignment compatible (§5.2) with the type of the expression, just as the value stored in a variable is always compatible with the type of the variable.

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 (§5.1.8); these rules in turn depend on whether or not the expression is *FP-strict*.

Every compile-time constant expression (§15.29) 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.

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. But 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 (§15.13). 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 (§15.21.2). 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, §15.17). 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 (§10.10, §15.14, §15.27.1). 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 (§14.19). 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.

The first two of the cases just listed ought never to result in detecting a type error. Thus, 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, §15.17); in this case a `ClassCastException` is thrown.
- 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 (§10.10, §15.14, §15.27.1); in this case an `ArrayStoreException` is thrown.
- When an exception is not caught by any `catch` handler (§11.3); in this case the thread of control that encountered the exception first invokes the method `uncaughtException` for its thread group and then terminates.

15.6 Normal and Abrupt Completion of Evaluation

No more: the end is sudden and abrupt.
—William Wordsworth, *Apology for the Foregoing Poems* (1831)

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 (§15.10), array creation expression (§15.11), or string concatenation operatior expression (§15.19.1) throws an `OutOfMemoryError` if there is insufficient memory available.
- An array creation expression throws a `NegativeArraySizeException` if the value of any dimension expression is less than zero (§15.11).
- A field access (§15.12) throws a `NullPointerException` if the value of the object reference expression is `null`.
- A method invocation expression (§15.13) that invokes an instance method throws a `NullPointerException` if the target reference is `null`.
- An array access (§15.14) throws a `NullPointerException` if the value of the array reference expression is `null`.
- An array access (§15.14) 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 (§15.17) throws a `ClassCastException` if a cast is found to be impermissible at run time.
- An integer division (§15.18.2) or integer remainder (§15.18.3) operator throws an `ArithmeticException` if the value of the right-hand operand expression is zero.
- An assignment to an array component of reference type (§15.27.1) throws an `ArrayStoreException` when the value to be assigned is not compatible with the component type of the array.

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, 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

Let all things be done decently and in order.
—I Corinthians 14:40

Java guarantees that the operands of operators appear to be evaluated in a specific *evaluation order*, namely, from left to right.

It is recommended that Java 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. For example, if the left-hand operand contains an assignment to a variable and the right-hand operand contains a reference to that same variable, then the value produced by the reference will reflect the fact that the assignment occurred first.

Thus:

```
class Test {
   public static void main(String[] args) {
      int i = 2;
      int j = (i=3) * i;
      System.out.println(j);
   }
}
```

prints:

```
9
```

It is not permitted for it to print `6` instead of `9`.

If the operator is a compound-assignment operator (§15.27.2), 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. So, for example, the test program:

```
class Test {
   public static void main(String[] args) {
      int a = 9;
      a += (a = 3);                 // first example
      System.out.println(a);
      int b = 9;
      b = b + (b = 3);              // second example
      System.out.println(b);
   }
}
```

prints:

```
12
12
```

because the two assignment statements both fetch and remember the value of the left-hand operand, which is 9, before the right-hand operand of the addition is evaluated, thereby setting the variable to 3. It is not permitted for either example to produce the result 6. Note that both of these examples have unspecified behavior in C, according to the ANSI/ISO standard.

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.

Thus, the test program:

```
class Test {
   public static void main(String[] args) {
      int j = 1;
      try {
         int i = forgetIt() / (j = 2);
      } catch (Exception e) {
         System.out.println(e);
         System.out.println("Now j = " + j);
      }
   }

   static int forgetIt() throws Exception {
      throw new Exception("I'm outta here!");
   }

}
```

prints:

```
java.lang.Exception: I'm outta here!
Now j = 1
```

because the left-hand operand `forgetIt()` of the operator `/` throws an exception before the right-hand operand and its embedded assignment of `2` to `j` occurs.

15.7.2 Evaluate Operands before Operation

Java also 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 `/` (§15.18.2) or integer remainder `%` (§15.18.3), 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.

So, for example, the program:

```
class Test {
   public static void main(String[] args) {
      int divisor = 0;
      try {
         int i = 1 / (divisor * loseBig());
      } catch (Exception e) {
         System.out.println(e);
      }
   }
   static int loseBig() throws Exception {
      throw new Exception("Shuffle off to Buffalo!");
   }
}
```

always prints:

```
java.lang.Exception: Shuffle off to Buffalo!
```

and not:

```
java.lang.ArithmeticException: / by zero
```

since no part of the division operation, including signaling of a divide-by-zero exception, may appear to occur before the invocation of `loseBig` completes, even though the implementation may be able to detect or infer that the division operation would certainly result in a divide-by-zero exception.

15.7.3 Evaluation Respects Parentheses and Precedence

That is too weighty a subject to be discussed parenthetically . . .
—John Stuart Mill, *On Liberty* (1869), Chapter IV

Java implementations must respect the order of evaluation as indicated explicitly by parentheses and implicitly by operator precedence. An implementation may not take advantage of algebraic identities such as the associative law to rewrite expressions into a more convenient computational order unless it can be proven that the replacement expression is equivalent in value and in its observable side effects, even in the presence of multiple threads of execution (using the thread execution model in §17), for all possible computational values that might be involved.

In the case of floating-point calculations, this rule applies also for infinity and not-a-number (NaN) values. For example, `!(x<y)` may not be rewritten as `x>=y`, because these expressions have different values if either `x` or `y` is NaN or both are NaN.

Specifically, floating-point calculations that appear to be mathematically associative are unlikely to be computationally associative. Such computations must not be naively reordered.

For example, it is not correct for a Java compiler to rewrite `4.0*x*0.5` as `2.0*x`; while roundoff happens not to be an issue here, there are large values of `x` for which the first expression produces infinity (because of overflow) but the second expression produces a finite result.

So, for example, the test program:

```
strictfp class Test {
   public static void main(String[] args) {
      double d = 8e+307;
      System.out.println(4.0 * d * 0.5);
      System.out.println(2.0 * d);
   }
}
```

prints:

```
Infinity
1.6e+308
```

because the first expression overflows and the second does not.

In contrast, integer addition and multiplication *are* provably associative in the Java programming language.

For example `a+b+c`, where `a`, `b`, and `c` are local variables (this simplifying assumption avoids issues involving multiple threads and `volatile` variables), will always produce the same answer whether evaluated as `(a+b)+c` or `a+(b+c)`;

if the expression `b+c` occurs nearby in the code, a smart compiler may be able to use this common subexpression.

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.

Thus:

```
class Test {

   public static void main(String[] args) {
      String s = "going, ";
      print3(s, s, s = "gone");
   }

   static void print3(String a, String b, String c) {
      System.out.println(a + b + c);
   }

}
```

always prints:

```
going, going, gone
```

because the assignment of the string `"gone"` to `s` occurs after the first two arguments to `print3` have been evaluated.

If evaluation of an argument expression completes abruptly, no part of any argument expression to its right appears to have been evaluated.

Thus, the example:

```
class Test {
   static int id;

   public static void main(String[] args) {
      try {
         test(id = 1, oops(), id = 3);
      } catch (Exception e) {
         System.out.println(e + ", id=" + id);
      }
   }
   static int oops() throws Exception {
      throw new Exception("oops");
   }
   static int test(int a, int b, int c) {
```

```
            return a + b + c;
        }
    }
```

prints:

```
java.lang.Exception: oops, id=1
```

because the assignment of `3` to `id` is not executed.

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 (§15.10.4)
- array creation expressions (§15.11.1)
- method invocation expressions (§15.13.4)
- array access expressions (§15.14.1)
- assignments involving array components (§15.27)

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 followed by a '.' and the token `class`. The type of a class literal is `Class`. It 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.

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 (§15.13), 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*.

In the example:

```
class IntVector {
   int[] v;

   boolean equals(IntVector other) {
      if (this == other)
         return true;
      if (v.length != other.v.length)
         return false;
      for (int i = 0; i < v.length; i++)
         if (v[i] != other.v[i])
            return false;
      return true;
   }
}
```

the class `IntVector` implements a method `equals`, which compares two vectors. If the `other` vector is the same vector object as the one for which the `equals` method was invoked, then the check can skip the length and value comparisons. The `equals` method implements this check by comparing the reference to the `other` object to `this`.

The keyword `this` is also used in a special explicit constructor invocation statement, which can appear at the beginning of a constructor body (§8.8.5).

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 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 (§8.1.2). 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.

Parentheses do not affect in any way the choice of value set (§4.2.3) for the value of an expression of type float or double.

15.9 Class Instance Creation Expressions

And now a new object took possession of my soul.
—Edgar Allen Poe, *A Tale of the Ragged Mountains* (1844)

A class instance creation expression is used to create new objects that are instances of classes.

ClassInstanceCreationExpression:
new *ClassOrInterfaceType* (*ArgumentList*^{opt}) *ClassBody*^{opt}

Primary.new *Identifier* (*ArgumentList*^{opt}) *ClassBody*^{opt}

ArgumentList:
Expression
ArgumentList , *Expression*

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 (§15.10.5).

- *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* (§15.10.5) 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). If *T* is the name of a class, then an anonymous direct subclass of the class named by *T* is declared. If *T* is the name of 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 an accessible (§6.6) inner class (§8.1.2) that is a member of the compile-time type of the *Primary.* 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 name a class that is

accessible (§6.6) 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 (§8.1.2) *T* that is a member of the compile-time type of the *Primary*. 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* the instance being created. If *C* is an inner class then *i* may have an immediately enclosing instance. The immediately enclosing instance of *i* (§8.1.2) is determined as follows:

- If *C* is an anonymous class, then:
  - If the class instance creation expression occurs in a static context (§8.1.2), then *i* has no immediately enclosing instance.
  - Otherwise, the immediately enclosing instance of *i* is `this`.
- If *C* is a local class (§14.3), *C* must be declared in a method declared in a lexically enclosing class *O*. 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` (§8.1.2).
- Otherwise, *C* is an inner member class (§8.5).
  - 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 *O*. 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* which is determined as follows:

- If *S* is a local class (§14.3), then *S* must be declared in a method declared in a lexically enclosing class *O*. 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).
  - 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 *O*. 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* is the object that is the value of the *Primary* expression.

15.9.3 Choosing the Constructor and its Arguments

Let *C* be the 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 the *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 (§15.13). As in method invocations, a compile-time method matching error results 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.

15.9.4 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.

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` (§15.10.6).

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 (§4.5.5).

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 compiler.

An anonymous class is never `abstract` (§8.1.1.1). An anonymous class is always an inner class (§8.1.2); it is never `static` (§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 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 (§15.13). 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 (§8.8.5.1) 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 $n-1$ st 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 (§15.13). 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 (§8.8.5.1) 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 explicit constructor invocation within an anonymous constructor must obey to the rules given in §8.8.5. An anonymous constructor never has a `throws` clause. As a consequence, if the superclass constructor invoked by the superclass constructor invocation has a `throws` clause, a compile-time error will occur.

Note that it is possible for the signature of the anonymous constructor to refer to an inaccessible type (for example, if such a type occured in the signature of the superclass constructor *cs)*. This does not, in itself, cause any errors at either compile time or run time.

15.9.6 Example: Evaluation Order and Out-of-Memory Detection

If evaluation of a class instance creation expression finds there is insufficient memory to perform the creation operation, then an `OutOfMemoryError` is thrown. This check occurs before any argument expressions are evaluated.

So, for example, the test program:

```
class List {
   int value;
   List next;
   static List head = new List(0);
   List(int n) { value = n; next = head; head = this; }
}
```

```
class Test {
   public static void main(String[] args) {
      int id = 0, oldid = 0;
      try {
         for (;;) {
            ++id;
            new List(oldid = id);
         }
      } catch (Error e) {
         System.out.println(e + ", " + (oldid==id));
      }
   }
}
```

prints:

```
java.lang.OutOfMemoryError: List, false
```

because the out-or-memory condition is detected before the argument expression `oldid = id` is evaluated.

Compare this to the treatment of array creation expressions (§15.11), for which the out-of-memory condition is detected after evaluation of the dimension expressions (§).

15.10 Array Creation Expressions

This was all as it should be, and I went out in my new array . . .
—Charles Dickens, *Great Expectations* (1861)

An array instance creation expression is used to create new arrays (§10).

ArrayCreationExpression:
`new` *PrimitiveType DimExprs Dims*^{opt}
`new` *TypeName DimExprs Dims*^{opt}

`new` *PrimitiveType Dims ArrayInitializer*
`new` *TypeName 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 *TypeName*. The *TypeName* may name any named reference type, even an `abstract` class type (§8.1.1.1) or an interface type (§9).

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.

For example, the type of the creation expression:

```
new double[3][3][]
```

is:

```
double[][][]
```

The type of each dimension expression within a *DimExpr* must be 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; this means, specifically, that the type of a dimension expression must not be `long`.

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 single-dimensional array is created of the specified length, and each component of the array is initialized to its default value (§4.5.5).

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.

For example, the declaration:

```
float[][] matrix = new float[3][3];
```

is equivalent in behavior to:

```
float[][] matrix = new float[3][];
for (int d = 0; d < matrix.length; d++)
   matrix[d] = new float[3];
```

and:

```
Age[][][][][] Aquarius = new Age[6][10][8][12][];
```

is equivalent to:

```
Age[][][][][] Aquarius = new Age[6][][][][];
for (int d1 = 0; d1 < Aquarius.length; d1++) {
   Aquarius[d1] = new Age[10][][][];
   for (int d2 = 0; d2 < Aquarius[d1].length; d2++) {
      Aquarius[d1][d2] = new Age[8][][];
      for (int d3 = 0; d3 < Aquarius[d1][d2].length; d3++) {
         Aquarius[d1][d2][d3] = new Age[12][];
      }
   }
}
```

with *d, d1*, *d2* and *d3* replaced by names that are not already locally declared. Thus, a single `new` expression actually creates one array of length 6, 6 arrays of length 10, $6 \times 10 = 60$ arrays of length 8, and $6 \times 10 \times 8 = 480$ arrays of length 12. This example leaves the fifth dimension, which would be arrays containing the actual array elements (references to `Age` objects), initialized only to null references. These arrays can be filled in later by other code, such as:

```
Age[] Hair = { new Age("quartz"), new Age("topaz") };
Aquarius[1][9][6][9] = Hair;
```

A multidimensional array need not have arrays of the same length at each level.

Thus, a triangular matrix may be created by:

```
float triang[][] = new float[100][];
for (int i = 0; i < triang.length; i++)
   triang[i] = new float[i+1];
```

15.10.2 Example: Array Creation Evaluation Order

In an array creation expression (§15.11), 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.

Thus:

```
class Test {
   public static void main(String[] args) {
      int i = 4;
      int ia[][] = new int[i][i=3];
      System.out.println(
         "[" + ia.length + "," + ia[0].length + "]");
   }
}
```

prints:

```
[4,3]
```

because the first dimension is calculated as `4` before the second dimension expression sets `i` to `3`.

If evaluation of a dimension expression completes abruptly, no part of any dimension expression to its right will appear to have been evaluated. Thus, the example:

```
class Test {
   public static void main(String[] args) {
      int[][] a = { { 00, 01 }, { 10, 11 } };
      int i = 99;
      try {
         a[val()][i = 1]++;
      } catch (Exception e) {
         System.out.println(e + ", i=" + i);
      }
   }
   static int val() throws Exception {
      throw new Exception("unimplemented");
   }
}
```

prints:

```
java.lang.Exception: unimplemented, i=99
```

because the embedded assignment that sets `i` to `1` is never executed.

Example: Array Creation and Out-of-Memory Detection

If evaluation of an array creation expression finds there is insufficient memory to perform the creation operation, then an `OutOfMemoryError` is thrown. This check occurs only after evaluation of all dimension expressions has completed normally.

