Contents

The Java™ Language Specification—DRAFT

This document contains a complete specification of version 1.0 of the Java™ Language, a class-based object-oriented programming language, and of the Java package `java.lang`, which contains certain built-in classes that are defined to be part of the Java Language.

The Java Language is designed to be machine-independent and platform-independent. The language is fully specified. No part or behavior of the language is described as "implementation-dependent" or "undefined".

Our intention is that the behavior of every construct be specified by the language specification, and that all compilers should accept the same programs. Excepting programs which are timing dependent or otherwise non-deterministic, and given sufficient time and sufficient memory, a Java program should compute the same result on all machines.

We believe that less is more. The Java applet API's provide a small set of packages that can be supported on many host platforms and in many standalone environments. The intention is to provide a basic set of facilities which the programmer can count on across all Java implementations.

We have opted for a simple language and base system in the hope that it will be widely and completely understood, implemented and adopted. We are familiar with much more complicated systems, but realizing that the system will only get larger in later releases we have elected to start small. Besides, with the explosion of the Internet and dynamic linking a new package is just a TCP connection away...

In their book on "The C Programming Language", Brian Kernighan and Dennis Ritchie said that they felt that the C language "wears well as one's experience with it grows." If you like C we think you will like Java, and we hope that it, too, wears well for you.

P.S.:Please send comments on the Java Language and System Specification by electronic mail to java@