So, for example, the test program:

```
class Test {
   public static void main(String[] args) {
      int len = 0, oldlen = 0;
      Object[] a = new Object[0];
      try {
         for (;;) {
            ++len;
            Object[] temp = new Object[oldlen = len];
            temp[0] = a;
            a = temp;
         }
      } catch (Error e) {
         System.out.println(e + ", " + (oldlen==len));
      }
   }
}
```

prints:

```
java.lang.OutOfMemoryError, true
```

because the out-of-memory condition is detected after the dimension expression `oldlen` = `len` is evaluated.

Compare this to class instance creation expressions (§15.10), which detect the out-of-memory condition before evaluating argument expressions (§15.10.6).

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`. (It is also possible to refer to a field of the current instance or current class by using a simple name; see §6.5.6.)

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 (§6.6), 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 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 declared type of the field. At run time, the result of the field access expression is computed as follows:
  - If the field is `static`:
    - If the field is `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`, 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`:
    - If the value of the *Primary* is `null`, then a `NullPointerException` is thrown.
    - If the field is `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`, 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.

Thus, the example:

```
class S { int x = 0; }

class T extends S { int x = 1; }

class Test {

   public static void main(String[] args) {
```

```
        T t = new T();
        System.out.println("t.x=" + t.x + when("t", t));
        S s = new S();
        System.out.println("s.x=" + s.x + when("s", s));
        s = t;
        System.out.println("s.x=" + s.x + when("s", s));
    }
    static String when(String name, Object t) {
        return " when " + name + " holds a "
            + t.getClass() + " at run time.";
    }
}
```

produces the output:

```
t.x=1 when t holds a class T at run time.
s.x=0 when s holds a class S at run time.
s.x=0 when s holds a class T at run time.
```

The last line shows that, indeed, the field that is accessed does not depend on the run-time class of the referenced object; even if `s` holds a reference to an object of class `T`, the expression `s.x` refers to the `x` field of class `S`, because the type of the expression `s` is `S`. Objects of class `T` contain two fields named `x`, one for class `T` and one for its superclass `S`.

This lack of dynamic lookup for field accesses allows programs to be run efficiently with straightforward implementations. The power of late binding and overriding is available in, but only when instance methods are used. Consider the same example using instance methods to access the fields:

```
class S { int x = 0; int z() { return x; } }

class T extends S { int x = 1; int z() { return x; } }

class Test {
    public static void main(String[] args) {
        T t = new T();
        System.out.println("t.z()=" + t.z() + when("t", t));
        S s = new S();
        System.out.println("s.z()=" + s.z() + when("s", s));
        s = t;
        System.out.println("s.z()=" + s.z() + when("s", s));
    }
    static String when(String name, Object t) {
        return " when " + name + " holds a "
```

```
            + t.getClass() + " at run time.";
    }

}
```

Now the output is:

```
t.z()=1 when t holds a class T at run time.
s.z()=0 when s holds a class S at run time.
s.z()=1 when s holds a class T at run time.
```

The last line shows that, indeed, the method that is accessed *does* depend on the run-time class of referenced object; when `s` holds a reference to an object of class `T`, the expression `s.z()` refers to the `z` method of class `T`, despite the fact that the type of the expression `s` is `S`. Method `z` of class `T` overrides method `z` of class `S`.

The following example demonstrates that a null reference may be used to access a class (`static`) variable without causing an exception:

```
class Test {

    static String mountain = "Chocorua";

    static Test favorite(){
        System.out.print("Mount ");
        return null;
    }

    public static void main(String[] args) {
        System.out.println(favorite().mountain);
    }

}
```

It compiles, executes, and prints:

```
Mount Chocorua
```

Even though the result of `favorite()` is `null`, a `NullPointerException` is *not* thrown. That "`Mount `" is printed demonstrates that the *Primary* expression is indeed fully evaluated at run time, despite the fact that only its type, not its value, is used to determine which field to access (because the field `mountain` is `static`).

15.11.2 Accessing Superclass Members using `super`

The special 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 (§15.9.3). The forms involving `super` may not be used anywhere in the class `Object`, since `Object` has no superclass; if `super` appears in class `Object`, then a compile-time error results.

Suppose that 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 `((`*S*`)this).`*`name`*; thus, it refers to the field named *name* of the current object, but with the current object viewed as an instance of the superclass. Thus it can access the field named *name* that is visible in class *S*, even if that field is hidden by a declaration of a field named *name* in class *C*.

The use of `super` is demonstrated by the following example:

```
interface I { int x = 0; }
class T1 implements I { int x = 1; }
class T2 extends T1 { int x = 2; }
class T3 extends T2 {
   int x = 3;
   void test() {
      System.out.println("x=\t\t"+x);
      System.out.println("super.x=\t\t"+super.x);
      System.out.println("((T2)this).x=\t"+((T2)this).x);
      System.out.println("((T1)this).x=\t"+((T1)this).x);
      System.out.println("((I)this).x=\t"+((I)this).x);
   }
}
class Test {
   public static void main(String[] args) {
      new T3().test();
   }
}
```

which produces the output:

```
x=              3
super.x=        2
((T2)this).x=   2
((T1)this).x=   1
((I)this).x=    0
```

Within class `T3`, the expression `super.x` is treated exactly as if it were:

```
((T2)this).x
```

Suppose that 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 `((`*S*`)T.this).`*`name`*.

Thus the expression `T.super.`*`name`* can access the field named *name* that is visible in the class named by *S*, even if that field is hidden by a declaration of a field named *name* in the class named by *T*.

It is a compile time error if the class denoted by *T* is not a lexically enclosing class of the current class.

15.12 Method Invocation Expressions

A method invocation expression is used to invoke a class or instance method.

MethodInvocation:
MethodName (*ArgumentList*^{opt})
Primary . *Identifier* (*ArgumentList*^{opt})
`super` . *Identifier* (*ArgumentList*^{opt})
ClassName . `super` . *Identifier* (*ArgumentList*^{opt})

The definition of *ArgumentList* from §15.10 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 §15.13.3.

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 non-hidden method declaration with that name, then there must be an enclosing type declaration *T* of which that method is a member. The class or interface to search is *T*.

- 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*; then the name of the method is the *Identifier* and the class or interface to search is the declared type of the field named by the *FieldName*.

- If the form is *Primary* `.` *Identifier*, then the name of the method is the *Identifier* and the class or interface to be searched is the type of the *Primary* expression.
- If the form is `super .` *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 any of the following situations occur:
  - *T* is the class `Object`.
  - *T* is an interface.
  - The method invocation occurs in a a `static` method, static initializer, or the initializer for a `static` variable of *T*.

It follows that a method invocation of this form may appear only in a class other than `Object`, and only in the body of an instance method, instance initializer, the body of a constructor, or an initializer for an instance variable.

- If the form is *ClassName*`.super .` *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 any of the following situations occur:
  - *T* is the class `Object`.
  - *T* is an interface.
  - The method invocation occurs in a a `static` method, static initializer, or the initializer for a `static` variable of *T*.

It follows that a method invocation of this form may appear only in a class other than `Object`, and only in the body of an instance method, instance initializer, the body of a constructor, or an initializer for an instance variable.

15.12.2 Compile-Time Step 2: Determine Method Signature

The hand-writing experts were called upon for their opinion of the signature . . .
—Agatha Christie, *The Mysterious Affair at Styles* (1920), Chapter 11

The second step searches the class or interface determined in the previous step for method declarations. This step uses the name of the method and the types of the argument expressions to locate method declarations that are both *applicable* and *accessible*, that is, declarations that can be correctly invoked on the given arguments. There may be more than one such method declaration, in which case the *most specific* one is chosen. The descriptor (signature plus return type) of the most specific method declaration is one used at run time to do the method dispatch.

15.12.2.1 *Find Methods that are Applicable and Accessible*

A method declaration is *applicable* to a method invocation if and only if both of the following are true:

- The number of parameters in the method declaration equals the number of argument expressions in the method invocation.
- The type of each actual argument can be converted by method invocation conversion (§5.3) to the type of the corresponding parameter. Method invocation conversion is the same as assignment conversion (§5.2), except that constants of type `int` are never implicitly narrowed to `byte`, `short`, or `char`.

The class or interface determined by the process described in §15.13.1 is searched for all method declarations applicable to this method invocation; method definitions inherited from superclasses and superinterfaces are included in this search.

Whether a method declaration is *accessible* to a method invocation depends on the access modifier (`public`, none, `protected`, or `private`) in the method declaration and on where the method invocation appears.

If the class or interface has no method declaration that is both applicable and accessible, then a compile-time error occurs.

In the example program:

```
public class Doubler {
   static int two() { return two(1); }
```

```
    private static int two(int i) { return 2*i; }
}
class Test extends Doubler {
    public static long two(long j) {return j+j; }
    public static void main(String[] args) {
        System.out.println(two(3));
        System.out.println(Doubler.two(3)); // compile-time error
    }
}
```

for the method invocation `two(1)` within class `Doubler`, there are two accessible methods named `two`, but only the second one is applicable, and so that is the one invoked at run time. For the method invocation `two(3)` within class `Test`, there are two applicable methods, but only the one in class `Test` is accessible, and so that is the one to be invoked at run time (the argument `3` is converted to type `long`). For the method invocation `Doubler.two(3)`, the class `Doubler`, not class `Test`, is searched for methods named `two`; the only applicable method is not accessible, and so this method invocation causes a compile-time error.

Another example is:

```
class ColoredPoint {
    int x, y;
    byte color;
    void setColor(byte color) { this.color = color; }
}
class Test {
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        byte color = 37;
        cp.setColor(color);
        cp.setColor(37);                    // compile-time error
    }
}
```

Here, a compile-time error occurs for the second invocation of `setColor`, because no applicable method can be found at compile time. The type of the literal `37` is `int`, and `int` cannot be converted to `byte` by method invocation conversion. Assignment conversion, which is used in the initialization of the variable `color`, performs an implicit conversion of the constant from type `int` to `byte`, which is permitted because the value `37` is small enough to be represented in type `byte`; but such a conversion is not allowed for method invocation conversion.

If the method `setColor` had, however, been declared to take an `int` instead of a `byte`, then both method invocations would be correct; the first invocation would

be allowed because method invocation conversion does permit a widening conversion from `byte` to `int`. However, a narrowing cast would then be required in the body of `setColor`:

```
void setColor(int color) { this.color = (byte)color; }
```

15.12.2.2 *Choose the Most Specific Method*

If more than one method declaration 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 declaration 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.

The precise definition is as follows. Let *m* be a name and suppose that there are two declarations of methods named *m*, each having *n* parameters. Suppose that one declaration appears within a class or interface *T* and that the types of the parameters are *T1*, ..., *Tn*; suppose moreover that the other declaration appears within a class or interface *U* and that the types of the parameters are *U1*, ..., *Un*. Then the method *m* declared in *T* is *more specific* than the method *m* declared in *U* if and only if both of the following are true:

- *T* can be converted to *U* by method invocation conversion.
- *Tj* can be converted to *Uj* by method invocation conversion, for all *j* from 1 to *n*.

A method is said to be *maximally specific* for a method invocation if it is applicable and accessible and there is no other applicable and accessible method that is more specific.

If there is exactly one maximally specific method, then it is in fact *the most specific* method; it is necessarily more specific than any other method that is applicable and accessible. It is then subjected to some further compile-time checks as described in §15.13.3.

It is possible that no method is the most specific, because there are two or more maximally specific method declarations. In this case, we say that the method invocation is *ambiguous*, and a compile-time error occurs.

15.12.2.3 *Example: Overloading Ambiguity*

Consider the example:

```
class Point { int x, y; }
```

```
class ColoredPoint extends Point { int color; }


class Test {
   static void test(ColoredPoint p, Point q) {
      System.out.println("(ColoredPoint, Point)");
   }

   static void test(Point p, ColoredPoint q) {
      System.out.println("(Point, ColoredPoint)");
   }

   public static void main(String[] args) {
      ColoredPoint cp = new ColoredPoint();
      test(cp, cp);                        // compile-time error
   }

}
```

This example produces an error at compile time. The problem is that there are two declarations of *test* that are applicable and accessible, and neither is more specific than the other. Therefore, the method invocation is ambiguous.

If a third definition of `test` were added:

```
static void test(ColoredPoint p, ColoredPoint q) {
   System.out.println("(ColoredPoint, ColoredPoint)");
}
```

then it would be more specific than the other two, and the method invocation would no longer be ambiguous.

15.12.2.4 *Example: Return Type Not Considered*

As another example, consider:

```
class Point { int x, y; }

class ColoredPoint extends Point { int color; }

class Test {
   static int test(ColoredPoint p) {
      return p.color;
   }

   static String test(Point p) {
      return "Point";
   }

   public static void main(String[] args) {
      ColoredPoint cp = new ColoredPoint();
```

```
        String s = test(cp);                    // compile-time error
    }

}
```

Here the most specific declaration of method `test` is the one taking a parameter of type `ColoredPoint`. Because the result type of the method is `int`, a compile-time error occurs because an `int` cannot be converted to a `String` by assignment conversion. This example shows that the result types of methods do not participate in resolving overloaded methods, so that the second `test` method, which returns a `String`, is not chosen, even though it has a result type that would allow the example program to compile without error.

15.12.2.5 *Example: Compile-Time Resolution*

The most applicable method is chosen at compile time; its descriptor determines what method is actually executed at run time. If a new method is added to a class, then source code that was compiled with the old definition of the class might not use the new method, even if a recompilation would cause this method to be chosen.

So, for example, consider two compilation units, one for class `Point`:

```
package points;

public class Point {
    public int x, y;

    public Point(int x, int y) { this.x = x; this.y = y; }

    public String toString() { return toString(""); }

    public String toString(String s) {
        return "(" + x + "," + y + s + ")";
    }
}
```

and one for class `ColoredPoint`:

```
package points;

public class ColoredPoint extends Point {
    public static final int
        RED = 0, GREEN = 1, BLUE = 2;

    public static String[] COLORS =
        { "red", "green", "blue" };

    public byte color;
```

```
    public ColoredPoint(int x, int y, int color) {
        super(x, y); this.color = (byte)color;
    }

    /** Copy all relevant fields of the argument into
        this ColoredPoint object. */
    public void adopt(Point p) { x = p.x; y = p.y; }

    public String toString() {
        String s = "," + COLORS[color];
        return super.toString(s);
    }

}
```

Now consider a third compilation unit that uses `ColoredPoint`:

```
import points.*;

class Test {
    public static void main(String[] args) {
        ColoredPoint cp =
            new ColoredPoint(6, 6, ColoredPoint.RED);
        ColoredPoint cp2 =
            new ColoredPoint(3, 3, ColoredPoint.GREEN);
        cp.adopt(cp2);
        System.out.println("cp: " + cp);
    }
}
```

The output is:

```
cp: (3,3,red)
```

The application programmer who coded class `Test` has expected to see the word `green`, because the actual argument, a `ColoredPoint`, has a `color` field, and `color` would seem to be a "relevant field" (of course, the documentation for the package `Points` ought to have been much more precise!).

Notice, by the way, that the most specific method (indeed, the only applicable method) for the method invocation of `adopt` has a signature that indicates a method of one parameter, and the parameter is of type `Point`. This signature becomes part of the binary representation of class `Test` produced by the compiler and is used by the method invocation at run time.

Suppose the programmer reported this software error and the maintainer of the `points` package decided, after due deliberation, to correct it by adding a method to class `ColoredPoint`:

```
public void adopt(ColoredPoint p) {
    adopt((Point)p); color = p.color;
}
```

If the application programmer then runs the old binary file for `Test` with the new binary file for `ColoredPoint`, the output is still:

```
cp: (3,3,red)
```

because the old binary file for `Test` still has the descriptor "one parameter, whose type is `Point`; `void`" associated with the method call `cp.adopt(cp2)`. If the source code for `Test` is recompiled, the compiler will then discover that there are now two applicable `adopt` methods, and that the signature for the more specific one is "one parameter, whose type is `ColoredPoint`; `void`"; running the program will then produce the desired output:

```
cp: (3,3,green)
```

With forethought about such problems, the maintainer of the `points` package could fix the `ColoredPoint` class to work with both newly compiled and old code, by adding defensive code to the old `adopt` method for the sake of old code that still invokes it on `ColoredPoint` arguments:

```
public void adopt(Point p) {
   if (p instanceof ColoredPoint)
      color = ((ColoredPoint)p).color;
   x = p.x; y = p.y;
}
```

Ideally, source code should be recompiled whenever code that it depends on is changed. However, in an environment where different classes are maintained by different organizations, this is not always feasible. Defensive programming with careful attention to the problems of class evolution can make upgraded code much more robust. See §13 for a detailed discussion of binary compatibility and type evolution.

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. Three 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 invocation appears within a `static` method, a static initializer, or the initializer for a `static` variable, then the compile-time declaration must be `static`. If, instead, 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 cannot

be used to invoke an instance method in places where `this` (§15.9.3) is not defined.)

- If the method invocation has, before the left parenthesis, a *MethodName* of the form *TypeName* `.` *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 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 (§14.13), 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`, as declared in the compile-time declaration.
- 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 four productions for *MethodInvocation* (§15.13) is involved:

- 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 (§8.1.2) of the class whose declaration immediately contains the method invocation. Then the target reference is the *n*th lexically enclosing instance (§8.1.2) of `this`. It is a compile-time error if the *n*th lexically enclosing instance (§8.1.2) 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.
    - Otherwise, the target reference is the value of the expression *FieldName*.
- 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.

- If the third production for *MethodInvocation*, which includes the keyword `super`, is involved, then the target reference is the value of `this`.
- If the fourth production for *MethodInvocation*, *ClassName*`.super`, is involved, then the target reference is the value of *ClassName*`.this`.

15.12.4.2 *Evaluate Arguments*

The argument expressions 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.

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 *m* be the name of the method, as determined at compile time (§15.13.3). An implementation of the Java programming language must insure, 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 insure, 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 `IllegalAccessError` occurs (§12.3).

15.12.4.4 *Locate Method to Invoke*

Here inside my paper cup,
Everything is looking up.
—Jim Webb, *Paper Cup* (1967)

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 target object is an array, *R* is the class `Object`. (Note that for invocation mode `interface`, *R* necessarily implements *T*; for invocation mode `virtual`, *R* is necessarily either *T* or a subclass of *T*.)
- If the invocation mode is `super`, then *S* is initially the superclass of the class *C* that contains 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.

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 (§15.13.3), 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 (§8.4.6.1) *X.m*, then the method declared in *S* is the method to be invoked, and the procedure terminates.

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 will always 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, Second Edition*.

We note that the dynamic lookup process, while described here explicitly, will often be implemented implicitly, for example as a side-effect of the construction and use of per-class method dispatch tables, or the construction of other per-class structures used for efficient dispatch.

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, an `OutOfMemoryError` 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 (§5.1.8).

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 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 (§14.18).

15.12.4.6 *Example: Target Reference and Static Methods*

When a target reference is computed and then discarded because the invocation mode is `static`, the reference is not examined to see whether it is `null`:

```
class Test {
    static void mountain() {
        System.out.println("Monadnock");
    }
    static Test favorite(){
        System.out.print("Mount ");
        return null;
    }
    public static void main(String[] args) {
        favorite().mountain();
    }
}
```

which prints:

```
Mount Monadnock
```

Here `favorite` returns `null`, yet no `NullPointerException` is thrown.

15.12.4.7 *Example: Evaluation Order*

As part of an instance method invocation (§15.13), there is an expression that denotes the object to be invoked. This expression appears to be fully evaluated before any part of any argument expression to the method invocation is evaluated.

So, for example, in:

```
class Test {
    public static void main(String[] args) {
        String s = "one";
```

```
        if (s.startsWith(s = "two"))
            System.out.println("oops");
    }
}
```

the occurrence of `s` before "`.startsWith`" is evaluated first, before the argument expression `s="two"`. Therefore, a reference to the string `"one"` is remembered as the target reference before the local variable `s` is changed to refer to the string `"two"`. As a result, the `startsWith` method is invoked for target object `"one"` with argument `"two"`, so the result of the invocation is `false`, as the string `"one"` does not start with `"two"`. It follows that the test program does not print "`oops`".

15.12.4.8 *Example: Overriding*

In the example:

```
class Point {
    final int EDGE = 20;
    int x, y;
    void move(int dx, int dy) {
        x += dx; y += dy;
        if (Math.abs(x) >= EDGE || Math.abs(y) >= EDGE)
            clear();
    }
    void clear() {
        System.out.println("\tPoint clear");
        x = 0; y = 0;
    }
}
class ColoredPoint extends Point {
    int color;

    void clear() {
        System.out.println("\tColoredPoint clear");
        super.clear();
        color = 0;
    }
}
```

the subclass `ColoredPoint` extends the `clear` abstraction defined by its superclass `Point`. It does so by overriding the `clear` method with its own method, which invokes the `clear` method of its superclass, using the form `super.clear`.

This method is then invoked whenever the target object for an invocation of `clear` is a `ColoredPoint`. Even the method `move` in `Point` invokes the `clear` method of class `ColoredPoint` when the class of `this` is `ColoredPoint`, as shown by the output of this test program:

```
class Test {
   public static void main(String[] args) {
      Point p = new Point();
      System.out.println("p.move(20,20):");
      p.move(20, 20);
      ColoredPoint cp = new ColoredPoint();
      System.out.println("cp.move(20,20):");
      cp.move(20, 20);
      p = new ColoredPoint();
      System.out.println("p.move(20,20), p colored:");
      p.move(20, 20);
   }
}
```

which is:

```
p.move(20,20):
   Point clear
cp.move(20,20):
   ColoredPoint clear
   Point clear
p.move(20,20), p colored:
   ColoredPoint clear
   Point clear
```

Overriding is sometimes called "late-bound self-reference"; in this example it means that the reference to `clear` in the body of `Point.move` (which is really syntactic shorthand for `this.clear`) invokes a method chosen "late" (at run time, based on the run-time class of the object referenced by `this`) rather than a method chosen "early" (at compile time, based only on the type of `this`). This provides the programmer a powerful way of extending abstractions and is a key idea in object-oriented programming.

15.12.4.9 *Example: Method Invocation using* `super`

An overridden instance method of a superclass may be accessed by using the keyword `super` to access the members of the immediate superclass, bypassing any overriding declaration in the class that contains the method invocation.

When accessing an instance variable, `super` means the same as a cast of `this` (§15.12.2), but this equivalence does not hold true for method invocation. This is demonstrated by the example:

```
class T1 {
   String s() { return "1"; }
}

class T2 extends T1 {
   String s() { return "2"; }
}

class T3 extends T2 {
   String s() { return "3"; }

   void test() {
      System.out.println("s()=\t\t"+s());
      System.out.println("super.s()=\t"+super.s());
      System.out.print("((T2)this).s()=\t");
         System.out.println(((T2)this).s());
      System.out.print("((T1)this).s()=\t");
         System.out.println(((T1)this).s());
   }

}

class Test {
   public static void main(String[] args) {
      T3 t3 = new T3();
      t3.test();
   }
}
```

which produces the output:

```
s()=            3
super.s()=      2
((T2)this).s()= 3
((T1)this).s()= 3
```

The casts to types `T1` and `T2` do not change the method that is invoked, because the instance method to be invoked is chosen according to the run-time class of the object referred to be `this`. A cast does not change the class of an object; it only checks that the class is compatible with the specified type.

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 (§15.11).

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 results. Then the type of the array access expression is *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.)

15.13.2 Examples: Array Access Evaluation Order

In an array access, the expression to the left of the brackets appears to be fully evaluated before any part of the expression within the brackets is evaluated. For example, in the (admittedly monstrous) expression `a[(a=b)[3]]`, the expression `a` is fully evaluated before the expression `(a=b)[3]`; this means that the original

value of `a` is fetched and remembered while the expression `(a=b)[3]` is evaluated. This array referenced by the original value of `a` is then subscripted by a value that is element `3` of another array (possibly the same array) that was referenced by `b` and is now also referenced by `a`.

Thus, the example:

```
class Test {
   public static void main(String[] args) {
      int[] a = { 11, 12, 13, 14 };
      int[] b = { 0, 1, 2, 3 };
      System.out.println(a[(a=b)[3]]);
   }
}
```

prints:

```
14
```

because the monstrous expression's value is equivalent to `a[b[3]]` or `a[3]` or `14`.

If evaluation of the expression to the left of the brackets completes abruptly, no part of the expression within the brackets will appear to have been evaluated. Thus, the example:

```
class Test {
   public static void main(String[] args) {
      int index = 1;
      try {
         skedaddle()[index=2]++;
      } catch (Exception e) {
         System.out.println(e + ", index=" + index);
      }
   }
   static int[] skedaddle() throws Exception {
      throw new Exception("Ciao");
   }
}
```

prints:

```
java.lang.Exception: Ciao, index=1
```

because the embedded assignment of `2` to `index` never occurs.

If the array reference expression produces `null` instead of a reference to an array, then a `NullPointerException` is thrown at run time, but only after all parts of the array access expression have been evaluated and only if these evaluations completed normally. Thus, the example:

```
class Test {
   public static void main(String[] args) {
      int index = 1;
      try {
         nada()[index=2]++;
      } catch (Exception e) {
         System.out.println(e + ", index=" + index);
      }
   }

   static int[] nada() { return null; }

}
```

prints:

```
java.lang.NullPointerException, index=2
```

because the embedded assignment of `2` to `index` occurs before the check for a null pointer. As a related example, the program:

```
class Test {
   public static void main(String[] args) {
      int[] a = null;
      try {
         int i = a[vamoose()];
         System.out.println(i);
      } catch (Exception e) {
         System.out.println(e);
      }
   }

   static int vamoose() throws Exception {
      throw new Exception("Twenty-three skidoo!");
   }

}
```

always prints:

```
java.lang.Exception: Twenty-three skidoo!
```

A `NullPointerException` never occurs, because the index expression must be completely evaluated before any part of the indexing operation occurs, and that includes the check as to whether the value of the left-hand operand is `null`.

15.14 Postfix Expressions

Postfix expressions include uses of the postfix ++ and -- operators. Also, as discussed in §15.9, 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 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 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) 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 value set conversion (§5.1.8). 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 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.

15.14.2 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 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) 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 value set conversion (§5.1.8). 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 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

The following productions from §15.17 are repeated here for convenience:

CastExpression:
`(` *PrimitiveType* `)` *UnaryExpression*
`(` *ReferenceType* `)` *UnaryExpressionNotPlusMinus*

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 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) 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 value set conversion (§5.1.8). 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 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 --

He must increase, but I must decrease.
—John 3:30

A unary expression preceded by a `--` operator is a prefix decrement expression. The result of the unary expression must be a variable of 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) 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 value set conversion (§5.1.8). 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 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 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 -

It is so very agreeable to hear a voice and to see all the signs of that expression.
—Gertrude Stein, *Rooms* (1914), in *Tender Buttons*

The type of the operand expression of the unary - operator must be 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 (§5.1.8). 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 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 a compile-time error occurs. The type of the unary logical complement expression is `boolean`.

At run time, the value of the unary logical complement expression is `true` if the operand value is `false` and `false` if the operand value is `true`.

15.16 Cast Expressions

My days among the dead are passed;
Around me I behold,

Where'er these casual eyes are cast,
The mighty minds of old . . .

—Robert Southey (1774–1843),
Occasional Pieces, xviii

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 Dims*^{opt} `)` *UnaryExpression*
`(` *ReferenceType* `)` *UnaryExpressionNotPlusMinus*

See §15.16 for a discussion of the distinction between *UnaryExpression* and *UnaryExpressionNotPlusMinus*.

The type of a cast expression is 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.

At run time, the operand value is converted by casting conversion (§5.5) to the type specified by the cast operator.

Not all casts are permitted by the language. Some casts result in an error at compile time. For example, a primitive value may not be cast to a reference type. Some casts can be proven, at compile time, always to be correct at run time. For example, it is always correct to convert a value of a class type to the type of its superclass; such a cast should require no special action at run time. Finally, some casts cannot be proven to be either always correct or always incorrect at compile time. Such casts require a test at run time.

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 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 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.

Note that binary numeric promotion performs value set conversion (§5.1.8).

15.17.1 Multiplication Operator *

Entia non sunt multiplicanda praeter necessitatem.
—William of Occam (c. 1320)

The binary `*` operator performs multiplication, producing the product of its operands. Multiplication is a commutative operation if the operand expressions have no side effects. While integer multiplication is associative when the operands are all of the same type, 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 governed 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 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 /

Gallia est omnis divisa in partes tres.
—Julius Caesar, *Commentaries on the Gallic Wars* (58 B.C.)

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 specification of IEEE 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, 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 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 %

And on the pedestal these words appear:
"My name is Ozymandias, king of kings:
Look on my works, ye Mighty, and despair!"
Nothing beside remains.

—Percy Bysshe Shelley, *Ozymandias* (1817)

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. This identity holds even in the special case that the dividend is the negative integer of largest possible magnitude for its type and the divisor is -1 (the remainder is 0). It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative, and can be positive only if the dividend is positive; moreover, the magnitude of the result is always less than the magnitude of the divisor. If the value of the divisor for an integer remainder operator is 0, then an ArithmeticException is thrown.

Examples:

```
5%3 produces 2          (note that 5/3 produces 1)
5%(-3) produces 2       (note that 5/(-3) produces -1)
(-5)%3 produces -2      (note that (-5)/3 produces -1)
(-5)%(-3) produces -2   (note that (-5)/(-3) produces 1)
```

Where should he have this gold?
It is some poor fragment, some slender ort of his remainder.

—William Shakespeare, *Timon of Athens* (1623), Act IV, scene i

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 Java floating-point remainder operation is determined by the rules of IEEE 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.

Examples:

`5.0%3.0` produces `2.0`
`5.0%(-3.0)` produces `2.0`
`(-5.0)%3.0` produces `-2.0`
`(-5.0)%(-3.0)` produces `-2.0`

The coffee we made of this water was the meanest compound man has yet invented. It was really viler to the taste than the unameliorated water itself. Mr. Ballou, being the architect and builder of the beverage felt constrained to endorse and uphold it, and so drank half a cup, by little sips, making

shift to praise it faintly the while, but finally threw out the remainder,
and said frankly it was "too technical for him."
—Mark Twain, *Roughing It* (1871), Chapter 27

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 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 primitive numeric type, or a compile-time error occurs.

15.18.1 String Concatenation Operator +

"*The fifth string was added after an unfortunate*
episode in the Garden of Eden . . ."
—John Philip Sousa, *The Fifth String* (1902), Chapter 6

If only one operand expression is of type `String`, then string conversion is performed on the other operand to produce a string at run time. The result is a reference to a newly created `String` object 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.

15.18.1.1 *String Conversion*

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`.

15.18.1.2 *Optimization of String Concatenation*

An implementation may choose to perform conversion and concatenation in one step to avoid creating and then discarding an intermediate `String` object. To increase the performance of repeated string concatenation, a Java compiler may use the `StringBuffer` class or a similar technique to reduce the number of intermediate `String` objects that are created by evaluation of an expression.

For primitive types, an implementation may also optimize away the creation of a wrapper object by converting directly from a primitive type to a string.

15.18.1.3 *Examples of String Concatenation*

The example expression:

```
"The square root of 2 is " + Math.sqrt(2)
```

produces the result:

```
"The square root of 2 is 1.4142135623730952"
```

The + operator is syntactically left-associative, no matter whether it is later determined by type analysis to represent string concatenation or addition. In some cases care is required to get the desired result. For example, the expression:

```
a + b + c
```

is always regarded as meaning:

```
(a + b) + c
```

Therefore the result of the expression:

```
1 + 2 + " fiddlers"
```

is:

```
"3 fiddlers"
```

but the result of:

```
"fiddlers " + 1 + 2
```

is:

```
"fiddlers 12"
```

In this jocular little example:

```
class Bottles {
   static void printSong(Object stuff, int n) {
      String plural = (n == 1) ? "" : "s";
      loop: while (true) {
         System.out.println(n + " bottle" + plural
            + " of " + stuff + " on the wall,");
         System.out.println(n + " bottle" + plural
            + " of " + stuff + ";");
         System.out.println("You take one down "
            + "and pass it around:");
         --n;
         plural = (n == 1) ? "" : "s";
         if (n == 0)
            break loop;
         System.out.println(n + " bottle" + plural
            + " of " + stuff + " on the wall!");
         System.out.println();
      }
      System.out.println("No bottles of " +
                         stuff + " on the wall!");
   }
}
```

the method `printSong` will print a version of a children's song. Popular values for stuff include `"pop"` and `"beer"`; the most popular value for `n` is `100`. Here is the output that results from `Bottles.printSong("slime", 3)`:

```
3 bottles of slime on the wall,
3 bottles of slime;
You take one down and pass it around:
2 bottles of slime on the wall!

2 bottles of slime on the wall,
2 bottles of slime;
You take one down and pass it around:
```

```
1 bottle of slime on the wall!

1 bottle of slime on the wall,
1 bottle of slime;
You take one down and pass it around:
No bottles of slime on the wall!
```

In the code, note the careful conditional generation of the singular "`bottle`" when appropriate rather than the plural "`bottles`"; note also how the string concatenation operator was used to break the long constant string:

```
"You take one down and pass it around:"
```

into two pieces to avoid an inconveniently long line in the source code.

15.18.2 Additive Operators (+ and -) for Numeric Types

We discern a grand force in the lover which he lacks whilst a free man; but there is a breadth of vision in the free man which in the lover we vainly seek. Where there is much bias there must be some narrowness, and love, though added emotion, is subtracted capacity.

—Thomas Hardy, *Far from the Madding Crowd* (1874), Act IV, scene i

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.

Note that binary numeric promotion performs value set conversion (§5.1.8).

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 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 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 nega-

tion. 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

What, I say, is to become of those wretches?
. . . What more can you say to them than "shift for yourselves?"
—Thomas Paine, *The American Crisis* (1780)

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 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 `&` (§15.23.1) 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 `&` (§15.23.1) 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 nonnegative 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>>>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>>s`; if `n` is negative, the result is equal to that of the expression `(n>>s)+(2<<~s)` if the type of the left-hand operand is `int`, and to the result of the expression `(n>>s)+(2L<<~s)` if the type of the left-hand operand is `long`. The added term `(2<<~s)` or `(2L<<~s)` cancels out the propagated sign bit. (Note that, because of the implicit masking of the right-hand operand of a shift operator, `~s` as a shift distance is equivalent to `31-s` when shifting an `int` value and to `63-s` when shifting a `long` value.)

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 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 this promoted type is `float` or `double`, then floating-point comparison is performed.

Note that binary numeric promotion performs value set conversion (§5.1.8). 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. Therefore, `-0.0<0.0` is `false`, for example, but `-0.0<=0.0` is `true`. (Note, however, that the methods `Math.min` and `Math.max` treat negative zero as being strictly smaller than positive zero.)

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. The *ReferenceType* mentioned after the `instanceof` operator must denote a reference type; otherwise, a compile-time error occurs.

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 (§15.17) 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`.

Consider the example program:

```
class Point { int x, y; }

class Element { int atomicNumber; }

class Test {
   public static void main(String[] args) {
      Point p = new Point();
      Element e = new Element();
      if (e instanceof Point) {          // compile-time error
         System.out.println("I get your point!");
         p = (Point)e;                   // compile-time error
      }
   }
}
```

This example results in two compile-time errors. The cast `(Point)e` is incorrect because no instance of `Element` or any of its possible subclasses (none are shown here) could possibly be an instance of any subclass of `Point`. The `instanceof` expression is incorrect for exactly the same reason. If, on the other hand, the class `Point` were a subclass of `Element` (an admittedly strange notion in this example):

```
class Point extends Element { int x, y; }
```

then the cast would be possible, though it would require a run-time check, and the `instanceof` expression would then be sensible and valid. The cast `(Point)e` would never raise an exception because it would not be executed if the value of `e` could not correctly be cast to type `Point`.

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 of numeric type, or two operands of type `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 primitive 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.

Note that binary numeric promotion performs value set conversion (§5.1.8). 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. (The methods `Float.isNaN` and `Double.isNaN` may also be used to test whether a value is NaN.)
- Positive zero and negative zero are considered equal. Therefore, `-0.0==0.0` is `true`, for example.
- 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`, then the operation is boolean equality. The `boolean` equality operators are associative.

The result of == is `true` if the operands 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`. Thus != behaves the same as ^ (§15.23.2) when applied to boolean operands.

15.21.3 Reference Equality Operators == and !=

Things are more like they are now than they ever were before.
—Dwight D. Eisenhower

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 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.

For example, the result of the expression `0xff00 & 0xf0f0` is `0xf000`. The result of `0xff00 ^ 0xf0f0` is `0x0ff0`.The result of `0xff00 | 0xf0f0` is `0xfff0`.

15.22.2 Boolean Logical Operators &, ^, and |

When both operands of a &, ^, or | operator are of type `boolean`, then the type of the bitwise operator expression is `boolean`.

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 & (§15.23.2), 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 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 its 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 and its 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 | (§15.23.2), 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 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 its 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 and its value becomes the value of the conditional-or 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.

15.25 Conditional Operator `? :`

But be it as it may. I here entail
The crown to thee and to thine heirs for ever;
Conditionally . . .
—William Shakespeare, *Henry VI, Part III* (1623), Act I, scene i

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 a compile-time error occurs.

The conditional operator may be used to choose between second and third operands of numeric type, or second and third operands of type `boolean`, or second and third operands that are each of either reference type or the null type. All other cases result in a compile-time error.

Note that it is not permitted for either the second or the third operand expression to be an invocation of a `void` method. In fact, it is not permitted for a conditional expression to appear in any context where an invocation of a `void` method could appear (§14.8).

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.
- Otherwise, if the second and third operands have numeric type, then there are several cases:
  - If one of the operands is of type `byte` and the other is of type `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*.
  - 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. Note that binary numeric promotion performs value set conversion (§5.1.8).
- 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.
- If the second and third operands are of different reference types, then it must be possible to convert one of the types to the other type (call this latter type *T*) by assignment conversion (§5.2); the type of the conditional expression is *T*. It is a compile-time error if neither type is assignment compatible with the other type.

At run time, the first operand expression of the conditional expression is evaluated first; its `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. 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 (§15.12) or an array access (§15.14). The type of the assignment expression is the type of the variable.

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 a blank final variable (§4.5.4)) , because when an access of 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 two ways. If the left-hand operand expression is not an array access expression, then 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 (§5.1.8) 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 (§15.14), then many steps are required:

- 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.
  - 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 compo-

nent, is subjected to value set conversion (§5.1.8) 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*C 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.

  The 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 the 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. This check is similar to a narrowing cast (§5.5, §15.17), except that if the check fails, an `ArrayStoreException` is thrown rather than a `ClassCastException`. Therefore:

  - 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.

The rules for assignment to an array component are illustrated by the following example program:

```
class ArrayReferenceThrow extends RuntimeException { }
class IndexThrow extends RuntimeException { }
class RightHandSideThrow extends RuntimeException { }
class IllustrateSimpleArrayAssignment {
   static Object[] objects = { new Object(), new Object() };
   static Thread[] threads = { new Thread(), new Thread() };
   static Object[] arrayThrow() {
      throw new ArrayReferenceThrow();
   }
   static int indexThrow() { throw new IndexThrow(); }
   static Thread rightThrow() {
      throw new RightHandSideThrow();
   }
   static String name(Object q) {
      String sq = q.getClass().getName();
      int k = sq.lastIndexOf('.');
      return (k < 0) ? sq : sq.substring(k+1);
   }
```

```
static void testFour(Object[] x, int j, Object y) {
   String sx = x == null ? "null" : name(x[0]) + "s";
   String sy = name(y);
   System.out.println();
   try {
      System.out.print(sx + "[throw]=throw => ");
      x[indexThrow()] = rightThrow();
      System.out.println("Okay!");
   } catch (Throwable e) { System.out.println(name(e)); }
   try {
      System.out.print(sx + "[throw]=" + sy + " => ");
      x[indexThrow()] = y;
      System.out.println("Okay!");
   } catch (Throwable e) { System.out.println(name(e)); }
   try {
      System.out.print(sx + "[" + j + "]=throw => ");
      x[j] = rightThrow();
      System.out.println("Okay!");
   } catch (Throwable e) { System.out.println(name(e)); }
   try {
      System.out.print(sx + "[" + j + "]=" + sy + " => ");
      x[j] = y;
      System.out.println("Okay!");
   } catch (Throwable e) { System.out.println(name(e)); }
}
public static void main(String[] args) {
   try {
      System.out.print("throw[throw]=throw => ");
      arrayThrow()[indexThrow()] = rightThrow();
      System.out.println("Okay!");
   } catch (Throwable e) { System.out.println(name(e)); }
   try {
      System.out.print("throw[throw]=Thread => ");
      arrayThrow()[indexThrow()] = new Thread();
      System.out.println("Okay!");
   } catch (Throwable e) { System.out.println(name(e)); }
   try {
      System.out.print("throw[1]=throw => ");
      arrayThrow()[1] = rightThrow();
      System.out.println("Okay!");
   } catch (Throwable e) { System.out.println(name(e)); }
   try {
      System.out.print("throw[1]=Thread => ");
      arrayThrow()[1] = new Thread();
      System.out.println("Okay!");
   } catch (Throwable e) { System.out.println(name(e)); }
```

```
            testFour(null, 1, new StringBuffer());
            testFour(null, 1, new StringBuffer());
            testFour(null, 9, new Thread());
            testFour(null, 9, new Thread());
            testFour(objects, 1, new StringBuffer());
            testFour(objects, 1, new Thread());
            testFour(objects, 9, new StringBuffer());
            testFour(objects, 9, new Thread());
            testFour(threads, 1, new StringBuffer());
            testFour(threads, 1, new Thread());
            testFour(threads, 9, new StringBuffer());
            testFour(threads, 9, new Thread());
        }
    }
```

This program prints:

```
throw[throw]=throw => ArrayReferenceThrow
throw[throw]=Thread => ArrayReferenceThrow
throw[1]=throw => ArrayReferenceThrow
throw[1]=Thread => ArrayReferenceThrow

null[throw]=throw => IndexThrow
null[throw]=StringBuffer => IndexThrow
null[1]=throw => RightHandSideThrow
null[1]=StringBuffer => NullPointerException

null[throw]=throw => IndexThrow
null[throw]=StringBuffer => IndexThrow
null[1]=throw => RightHandSideThrow
null[1]=StringBuffer => NullPointerException

null[throw]=throw => IndexThrow
null[throw]=Thread => IndexThrow
null[9]=throw => RightHandSideThrow
null[9]=Thread => NullPointerException

null[throw]=throw => IndexThrow
null[throw]=Thread => IndexThrow
null[9]=throw => RightHandSideThrow
null[9]=Thread => NullPointerException

Objects[throw]=throw => IndexThrow
Objects[throw]=StringBuffer => IndexThrow
Objects[1]=throw => RightHandSideThrow
Objects[1]=StringBuffer => Okay!

Objects[throw]=throw => IndexThrow
Objects[throw]=Thread => IndexThrow
Objects[1]=throw => RightHandSideThrow
Objects[1]=Thread => Okay!
```

```
Objects[throw]=throw => IndexThrow
Objects[throw]=StringBuffer => IndexThrow
Objects[9]=throw => RightHandSideThrow
Objects[9]=StringBuffer => ArrayIndexOutOfBoundsException

Objects[throw]=throw => IndexThrow
Objects[throw]=Thread => IndexThrow
Objects[9]=throw => RightHandSideThrow
Objects[9]=Thread => ArrayIndexOutOfBoundsException

Threads[throw]=throw => IndexThrow
Threads[throw]=StringBuffer => IndexThrow
Threads[1]=throw => RightHandSideThrow
Threads[1]=StringBuffer => ArrayStoreException

Threads[throw]=throw => IndexThrow
Threads[throw]=Thread => IndexThrow
Threads[1]=throw => RightHandSideThrow
Threads[1]=Thread => Okay!

Threads[throw]=throw => IndexThrow
Threads[throw]=StringBuffer => IndexThrow
Threads[9]=throw => RightHandSideThrow
Threads[9]=StringBuffer => ArrayIndexOutOfBoundsException

Threads[throw]=throw => IndexThrow
Threads[throw]=Thread => IndexThrow
Threads[9]=throw => RightHandSideThrow
Threads[9]=Thread => ArrayIndexOutOfBoundsException
```

The most interesting case of the lot is the one thirteenth from the end:

```
Threads[1]=StringBuffer => ArrayStoreException
```

which indicates that the attempt to store a reference to a `StringBuffer` into an array whose components are of type `Thread` throws an `ArrayStoreException`. The code is type-correct at compile time: the assignment has a left-hand side of type `Object[]` and a right-hand side of type `Object`. At run time, the first actual argument to method `testFour` is a reference to an instance of "array of `Thread`" and the third actual argument is a reference to an instance of class `StringBuffer`.

15.26.2 Compound Assignment Operators

All compound assignment operators require both operands to be of primitive type, except for +=, which allows the right-hand operand to be of any type if the left-hand operand is of type `String`.

A compound assignment expression of the form *E1 op= E2* is equivalent to *E1* = (*T*)((*E1*) *op* (*E2*)), where *T* is the type of *E1*, except that *E1* is evaluated

only once. Note that the implied cast to type *T* may be either an identity conversion (§5.1.1) or a narrowing primitive conversion (§5.1.3). For example, the following code is correct:

```
short x = 3;
x += 4.6;
```

and results in `x` having the value `7` because it is equivalent to:

```
short x = 3;
x = (short)(x + 4.6);
```

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 four 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 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 (the only possibility is an integer division by zero—see §15.18.2), 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 left-hand variable, subjected to value set conversion (§5.1.8) 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 (§15.14), then many steps are required:

- 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 assign-

ment 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. (For a simple assignment operator, the evaluation of the right-hand operand occurs before the checks of the array reference subexpression and the index subexpression, but for a compound assignment operator, the evaluation of the right-hand operand occurs after these checks.)
- 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 §15.18.2), 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 (§5.1.8) 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`. Therefore the run-time check that is sometimes required for the simple assignment operator is never required for a compound assignment operator.

- 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.

The rules for compound assignment to an array component are illustrated by the following example program:

```
class ArrayReferenceThrow extends RuntimeException { }
class IndexThrow extends RuntimeException { }
class RightHandSideThrow extends RuntimeException { }
class IllustrateCompoundArrayAssignment {
   static String[] strings = { "Simon", "Garfunkel" };
   static double[] doubles = { Math.E, Math.PI };
   static String[] stringsThrow() {
      throw new ArrayReferenceThrow();
   }
   static double[] doublesThrow() {
      throw new ArrayReferenceThrow();
   }
   static int indexThrow() { throw new IndexThrow(); }
   static String stringThrow() {
      throw new RightHandSideThrow();
   }
   static double doubleThrow() {
      throw new RightHandSideThrow();
   }
   static String name(Object q) {
      String sq = q.getClass().getName();
      int k = sq.lastIndexOf('.');
      return (k < 0) ? sq : sq.substring(k+1);
   }
   static void testEight(String[] x, double[] z, int j) {
      String sx = (x == null) ? "null" : "Strings";
      String sz = (z == null) ? "null" : "doubles";
      System.out.println();
      try {
         System.out.print(sx + "[throw]+=throw => ");
         x[indexThrow()] += stringThrow();
```

```
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sz + "[throw]+=throw => ");
            z[indexThrow()] += doubleThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[throw]+=\"heh\" => ");
            x[indexThrow()] += "heh";
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sz + "[throw]+=12345 => ");
            z[indexThrow()] += 12345;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[" + j + "]+=throw => ");
            x[j] += stringThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sz + "[" + j + "]+=throw => ");
            z[j] += doubleThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[" + j + "]+=\"heh\" => ");
            x[j] += "heh";
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sz + "[" + j + "]+=12345 => ");
            z[j] += 12345;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
    }
    public static void main(String[] args) {
        try {
            System.out.print("throw[throw]+=throw => ");
            stringsThrow()[indexThrow()] += stringThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[throw]+=throw => ");
            doublesThrow()[indexThrow()] += doubleThrow();
```

```
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[throw]+=\"heh\" => ");
            stringsThrow()[indexThrow()] += "heh";
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[throw]+=12345 => ");
            doublesThrow()[indexThrow()] += 12345;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]+=throw => ");
            stringsThrow()[1] += stringThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]+=throw => ");
            doublesThrow()[1] += doubleThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]+=\"heh\" => ");
            stringsThrow()[1] += "heh";
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]+=12345 => ");
            doublesThrow()[1] += 12345;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        testEight(null, null, 1);
        testEight(null, null, 9);
        testEight(strings, doubles, 1);
        testEight(strings, doubles, 9);
    }
}
```

This program prints:

```
throw[throw]+=throw => ArrayReferenceThrow
throw[throw]+=throw => ArrayReferenceThrow
throw[throw]+="heh" => ArrayReferenceThrow
throw[throw]+=12345 => ArrayReferenceThrow
throw[1]+=throw => ArrayReferenceThrow
throw[1]+=throw => ArrayReferenceThrow
```

```
throw[1]+="heh" => ArrayReferenceThrow
throw[1]+=12345 => ArrayReferenceThrow

null[throw]+=throw => IndexThrow
null[throw]+=throw => IndexThrow
null[throw]+="heh" => IndexThrow
null[throw]+=12345 => IndexThrow
null[1]+=throw => NullPointerException
null[1]+=throw => NullPointerException
null[1]+="heh" => NullPointerException
null[1]+=12345 => NullPointerException

null[throw]+=throw => IndexThrow
null[throw]+=throw => IndexThrow
null[throw]+="heh" => IndexThrow
null[throw]+=12345 => IndexThrow
null[9]+=throw => NullPointerException
null[9]+=throw => NullPointerException
null[9]+="heh" => NullPointerException
null[9]+=12345 => NullPointerException

Strings[throw]+=throw => IndexThrow
doubles[throw]+=throw => IndexThrow
Strings[throw]+="heh" => IndexThrow
doubles[throw]+=12345 => IndexThrow
Strings[1]+=throw => RightHandSideThrow
doubles[1]+=throw => RightHandSideThrow
Strings[1]+="heh" => Okay!
doubles[1]+=12345 => Okay!

Strings[throw]+=throw => IndexThrow
doubles[throw]+=throw => IndexThrow
Strings[throw]+="heh" => IndexThrow
doubles[throw]+=12345 => IndexThrow
Strings[9]+=throw => ArrayIndexOutOfBoundsException
doubles[9]+=throw => ArrayIndexOutOfBoundsException
Strings[9]+="heh" => ArrayIndexOutOfBoundsException
doubles[9]+=12345 => ArrayIndexOutOfBoundsException
```

The most interesting cases of the lot are tenth and eleventh from the end:

```
Strings[1]+=throw => RightHandSideThrow
doubles[1]+=throw => RightHandSideThrow
```

They are the cases where a right-hand side that throws an exception actually gets to throw the exception; moreover, they are the only such cases in the lot. This demonstrates that the evaluation of the right-hand operand indeed occurs after the checks for a null array reference value and an out-of-bounds index value.

The following program illustrates the fact that the value of the left-hand side of a compound assignment is saved before the right-hand side is evaluated:

```
class Test {
   public static void main(String[] args) {
      int k = 1;
      int[] a = { 1 };
      k += (k = 4) * (k + 2);
      a[0] += (a[0] = 4) * (a[0] + 2);
      System.out.println("k==" + k + " and a[0]==" + a[0]);
   }
}
```

This program prints:

```
k==25 and a[0]==25
```

The value `1` of `k` is saved by the compound assignment operator `+=` before its right-hand operand `(k = 4) * (k + 2)` is evaluated. Evaluation of this right-hand operand then assigns `4` to `k`, calculates the value `6` for `k + 2`, and then multiplies `4` by `6` to get `24`. This is added to the saved value `1` to get `25`, which is then stored into `k` by the `+=` operator. An identical analysis applies to the case that uses `a[0]`. In short, the statements

```
k += (k = 4) * (k + 2);
a[0] += (a[0] = 4) * (a[0] + 2);
```

behave in exactly the same manner as the statements:

```
k = k + (k = 4) * (k + 2);
a[0] = a[0] + (a[0] = 4) * (a[0] + 2);
```

15.27 Expression

An *Expression* is any assignment expression:

Expression:
AssignmentExpression

Unlike C and C++, the Java language has no comma operator.

15.28 Constant Expression

. . . the old and intent expression was a constant, not an occasional, thing . . .
—Charles Dickens, *A Tale of Two Cities* (1859)

ConstantExpression:
Expression

A compile-time *constant expression* is an expression denoting a value of primitive type or a `String` that is composed using only the following:

- Literals of primitive type and literals of type `String`
- 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 `? :`
- Simple names that refer to `final` variables whose initializers are constant expressions
- Qualified names of the form *TypeName* `.` *Identifier* that refer to `final` variables whose initializers are constant expressions

Compile-time constant expressions are used in `case` labels in `switch` statements (§14.10) and have a special significance for assignment conversion (§5.2).

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.

Examples of constant expressions:

```
true

(short)(1*2*3*4*5*6)

Integer.MAX_VALUE / 2

2.0 * Math.PI

"The integer " + Long.MAX_VALUE + " is mighty big."
```

. . . when faces of the throng turned toward him and ambiguous eyes stared into his, he assumed the most romantic of expressions . . .

—F. Scott Fitzgerald, *This Side of Paradise* (1920)

CHAPTER 16

Definite Assignment

All the evolution we know of proceeds from the vague to the definite.
—Charles Peirce

EACH local variable (§14.4) and every blank `final` (§4.5.4) 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, or the simple name qualified by `this`, of the variable occurring anywhere in an expression except as the left-hand operand of the simple assignment operator `=`. A Java compiler must carry out a specific conservative flow analysis to make sure that, for every access of a local variable or blank `final` field *f*, *f* is definitely assigned before the access; otherwise a compile-time error must occur.

Similarly, every blank `final` variable must be assigned at most once; it must be *definitely unassigned* when an assignment to it occurs. A Java compiler must carry out a specific conservative flow analysis to make sure that, for every assignment to a blank `final` variable, the variable is definitely unassigned before the assignment; otherwise a compile-time error must occur.

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.

For example, a Java compiler recognizes that `k` is definitely assigned before its access (as an argument of a method invocation) in the code:

```
{
    int k;
```

```
        if (v > 0 && (k = System.in.read()) >= 0)
            System.out.println(k);
    }
```

because the access occurs only if the value of the expression:

```
v > 0 && (k = System.in.read()) >= 0
```

is true, and the value can be `true` only if the assignment to `k` is executed (more properly, evaluated).

Similarly, a Java compiler will recognize that in the code:

```
{
    int k;
    while (true) {
        k = n;
        if (k >= 5) break;
        n = 6;
    }
    System.out.println(k);
}
```

the variable `k` is definitely assigned by the `while` statement because the condition expression `true` never has the value `false`, so only the `break` statement can cause the `while` statement to complete normally, and `k` is definitely assigned before the `break` statement.

On the other hand, the code

```
{
    int k;
    while (n < 4) {
        k = n;
        if (k >= 5) break;
        n = 6;
    }
    System.out.println(k);// k is not "definitely assigned" before this
}
```

must be rejected by a Java compiler, because in this case the `while` statement is not guaranteed to execute its body as far as the rules of definite assignment are concerned.

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. For example, a Java compiler must produce a compile-time error for the code:

```
{
    int k;
    int n = 5;
    if (n > 2)
```

```
        k = 3;
    System.out.println(k);// k is not "definitely assigned" before this
}
```

even though the value of `n` is known at compile time, and in principle it can be known at compile time that the assignment to `k` will always be executed (more properly, evaluated). A Java compiler must operate according to the rules laid out in this section. The rules recognize only constant expressions; in this example, the expression `n > 2` is not a constant expression as defined in §15.28.

As another example, a Java compiler will accept the code:

```
void flow(boolean flag) {
    int k;
    if (flag)
        k = 3;
    else
        k = 4;
    System.out.println(k);
}
```

as far as definite assignment of `k` is concerned, because the rules outlined in this section allow it to tell that `k` is assigned no matter whether the flag is `true` or `false`. But the rules do not accept the variation:

```
void flow(boolean flag) {
    int k;
    if (flag)
        k = 3;
    if (!flag)
        k = 4;
    System.out.println(k); // k is not "definitely assigned" before here
}
```

and so compiling this program must cause a compile-time error to occur.

A related example illustrates rules of definite unassignment. A Java compiler will accept the code:

```
void unflow(boolean flag) {
    final int k;
    if (flag) {
        k = 3;
        System.out.println(k);
    }
    else {
        k = 4;
        System.out.println(k);
    }
}
```

as far as definite unassignment of `k` is concerned, because the rules outlined in this section allow it to tell that `k` is assigned at most once (indeed, exactly once) no matter whether the flag is `true` or `false`. But the rules do not accept the variation:

```
void unflow(boolean flag) {
    final int k;
    if (flag) {
        k = 3;
        System.out.println(k);
    }
    if (!flag) {
        k = 4;          // k is not "definitely unassigned" before here
        System.out.println(k);
    }
}
```

and so compiling this program must cause a compile-time error to occur.

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. For example, the local variable k is definitely assigned a value after evaluation of the expression

```
a && ((k=m) > 5)
```

when the expression is `true` but not when the expression is `false` (because if `a` is `false`, then the assignment to `k` is not necessarily executed (more properly, evaluated)).

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. A peculiar consequence of this definition is that "*V* is definitely assigned after `break;`" is always true! Because a

`break` statement never completes normally, it is vacuously true that *V* has been assigned a value if the `break` statement completes normally.

Similarly, 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". An even more peculiar consequence of this definition is that "*V* is definitely unassigned after `break;`" is always true! Because a `break` statement never completes normally, it is vacuously true that *V* has not been assigned a value if the `break` statement completes normally. (For that matter, it is also vacuously true that the moon is made of green cheese if the `break` statement 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.)

To shorten the rules, the customary abbreviation "iff" is used to mean "if and only if". We also use an abbreviation convention: if a rule contains one or more occurrences of "[un]assigned" then it stands for two rules, one with every occurrence of "[un]assigned" replaced by "definitely assigned" and one with every occurrence of "[un]assigned" replaced by "definitely unassigned". For example:

- *V* is [un]assigned after an empty statement iff it is [un]assigned before the empty statement.

should be understood to stand for two rules:

- *V* is definitely assigned after an empty statement iff it is definitely assigned before the empty statement.
- *V* is definitely unassigned after an empty statement iff it is definitely unassigned before the empty statement.

The definite unassignment analysis of loop statements raises a special problem. Consider the statement `while (`*e*`)` *S*. In order to determine whether *V* is definitely unassigned within some subexpression of *e*, we need to determine whether *V* is definitely unassigned before *e*. One might argue, by analogy with the rule for

definite assignment (§16.2.9), that *V* is definitely unassigned before *e* iff it is definitely unassigned before the `while` statement. However, such a rule is inadequate for our purposes. If *e* evaluates to true, the statement *S* will be executed. Later, if *V* is assigned by *S* , then in the following iteration(s) *V* will have already been assigned when *e* is evaluated. Under the rule suggested above, it would be possible to assign *V* multiple times, which is exactly what we have sought to avoid by introducing these rules.

A revised rule would be: "*V* is definitely unassigned before *e* iff it is definitely unassigned before the while statement and definitely unassigned after *S*". However, when we formulate the rule for *S*, we find: "*V* is definitely unassigned before *S* iff it is definitely unassigned after *e* when true". This leads to a circularity. In effect, *V* is definitely unassigned *before* the loop condition *e* only if it is unassigned *after* the loop as a whole!

We break this vicious circle using a *hypothetical* analysis of the loop condition and body. For example, if we assume that *V* is definitely unassigned before *e* (regardless of whether *V* really is definitely unassigned before *e*), and can then prove that *V* was definitely unassigned after *e* then we know that *e* does not assign *V*. This is stated more formally as :

Assuming *V* is definitely unassigned before *e*, *V* is definitely unassigned after *e*.

Variations on the above analysis are used to define well founded definite unassignment rules for all loop statements in the language.

Throughout the rest of this chapter, we will, unless explicitly stated otherwise, write *V* to represent a local variable or a blank `final` field (for rules of definite assignment) or a blank `final` variable (for rules of definite unassignment). Likewise, we will use *a*, *b*, *c*, and *e* to represent expressions, and *S* and *T* to represent statements.

16.1 Definite Assignment and Expressions

Driftwood: The party of the first part shall be known in this contract as the party of the first part.

—Groucho Marx, *A Night at the Opera* (1935)

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.

Because a constant expression whose value is `true` never has the value `false`, and a constant expression whose value is `false` never has the value `true`, the two preceding rules are vacuously satisfied. They are helpful in analyzing expressions involving the operators && (§16.1.2), || (§16.1.3), ! (§16.1.4), and ? : (§16.1.5).

- *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. (This is equivalent to saying that *V* is [un]assigned after *e* iff *V* is [un]assigned before *e*.)

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* || *b* iff *V* is [un]assigned after *a* || *b* when true and *V* is [un]assigned after *a* || *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. (This is equivalent to saying that *V* is [un]assigned after !*a* iff *V* is [un]assigned after *a*.)

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 Assignment Expressions

Driftwood: *Would you like to hear it once more?*
Fiorello: *Just the first part.*
Driftwood: *What do you mean? The party of the first part?*
Fiorello: *No, the first part of the party of the first part.*

—Groucho Marx and Chico Marx,
A Night at the Opera (1935)

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*.

Note that if *a* is *V* and *V* is not definitely assigned before a compound assignment such as *a* &= *b*, then a compile-time error will necessarily occur. The first rule for definite assignment stated above includes the disjunct "*a* is *V*" even for compound assignment expressions, not just simple assignments, so that *V* will be considered to have been definitely assigned at later points in the code. Including the disjunct "*a* is *V*" does not affect the binary decision as to whether a program is acceptable or will result in a compile-time error, but it affects *how many* different points in the code may be regarded as erroneous, and so in practice it can improve the quality of error reporting. A similar remark applies to the inclusion of the conjunct "*a* is not *V*" in the first rule for definite unassignment stated above.

If the assignment expression is boolean-valued, then these rules also apply:

- *V* is [un]assigned after the assignment expression when true iff *V* is [un]assigned after the assignment expression .
- *V* is [un]assigned after the assignment expression when false iff *V* is [un]assigned after the assignment expression .

16.1.8 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*--.

16.1.9 Other Expressions

> *Driftwood: All right. It says the, uh, the first part of the party of the first part, should be known in this contract as the first part of the party of the first part, should be known in this contract . . .*
>
> —Groucho Marx, *A Night at the Opera* (1935)

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.

There is a piece of subtle reasoning behind the assertion that a variable *V* can be known to be definitely unassigned after a method invocation. Taken by itself, at face value and without qualification, such an assertion is not always true, because an invoked method can perform assignments. But it must be remembered that, for the purposes of the Java programming language, the concept of definite unassignment is applied *only* to blank `final` variables. If *V* is a blank `final` local variable, then only the method to which its declaration belongs can perform assignments to *V*. If *V* is a blank `final` field, then only a constructor or an initializer for the class containing the declaration for *V* can perform assignments to *V*; no method can perform assignments to *V*. Finally, explicit constructor invocations (§8.8.5) are handled specially (§16.6); although they are syntactically similar to expression statements containing method invocations, they are not expression statements and therefore the rules of this section do not apply to explicit constructor invocations.

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

Driftwood: The party of the second part shall be known in this contract as the party of the second part.

—Groucho Marx, *A Night at the Opera* (1935)

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*.
- 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*.
- *V* is [un]assigned after an empty block iff it is [un]assigned before the empty block.
- *V* is [un]assigned after a nonempty block iff it is [un]assigned after the last statement in the block.
- *V* is [un]assigned before the first statement of the block iff it is [un]assigned before the block.
- *V* is [un]assigned before any other statement *S* of the block iff it is [un]assigned after the statement immediately preceding *S* in the block.

16.2.3 Local Class Declaration Statements

- *V* is [un]assigned after a local class declaration statement iff it 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 it 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 it 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 it 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 iff it 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 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 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 last block-statement of the preceding block-statement-group.
- *V* is [un]assigned before any block-statement other than the first of any block-statement-group in the switch block iff *V* is [un]assigned after the last block-statement of the preceding block-statement-group.

16.2.9 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.

16.2.10 do Statements

- *V* is [un]assigned after `do` *S* `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.

16.2.11 for Statements

- *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.

16.2.11.1 *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 *E* other than the first in the initialization part iff *V* is [un]assigned after the expression statement immediately preceding *E*.

16.2.11.2 *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 *E* other than the first in the incrementation part iff *V* is [un]assigned after the expression statement immediately preceding *E*.

16.2.12 `break`, `continue`, `return`, and `throw` Statements

> *Fiorello:* *Hey, look! Why can't the first part of the second party be the second part of the first party? Then you've got something!*
>
> —Chico Marx, *A Night at the Opera* (1935)

- 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.

16.2.13 `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*.

16.2.14 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 *V* is definitely unassigned after the try block and *V* is definitely unassigned before every return statement that belongs to the try block, every throw statement that belongs to the try block, every break statement that belongs to the try block and whose break target contains (or is) the try statement, and 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 *V* is definitely unassigned after the try block and *V* is definitely unassigned before every return statement that belongs to the try block, every throw statement that belongs to the try block, every break statement that belongs to the try block and whose break target contains (or is) the try statement, and every continue statement that belongs to the try block and whose continue target contains the try statement and *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 it is [un]assigned before the empty array initializer.
- *V* is [un]assigned after a nonempty array initializer iff it 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 it is [un]assigned before the array initializer.
- *V* is [un]assigned before any other variable initializer *I* of the array initializer iff it is [un]assigned after the variable initializer to the left of *I* in the array initializer.

16.5 Definite Assignment and Static Initializers

Let *C* be a class, and let *V* be a blank `final static` member field of *C*, declared in *C*. Then:

- *V* is definitely unassigned (and moreover is not definitely assigned) before the leftmost `static` initializer or `static` variable initializer of *C*.
- *V* is [un]assigned before a `static` initializer or `static` variable initializer of *C* other than the leftmost iff *V* is [un]assigned after the preceding `static` initializer or `static` variable initializer of *C*.

Let *C* be a class, and let *V* be a blank `final static` 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 static initializer of *C*.
- *V* is definitely assigned (and moreover is not definitely unassigned) before every static variable initializer of *C*.

16.6 Definite Assignment, Constructor Invocations, and Instance Variable Initializers

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.
- *V* is definitely unassigned (and moreover is not definitely assigned) before an explicit or implicit superclass constructor invocation.
- 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*.

CHAPTER 17

Threads and Locks

And oft-times in the most forbidding den
Of solitude, with love of science strong,
How patiently the yoke of thought they bear;
How subtly glide its finest threads along!
—William Wordsworth, *Monks and Schoolmen*,
in *Ecclesiastical Sonnets* (1822)

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.

The Java programming language supports the coding of programs that, though concurrent, still exhibit deterministic behavior, by providing mechanisms for *synchronizing* the concurrent activity of threads. To synchronize threads, the Java programming language uses *monitors*, which are a high-level mechanism for allowing only one thread at a time to execute a region of code protected by the monitor. The behavior of monitors is explained in terms of *locks*; there is a lock associated with each object.

The `synchronized` statement (§14.18) performs two special actions relevant only to multithreaded operation: (1) after computing a reference to an object but before executing its body, it *locks* a lock associated with the object, and (2) after execution of the body has completed, either normally or abruptly, it *unlocks* that same lock. As a convenience, a method may be declared `synchronized`; such a method behaves as if its body were contained in a `synchronized` statement.

The methods `wait` , `notify`, and `notifyAll` of class `Object` support an efficient transfer of control from one thread to another. Rather than simply "spinning" (repeatedly locking and unlocking an object to see whether some internal state has

changed), which consumes computational effort, a thread can suspend itself using `wait` until such time as another thread awakens it using `notify`. This is especially appropriate in situations where threads have a producer-consumer relationship (actively cooperating on a common goal) rather than a mutual exclusion relationship (trying to avoid conflicts while sharing a common resource).

As a thread executes code, it carries out a sequence of actions. A thread may *use* the value of a variable or *assign* it a new value. (Other actions include arithmetic operations, conditional tests, and method invocations, but these do not involve variables directly.) If two or more concurrent threads act on a shared variable, there is a possibility that the actions on the variable will produce timing-dependent results. This dependence on timing is inherent in concurrent programming, producing one of the few places in the language where the result of executing a program is not determined solely by this specification.

Each thread has a working memory, in which it may keep copies of the values of variables from the main memory that is shared between all threads. To access a shared variable, a thread usually first obtains a lock and flushes its working memory. This guarantees that shared values will thereafter be loaded from the shared main memory to the threads working memory. When a thread unlocks a lock it guarantees the values it holds in its working memory will be written back to the main memory.

This chapter explains the interaction of threads with the main memory, and thus with each other, in terms of certain low-level actions. There are rules about the order in which these actions may occur. These rules impose constraints on any implementation of the Java programming language, and a programmer may rely on the rules to predict the possible behaviors of a concurrent program. The rules do, however, intentionally give the implementor certain freedoms; the intent is to permit certain standard hardware and software techniques that can greatly improve the speed and efficiency of concurrent code.

Briefly put, these are the important consequences of the rules:

- Proper use of synchronization constructs will allow reliable transmission of values or sets of values from one thread to another through shared variables.
- When a thread uses the value of a variable, the value it obtains is in fact a value stored into the variable by that thread or by some other thread. This is true even if the program does not contain code for proper synchronization. For example, if two threads store references to different objects into the same reference value, the variable will subsequently contain a reference to one object or the other, not a reference to some other object or a corrupted reference value. (There is a special exception for `long` and `double` values; see §17.4.)

- In the absence of explicit synchronization, an implementation is free to update the main memory in an order that may be surprising. Therefore the programmer who prefers to avoid surprises should use explicit synchronization.

17.1 Terminology and Framework

A *variable* is any location within a program that may be stored into. This includes not only class variables and instance variables but also components of arrays. Variables are kept in a *main memory* that is shared by all threads. Because it is impossible for one thread to access parameters or local variables of another thread, it doesn't matter whether parameters and local variables are thought of as residing in the shared main memory or in the working memory of the thread that owns them.

Every thread has a *working memory* in which it keeps its own *working copy* of variables that it must use or assign. As the thread executes a program, it operates on these working copies. The main memory contains the *master copy* of every variable. There are rules about when a thread is permitted or required to transfer the contents of its working copy of a variable into the master copy or vice versa.

The main memory also contains *locks*; there is one lock associated with each object. Threads may compete to acquire a lock.

For the purposes of this chapter, the verbs *use*, *assign*, *load*, *store*, *lock*, and *unlock* name *actions* that a thread can perform. The verbs *read*, *write*, *lock*, and *unlock* name actions that the main memory subsystem can perform. Each of these actions is *atomic* (indivisible).

A *use* or *assign* action is a tightly coupled interaction between a thread's execution engine and the thread's working memory. A *lock* or *unlock* action is a tightly coupled interaction between a thread's execution engine and the main memory. But the transfer of data between the main memory and a thread's working memory is loosely coupled. When data is copied from the main memory to a working memory, two actions must occur: a *read* action performed by the main memory followed some time later by a corresponding *load* action performed by the working memory. When data is copied from a working memory to the main memory, two actions must occur: a *store* action performed by the working memory followed some time later by a corresponding *write* action performed by the main memory. There may be some transit time between main memory and a working memory, and the transit time may be different for each transaction; thus actions initiated by a thread on different variables may viewed by another thread as occurring in a different order. For each variable, however, the actions in main memory on behalf of any one thread are performed in the same order as the corresponding actions by that thread. (This is explained in greater detail below.)

A single thread issues a stream of *use*, *assign*, *lock*, and *unlock* actions as dictated by the semantics of the program it is executing. The underlying implementation is then required additionally to perform appropriate *load*, *store*, *read*, and *write* actions so as to obey a certain set of constraints, explained below. If the implementation correctly follows these rules and the application programmer follows certain other rules of programming, then data can be reliably transferred between threads through shared variables. The rules are designed to be "tight" enough to make this possible but "loose" enough to allow hardware and software designers considerable freedom to improve speed and throughput through such mechanisms as registers, queues, and caches.

Here are the detailed definitions of each of the actions:

- A *use* action (by a thread) transfers the contents of the thread's working copy of a variable to the thread's execution engine. This action is performed whenever a thread executes a virtual machine instruction that uses the value of a variable.
- An *assign* action (by a thread) transfers a value from the thread's execution engine into the thread's working copy of a variable. This action is performed whenever a thread executes a virtual machine instruction that assigns to a variable.
- A *read* action (by the main memory) transmits the contents of the master copy of a variable to a thread's working memory for use by a later *load* action.
- A *load* action (by a thread) puts a value transmitted from main memory by a *read* action into the thread's working copy of a variable.
- A *store* action (by a thread) transmits the contents of the thread's working copy of a variable to main memory for use by a later *write* action.
- A *write* action (by the main memory) puts a value transmitted from the thread's working memory by a *store* action into the master copy of a variable in main memory.
- A *lock* action (by a thread tightly synchronized with main memory) causes a thread to acquire one claim on a particular lock.
- An *unlock* action (by a thread tightly synchronized with main memory) causes a thread to release one claim on a particular lock.

Thus the interaction of a thread with a variable over time consists of a sequence of *use*, *assign*, *load*, and *store* actions. Main memory performs a *read* action for every *load* and a *write* action for every *store*. A thread's interactions

with a lock over time consists of a sequence of *lock* and *unlock* actions. All the globally visible behavior of a thread thus comprises all the thread's actions on variables and locks.

17.2 Execution Order

The rules of execution order constrain the order in which certain events may occur. There are four general constraints on the relationships among actions:

- The actions performed by any one thread are totally ordered; that is, for any two actions performed by a thread, one action precedes the other.
- The actions performed by the main memory for any one variable are totally ordered; that is, for any two actions performed by the main memory on the same variable, one action precedes the other.
- The actions performed by the main memory for any one lock are totally ordered; that is, for any two actions performed by the main memory on the same lock, one action precedes the other.
- It is not permitted for an action to follow itself.

The last rule may seem trivial, but it does need to be stated separately and explicitly for completeness. Without it, it would be possible to propose a set of actions by two or more threads and precedence relationships among the actions that would satisfy all the other rules but would require an action to follow itself.

Threads do not interact directly; they communicate only through the shared main memory. The relationships between the actions of a thread and the actions of main memory are constrained in three ways:

- Each *lock* or *unlock* action is performed jointly by some thread and the main memory.
- Each *load* action by a thread is uniquely paired with a *read* action by the main memory such that the *load* action follows the *read* action.
- Each *store* action by a thread is uniquely paired with a *write* action by the main memory such that the *write* action follows the *store* action.

Most of the rules in the following sections further constrain the order in which certain actions take place. A rule may state that one action must precede or follow some other action. Note that this relationship is transitive: if action A must precede action B, and B must precede C, then A must precede C. The programmer must

remember that these rules are the *only* constraints on the ordering of actions; if no rule or combination of rules implies that action *A* must precede action *B*, then an implementation is free to perform action *B* before action *A*, or to perform action *B* concurrently with action *A*. This freedom can be the key to good performance. Conversely, an implementation is not required to take advantage of all the freedoms given it.

In the rules that follow, the phrasing "*B* must intervene between *A* and *C*" means that action *B* must follow action *A* and precede action *C*.

17.3 Rules about Variables

Let *T* be a thread and *V* be a variable. There are certain constraints on the actions performed by *T* with respect to *V*:

- An *use* or *assign* by *T* of *V* is permitted only when dictated by execution by *T* of the program according to the Java programming language's execution model. For example, an occurrence of *V* as an operand of the + operator requires that a single *use* action occur on *V*; an occurrence of *V* as the left-hand operand of the assignment operator = requires that a single *assign* action occur. All *use* and *assign* actions by a given thread must occur in the order specified by the program being executed by the thread. If the following rules forbid *T* to perform a required *use* as its next action, it may be necessary for *T* to perform a *load* first in order to make progress.
- A *store* action by *T* on *V* must intervene between an *assign* by *T* of *V* and a subsequent *load* by *T* of *V*. (Less formally: a thread is not permitted to lose its most recent assign.)
- An *assign* action by *T* on *V* must intervene between a *load* or *store* by *T* of *V* and a subsequent *store* by *T* of *V*. (Less formally: a thread is not permitted to write data from its working memory back to main memory for no reason.)
- After a thread is created, it must perform an *assign* or *load* action on a variable before performing a *use* or *store* action on that variable. (Less formally: a new thread starts with an empty working memory.)
- After a variable is created, every thread must perform an *assign* or *load* action on that variable before performing a *use* or *store* action on that variable. (Less formally: a new variable is created only in main memory and is not initially in any thread's working memory.)

Provided that all the constraints above and below are obeyed, a *load* or *store* action may be issued at any time by any thread on any variable, at the whim of the implementation.

There are also certain constraints on the *read* and *write* actions performed by main memory:

- For every *load* action performed by any thread *T* on its working copy of a variable *V*, there must be a corresponding preceding *read* action by the main memory on the master copy of *V*, and the *load* action must put into the working copy the data transmitted by the corresponding *read* action.
- For every *store* action performed by any thread *T* on its working copy of a variable *V*, there must be a corresponding following *write* action by the main memory on the master copy of *V*, and the *write* action must put into the master copy the data transmitted by the corresponding *store* action.
- Let action *A* be a *load* or *store* by thread *T* on variable *V*, and let action *P* be the corresponding *read* or *write* by the main memory on variable *V*. Similarly, let action *B* be some other *load* or *store* by thread *T* on that same variable *V*, and let action *Q* be the corresponding *read* or *write* by the main memory on variable *V*. If *A* precedes *B*, then *P* must precede *Q*. (Less formally: actions on the master copy of any given variable on behalf of a thread are performed by the main memory in exactly the order that the thread requested.)

Note that this last rule applies *only* to actions by a thread on the *same* variable. However, there is a more stringent rule for `volatile` variables (§17.7).

17.4 Nonatomic Treatment of `double` and `long`

If a `double` or `long` variable is not declared `volatile`, then for the purposes of *load*, *store*, *read*, and *write* actions they are treated as if they were two variables of 32 bits each: wherever the rules require one of these actions, two such actions are performed, one for each 32-bit half. The manner in which the 64 bits of a `double` or `long` variable are encoded into two 32-bit quantities is implementation-dependent.

This matters only because a *read* or *write* of a `double` or `long` variable may be handled by an actual main memory as two 32-bit *read* or *write* actions that may be separated in time, with other actions coming between them. Consequently, if two threads concurrently assign distinct values to the same shared non-`volatile` `double` or `long` variable, a subsequent use of that variable may obtain a value that is not equal to either of the assigned values, but some implementation-dependent mixture of the two values.

An implementation is free to implement *load*, *store*, *read*, and *write* actions for `double` and `long` values as atomic 64-bit actions; in fact, this is strongly encouraged. The model divides them into 32-bit halves for the sake of several currently popular microprocessors that fail to provide efficient atomic memory transactions on 64-bit quantities. It would have been simpler to define all memory transactions on single variables as atomic; this more complex definition is a pragmatic concession to current hardware practice. In the future this concession may be eliminated. Meanwhile, programmers are cautioned always to explicitly synchronize access to shared `double` and `long` variables.

17.5 Rules about Locks

By the pricking of my thumbs,
Something wicked this way comes.
Open, locks,
Whoever knocks!

—William Shakespeare, *Macbeth*, Act IV, scene i

Let *T* be a thread and *L* be a lock. There are certain constraints on the actions performed by *T* with respect to *L*:

- A *lock* action by *T* on *L* may occur only if, for every thread *S* other than *T*, the number of preceding *unlock* actions by *S* on *L* equals the number of preceding *lock* actions by *S* on *L*. (Less formally: only one thread at a time is permitted to lay claim to a lock, and moreover a thread may acquire the same lock multiple times and doesn't relinquish ownership of it until a matching number of *unlock* actions have been performed.)
- An *unlock* action by thread *T* on lock *L* may occur only if the number of preceding *unlock* actions by *T* on *L* is strictly less than the number of preceding *lock* actions by *T* on *L*. (Less formally: a thread is not permitted to unlock a lock it doesn't own.)

With respect to a lock, the *lock* and *unlock* actions performed by all the threads are performed in some total sequential order. This total order must be consistent with the total order on the actions of each thread.

17.6 Rules about the Interaction of Locks and Variables

Let *T* be any thread, let *V* be any variable, and let *L* be any lock. There are certain constraints on the actions performed by *T* with respect to *V* and *L*:

- Between an *assign* action by *T* on *V* and a subsequent *unlock* action by *T* on *L*, a *store* action by *T* on *V* must intervene; moreover, the *write* action corresponding to that *store* must precede the *unlock* action, as seen by main memory. (Less formally: if a thread is to perform an *unlock* action on *any* lock, it must first copy *all* assigned values in its working memory back out to main memory.)
- Between a *lock* action by *T* on *L* and a subsequent *use* or *store* action by *T* on a variable *V*, an *assign* or *load* action on *V* must intervene; moreover, if it is a *load* action, then the *read* action corresponding to that *load* must follow the *lock* action, as seen by main memory. (Less formally: a *lock* action acts as if it flushes *all* variables from the thread's working memory; before use they must be assigned or loaded from main memory.)

17.7 Rules for Volatile Variables

If a variable is declared `volatile`, then additional constraints apply to the actions of each thread. Let *T* be a thread and let *V* and *W* be volatile variables.

- A *use* action by *T* on *V* is permitted only if the previous action by *T* on *V* was *load*, and a *load* action by *T* on *V* is permitted only if the next action by *T* on *V* is *use*. The *use* action is said to be "associated" with the *read* action that corresponds to the *load*.
- A *store* action by *T* on *V* is permitted only if the previous action by *T* on *V* was *assign*, and an *assign* action by *T* on *V* is permitted only if the next action by *T* on *V* is *store*. The *assign* action is said to be "associated" with the *write* action that corresponds to the *store*.
- Let action *A* be a *use* or *assign* by thread *T* on variable *V*, let action *F* be the *load* or *store* associated with *A*, and let action *P* be the *read* or *write* of *V* that corresponds to *F*. Similarly, let action *B* be a *use* or *assign* by thread *T* on variable *W*, let action *G* be the *load* or *store* associated with *B*, and let action *Q* be the *read* or *write* of *W* that corresponds to *G*. If *A* precedes *B*, then *P* must precede *Q*. (Less formally: actions on the master copies of volatile variables on behalf of a thread are performed by the main memory in exactly the order that the thread requested.)

17.8 Prescient Store Actions

If a variable is not declared `volatile`, then the rules in the previous sections are relaxed slightly to allow *store* actions to occur earlier than would otherwise be permitted. The purpose of this relaxation is to allow optimizing Java compilers to perform certain kinds of code rearrangement that preserve the semantics of properly synchronized programs but might be caught in the act of performing memory actions out of order by programs that are not properly synchronized.

Suppose that a *store* by *T* of *V* would follow a particular *assign* by *T* of *V* according to the rules of the previous sections, with no intervening *load* or *assign* by *T* of *V*. Then that *store* action would send to the main memory the value that the *assign* action put into the working memory of thread *T*. The special rule allows the *store* action to instead occur before the *assign* action, if the following restrictions are obeyed:

- If the *store* action occurs, the *assign* is bound to occur. (Remember, these are restrictions on what actually happens, not on what a thread plans to do. No fair performing a *store* and then throwing an exception before the *assign* occurs!)
- No *lock* action intervenes between the relocated *store* and the *assign*.
- No *load* of *V* intervenes between the relocated *store* and the *assign*.
- No other *store* of *V* intervenes between the relocated *store* and the *assign*.
- The *store* action sends to the main memory the value that the *assign* action will put into the working memory of thread *T*.

This last property inspires us to call such an early *store* action *prescient*: it has to know ahead of time, somehow, what value will be stored by the *assign* that it should have followed. In practice, optimized compiled code will compute such values early (which is permitted if, for example, the computation has no side effects and throws no exceptions), store them early (before entering a loop, for example), and keep them in working registers for later use within the loop.

17.9 Discussion

Any association between locks and variables is purely conventional. Locking any lock conceptually flushes *all* variables from a thread's working memory, and unlocking any lock forces the writing out to main memory of *all* variables that the thread has assigned. That a lock may be associated with a particular object or a class is purely a convention. In some applications, it may be appropriate always to

lock an object before accessing any of its instance variables, for example; synchronized methods are a convenient way to follow this convention. In other applications, it may suffice to use a single lock to synchronize access to a large collection of objects.

If a thread uses a particular shared variable only after locking a particular lock and before the corresponding unlocking of that same lock, then the thread will read the shared value of that variable from main memory after the *lock* action, if necessary, and will copy back to main memory the value most recently assigned to that variable before the *unlock* action. This, in conjunction with the mutual exclusion rules for locks, suffices to guarantee that values are correctly transmitted from one thread to another through shared variables.

The rules for `volatile` variables effectively require that main memory be touched exactly once for each *use* or *assign* of a `volatile` variable by a thread, and that main memory be touched in exactly the order dictated by the thread execution semantics. However, such memory actions are not ordered with respect to *read* and *write* actions on nonvolatile variables.

17.10 Example: Possible Swap

Consider a class that has class variables `a` and `b` and methods `hither` and `yon`:

```
class Sample {
   int a = 1, b = 2;
   void hither() {
      a = b;
   }
   void yon() {
      b = a;
   }
}
```

Now suppose that two threads are created, and that one thread calls `hither` while the other thread calls `yon`. What is the required set of actions and what are the ordering constraints?

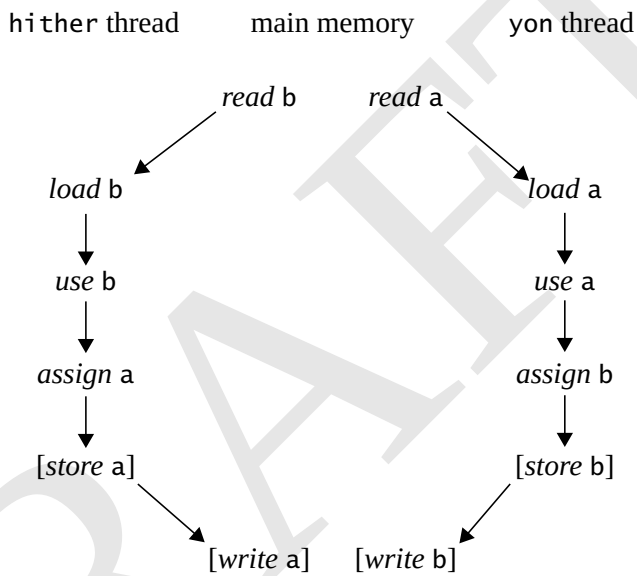
Let us consider the thread that calls `hither`. According to the rules, this thread must perform an *use* of `b` followed by an *assign* of `a`. That is the bare minimum required to execute a call to the method `hither`.

Now, the first action on variable `b` by the thread cannot be *use*. But it may be *assign* or *load*. An *assign* to `b` cannot occur because the program text does not call for such an *assign* action, so a *load* of `b` is required. This *load* action by the thread in turn requires a preceding *read* action for `b` by the main memory.

The thread may optionally *store* the value of `a` after the *assign* has occurred. If it does, then the *store* action in turn requires a following *write* action for `a` by the main memory.

The situation for the thread that calls `yon` is similar, but with the roles of `a` and `b` exchanged.

The total set of actions may be pictured as follows:



Here an arrow from action *A* to action *B* indicates that *A* must precede *B*.

In what order may the actions by the main memory occur? The only constraint is that it is not possible both for the *write* of `a` to precede the *read* of `a` and for the *write* of `b` to precede the *read* of `b`, because the causality arrows in the diagram would form a loop so that an action would have to precede itself, which is not allowed. Assuming that the optional *store* and *write* actions are to occur, there are three possible orderings in which the main memory might legitimately perform its actions. Let `ha` and `hb` be the working copies of `a` and `b` for the `hither` thread, let `ya` and `yb` be the working copies for the `yon` thread, and let `ma` and `mb` be the master copies in main memory. Initially `ma=1` and `mb=2`. Then the three possible orderings of actions and the resulting states are as follows:

- *write* `a`→*read* `a`, *read* `b`→*write* `b` (then `ha=2`, `hb=2`, `ma=2`, `mb=2`, `ya=2`, `yb=2`)
- *read* `a`→*write* `a`, *write* `b`→*read* `b` (then `ha=1`, `hb=1`, `ma=1`, `mb=1`, `ya=1`, `yb=1`)

- *read* a→*write* a, *read* b→*write* b (then ha=2, hb=2, ma=2, mb=1, ya=1, yb=1)

Thus the net result might be that, in main memory, b is copied into a, a is copied into b, or the values of a and b are swapped; moreover, the working copies of the variables might or might not agree. It would be incorrect, of course, to assume that any one of these outcomes is more likely than another. This is one place in which the behavior of a program is necessarily timing-dependent.

Of course, an implementation might also choose not to perform the *store* and *write* actions, or only one of the two pairs, leading to yet other possible results.

Now suppose that we modify the example to use synchronized methods:

```
class SynchSample {
   int a = 1, b = 2;
   synchronized void hither() {
      a = b;
   }
   synchronized void yon() {
      b = a;
   }
}
```

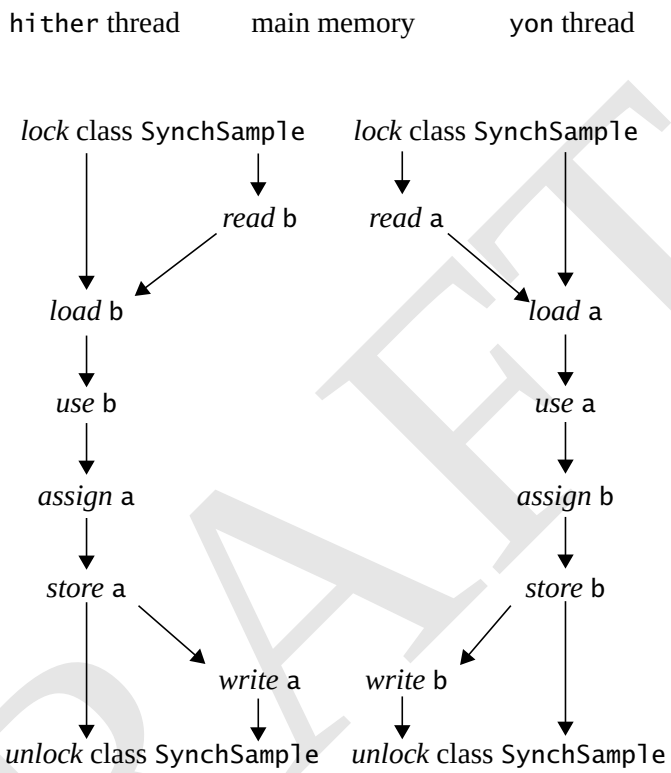
Let us again consider the thread that calls hither. According to the rules, this thread must perform a *lock* action (on the instance of class SynchSample on which the hither method is being called) before the body of method hither is executed. This is followed by a *use* of b and then an *assign* of a. Finally, an *unlock* action on that same instance of SynchSample must be performed after the body of method hither completes. That is the bare minimum required to execute a call to the method hither.

As before, a *load* of b is required, which in turn requires a preceding *read* action for b by the main memory. Because the *load* follows the *lock* action, the corresponding *read* must also follow the *lock* action.

Because an *unlock* action follows the *assign* of a, a *store* action on a is mandatory, which in turn requires a following *write* action for a by the main memory. The *write* must precede the *unlock* action.

The situation for the thread that calls yon is similar, but with the roles of a and b exchanged.

The total set of actions may be pictured as follows:



The *lock* and *unlock* actions provide further constraints on the order of actions by the main memory; the *lock* action by one thread cannot occur between the *lock* and *unlock* actions of the other thread. Moreover, the *unlock* actions require that the *store* and *write* actions occur. It follows that only two sequences are possible:

- *write* a→*read* a, *read* b→*write* b (then ha=2, hb=2, ma=2, mb=2, ya=2, yb=2)
- *read* a→*write* a, *write* b→*read* b (then ha=1, hb=1, ma=1, mb=1, ya=1, yb=1)

While the resulting state is timing-dependent, it can be seen that the two threads will necessarily agree on the values of a and b.

17.11 Example: Out-of-Order Writes

This example is similar to that in the preceding section, except that one method assigns to both variables and the other method reads both variables. Consider a class that has class variables `a` and `b` and methods `to` and `fro`:

```
class Simple {
   int a = 1, b = 2;
   void to() {
      a = 3;
      b = 4;
   }
   void fro() {
      System.out.println("a= " + a + ", b=" + b);
   }
}
```

Now suppose that two threads are created, and that one thread calls `to` while the other thread calls `fro`. What is the required set of actions and what are the ordering constraints?

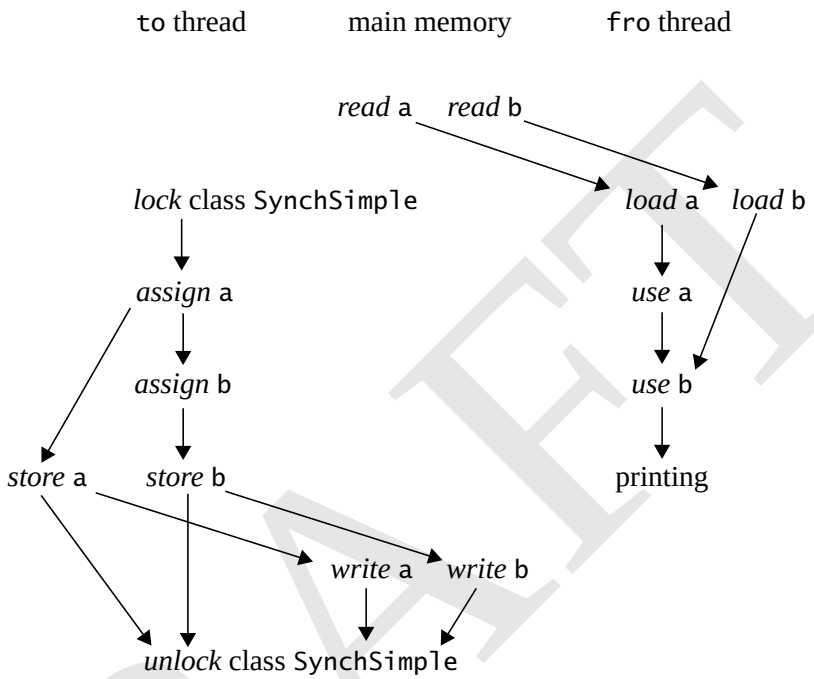
Let us consider the thread that calls `to`. According to the rules, this thread must perform an *assign* of `a` followed by an *assign* of `b`. That is the bare minimum required to execute a call to the method `to`. Because there is no synchronization, it is at the option of the implementation whether or not to *store* the assigned values back to main memory! Therefore the thread that calls `fro` may obtain either `1` or `3` for the value of `a`, and independently may obtain either `2` or `4` for the value of `b`.

Now suppose that `to` is `synchronized` but `fro` is not:

```
class SynchSimple {
   int a = 1, b = 2;
   synchronized void to() {
      a = 3;
      b = 4;
   }
   void fro() {
      System.out.println("a= " + a + ", b=" + b);
   }
}
```

In this case the method `to` will be forced to *store* the assigned values back to main memory before the *unlock* action at the end of the method. The method `fro` must, of course, *use* `a` and `b` (in that order) and so must *load* values for `a` and `b` from main memory.

The total set of actions may be pictured as follows:



Here an arrow from action *A* to action *B* indicates that *A* must precede *B*.

In what order may the actions by the main memory occur? Note that the rules do not require that *write* `a` occur before *write* `b`; neither do they require that *read* `a` occur before *read* `b`. Also, even though method `to` is `synchronized`, method `fro` is not `synchronized`, so there is nothing to prevent the *read* actions from occurring between the *lock* and *unlock* actions. (The point is that declaring one method `synchronized` does not of itself make that method behave as if it were atomic.)

As a result, the method `fro` could still obtain either `1` or `3` for the value of `a`, and independently could obtain either `2` or `4` for the value of `b`. In particular, `fro` might observe the value `1` for `a` and `4` for `b`. Thus, even though `to` does an *assign* to `a` and then an *assign* to `b`, the *write* actions to main memory may be observed by another thread to occur as if in the opposite order.

Finally, suppose that `to` and `fro` are both `synchronized`:

```
class SynchSynchSimple {
   int a = 1, b = 2;
   synchronized void to() {
      a = 3;
      b = 4;
   }
   synchronized void fro() {
      System.out.println("a= " + a + ", b=" + b);
   }
}
```

In this case, the actions of method `fro` cannot be interleaved with the actions of method `to`, and so `fro` will print either "`a=1, b=2`" or "`a=3, b=4`".

17.12 Threads

They plant dead trees for living, and the dead
They string together with a living thread . . .
But in no hush they string it . . . With a laugh, . . .
They bring the telephone and telegraph.

—Robert Frost, *The Line-gang* (1920)

Threads are created and managed by the built-in classes `Thread` and `ThreadGroup`. Creating a `Thread` object creates a thread and that is the only way to create a thread. When the thread is created, it is not yet active; it begins to run when its `start` method is called.

Every thread has a *priority*. When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion.

17.13 Locks and Synchronization

There is a lock associated with every object. The Java programming language does not provide a way to perform separate *lock* and *unlock* actions; instead, they are implicitly performed by high-level constructs that arrange always to pair such actions correctly. (We note, however, that the Java Virtual Machine provides separate *monitorenter* and *monitorexit* instructions that implement the *lock* and *unlock* actions.)

The `synchronized` statement (§14.18) computes a reference to an object; it then attempts to perform a *lock* action on that object and does not proceed further until the *lock* action has successfully completed. (A *lock* action may be delayed because the rules about locks can prevent the main memory from participating until some other thread is ready to perform one or more *unlock* actions.) 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 lock.

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 lock 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 lock 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 lock.

Best practice is that if a variable is ever to be assigned by one thread and used or assigned by another, then all accesses to that variable should be enclosed in `synchronized` methods or `synchronized` statements.

The Java programming language does not prevent, nor require 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 don't deadlock, if necessary.

17.14 Wait Sets and Notification

Every object, in addition to having an associated lock, has an associated *wait set*, which is a set of threads. When an object is first created, its wait set is empty.

Wait sets are used by the methods `wait`, `notify` , and `notifyAll` of class `Object`. These methods also interact with the scheduling mechanism for threads.

The method `wait` should be called for an object only when the current thread (call it *T*) has already locked the object's lock. Suppose that thread *T* has in fact performed *N lock* actions that have not been matched by *unlock* actions. The `wait` method then adds the current thread to the wait set for the object, disables the current thread for thread scheduling purposes, and performs *N unlock* actions to relinquish the lock. The thread *T* then lies dormant until one of three things happens:

- Some other thread invokes the `notify` method for that object and thread *T* happens to be the one arbitrarily chosen as the one to notify.
- Some other thread invokes the `notifyAll` method for that object.
- If the call by thread *T* to the `wait` method specified a timeout interval, the specified amount of real time has elapsed.

The thread *T* is then removed from the wait set and re-enabled for thread scheduling. It then locks the object again (which may involve competing in the usual manner with other threads); once it has gained control of the lock, it performs $N-1$ additional *lock* actions and then returns from the invocation of the `wait` method. Thus, on return from the `wait` method, the state of the object's lock is exactly as it was when the `wait` method was invoked.

The `notify` method should be called for an object only when the current thread has already locked the object's lock. If the wait set for the object is not empty, then some arbitrarily chosen thread is removed from the wait set and re-enabled for thread scheduling. (Of course, that thread will not be able to proceed until the current thread relinquishes the object's lock.)

The `notifyAll` method should be called for an object only when the current thread has already locked the object's lock. Every thread in the wait set for the object is removed from the wait set and re-enabled for thread scheduling. (Of course, those threads will not be able to proceed until the current thread relinquishes the object's lock.)

These pearls of thought in Persian gulfs were bred,
Each softly lucent as a rounded moon;
The diver Omar plucked them from their bed,
Fitzgerald strung them on an English thread.

—James Russell Lowell,
in a copy of Omar Khayyam

CHAPTER 18

Syntax

Is there grammar in a title. There is grammar in a title. Thank you.
—Gertrude Stein, *Arthur a Grammar,* in *How to Write* (1931)

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 ideally suited as a basis for a parser. The grammar presented in this chapter is the basis for the reference implementation.

The grammar below uses the following BNF-style conventions:

- *[x]* denotes zero or one occurences of *x*.
- *{x}* denotes zero or more occurences of *x*.
- *x* | *y* means one of either *x* or *y.*

18.1 The Grammar of the Java Programming Language

Ident:

IDENTIFIER

Qualident:

Ident { . Ident }

Literal:

IntegerLiteral

FloatingPointLiteral

CharacterLiteral

StringLiteral

BooleanLiteral

NullLiteral

Expression:

Expression1 [ExpressionRest]

ExpressionRest:

[AssignmentOperator Expression1]

AssignmentOperator:

=

+=

-=

*=

/=

&=

|=

^=

%=

<<=

>>=

>>>=

Type:

Ident { . Ident } BracketsOpt

BasicType

StatementExpression:

Expression

ConstantExpression:

Expression

Expression1:

Expression2 [Expression1Rest]

Expression1Rest:

[`?` *Expression* `:` *Expression1]*

Expression2 :

Expression3 [Expression2Rest]

Expression2Rest:

{Infixop Expression3}

Expression3 `instanceof` *Type*

Infixop:

`||`

`&&`

`|`

`^`

`&`

`==`

`!=`

`<`

`>`

`<=`

`>=`

`<<`

`>>`

`>>>`

```
        +
        -
        *
        /
        %
    Expression3:
        PrefixOp Expression3
          ( Expr | Type ) Expression3
         Primary {Selector} {PostfixOp}

    Primary:
        ( Expression )
        this [Arguments]
        super SuperSuffix
        Literal
        new Creator
        Ident { . Ident }[ [ ( ] BracketsOpt . class | Expression ]) | Arguments
            | . ( class | this | super Arguments | new InnerCreator )]
        BasicType BracketsOpt .class
        void.class

    PrefixOp:
        ++
        --
        !
        ~
        +
        -
```

PostfixOp:

```
++
--
```

Selector:

```
. Ident [Arguments]
. this
. super SuperSuffix
. new InnerCreator
[ Expression ]
```

SuperSuffix:

```
Arguments
. Ident [Arguments]
```

BasicType:

```
byte
short
char
int
long
float
double
boolean
```

ArgumentsOpt:

```
[ Arguments ]
```

Arguments:

(*[Expression { , Expression }]*)

BracketsOpt:

{[]*}*

Creator:

Qualident (ArrayCreatorRest | ClassCreatorRest)

InnerCreator:

Ident ClassCreatorRest

ArrayCreatorRest:

[*(*] *BracketsOpt ArrayInitializer | Expression*] *{*[*Expression*]*}* *BracketsOpt)*

ClassCreatorRest:

Arguments [ClassBody]

ArrayInitializer:

{ *[VariableInitializer {*, *VariableInitializer} [,]]* }

VariableInitializer:

ArrayInitializer

Expression

ParExpression:

(*Expression*)

Block:

{ *BlockStatements* }

BlockStatements:

{ BlockStatement }

BlockStatement :

LocalVariableDeclarationStatement

ClassOrInterfaceDeclaration

[Ident :] Statement

LocalVariableDeclarationStatement:

[`final`*] Type VariableDeclarators* `;`

Statement:

Block

`if` *ParExpression Statement [*`else` *Statement]*

`for` `(` *ForInitOpt* `;` *[Expression]* `;` *ForUpdateOpt* `)` *Statement*

`while` *ParExpression Statement*

`do` *Statement* `while` *ParExpression* `;`

`try` *Block (Catches* | *[Catches]* `finally` *Block)*

`switch` *ParExpression* `{` *SwitchBlockStatementGroups* `}`

`synchronized` *ParExpression Block*

`return` *[Expression]* `;`

`throw` *Expression* `;`

`break` *[Ident]*

`continue` *[Ident]*

`;`

ExpressionStatement

Ident `:` *Statement*

Catches:

CatchClause {CatchClause}

CatchClause:

`catch` `(` *FormalParameter* `)` *Block*

SwitchBlockStatementGroups:

{ SwitchBlockStatementGroup }

SwitchBlockStatementGroup:

SwitchLabel BlockStatements

SwitchLabel:

`case` *ConstantExpression* `:`

`default :`

MoreStatementExpressions:

{ `,` *StatementExpression }*

ForInit:

StatementExpression MoreStatementExpressions

[`final`*] Type VariableDeclarators*

ForUpdate:

StatementExpression MoreStatementExpressions

ModifiersOpt:

{ Modifier }

Modifier:

`public`

`protected`

`private`

`static`

`abstract`

`final`

`native`

`synchronized`

`transient`

`volatile`

`strictfp`

VariableDeclarators:

VariableDeclarator { `,` *VariableDeclarator }*

VariableDeclaratorsRest:

VariableDeclaratorRest { , *VariableDeclarator* }

ConstantDeclaratorsRest:

ConstantDeclaratorRest { , *ConstantDeclarator* }

VariableDeclarator:

Ident VariableDeclaratorRest

ConstantDeclarator:

Ident ConstantDeclaratorRest

VariableDeclaratorRest:

BracketsOpt [= *VariableInitializer*]

ConstantDeclaratorRest:

BracketsOpt = *VariableInitializer*

VariableDeclaratorId:

Ident BracketsOpt

CompilationUnit:

[`package` *Qualident* ;] {*ImportDeclaration*} {*TypeDeclaration*}

ImportDeclaration:

`import` *Ident* { . *Ident* } [. *] ;

TypeDeclaration:

ClassOrInterfaceDeclaration

;

ClassOrInterfaceDeclaration:

ModifiersOpt (*ClassDeclaration* | *InterfaceDeclaration*)

ClassDeclaration:

`class` *Ident* [`extends` *Type*] [`implements` *TypeList*] *ClassBody*

InterfaceDeclaration:

`interface` *Ident* [`extends` *TypeList*] *InterfaceBody*

TypeList:

Type { , *Type*}

ClassBody:

{ *{ClassBodyDeclaration}* }

InterfaceBody:

{ *{InterfaceBodyDeclaration}* }

ClassBodyDeclaration:

;

[`static`*] Block*

ModifiersOpt ((Type Ident (VariableDeclaratorsRest ; | *MethodDeclaratorRest))* | `void` *Ident MethodDeclaratorRest* | *Ident ConstructorDeclaratorRest* | *ClassOrInterfaceDeclaration)*

InterfaceBodyDeclaration:

;

ModifiersOpt (((Type Ident((ConstantDeclaratorsRest `;`*)*| *InterfaceMethodDeclaratorRest))* | *(*`void` *Ident VoidInterfaceMethodDeclaratorRest))* | *ClassOrInterfaceDeclaration)*

MethodDeclaratorRest:

FormalParameters BracketsOpt [`throws` *QualidentList] (MethodBody* | ; *)*

VoidMethodDeclaratorRest:

FormalParameters [`throws` *QualidentList] (MethodBody* | ; *)*

InterfaceMethodDeclaratorRest:

FormalParameters BracketsOpt [`throws` *QualidentList]* ;

VoidInterfaceMethodDeclaratorRest:

FormalParameters [`throws` *QualidentList]* ;

ConstructorDeclaratorRest:

FormalParameters [`throws` *QualidentList] MethodBody*

QualidentList:

Qualident { , *Qualident}*

FormalParameters:

(*[FormalParameter { , FormalParameter}]*)

FormalParameter:

[`final`*] Type VariableDeclaratorId*

MethodBody:

Block

Credits

THE following organizations and copyright holders granted permission for quotations used in this book.

Time after Time. Words and Music by Cyndi Lauper and Rob Hyman © 1983 Rellla Music Co. and Dub Notes. All Rights Administered by Sony/ATV Music Publishing, 8 Music Square West, Nashville, TN 37203. International Copyright Secured. All Rights Reserved.

The Lion Sleeps Tonight. New lyric and revised music by George David Weiss, Hugo Peretti and Luigi Creatore. © 1961 Folkways Music Publishers, Inc. © Renewed 1989 by George David Weiss, Luigi Creatore and June Peretti. © Assigned to Abilene Music, Inc. All Rights Reserved. Used by Permission. WARNER BROS. PUBLICATIONS U.S. INC., Miami, FL 33014.

Lyric excerpt of "*My Favorite Things*" by Richard Rodgers and Oscar Hammerstein II. Copyright © 1959 by Richard Rodgers and Oscar Hammerstein II. Copyright Renewed. WILLIAMSON MUSIC owner of publication and allied rights throughout the world. International Copyright Secured. All Rights Reserved.

Up, Up and Away. Words and Music by Jimmy Webb. Copyright © 1967 (Renewed 1995) CHARLES KOPPELMAN MUSIC, MARTIN BANDIER MUSIC and JONATHAN THREE MUSIC CO. International Copyright Secured. All Rights Reserved.

Did You Ever Have to Make Up Your Mind? Words and Music by John Sebastian. Copyright © 1965, 1966 (Copyrights Renewed) by Alley Music and Trio Music, Inc. All rights administered by Hudson Bay Music, Inc. International Copyright Secured. All Rights Reserved. Used by Permission. WARNER BROS. PUBLICATIONS U.S. INC., Miami, FL 33014.

Way Down Yonder in New Orleans. Words and Music by Henry Creamer and J. Turner Layton. Copyright © 1922 Shapiro, Bernstein & Co., Inc., New York. Copyright Renewed. International Copyright Secured. All Rights Reserved. Used by Permission.

Lyric excerpt of "*Space Oddity*" by David Bowie. Used by Permission. © 1969 David Bowie.

"*From Arthur a Grammar*", HOW TO WRITE, Gertrude Stein, 1931. Republished by Dover Publications, 1975. Reprinted with permission.

A NIGHT AT THE OPERA, Groucho Marx 1935. © 1935 Turner Entertainment Co. All rights reserved.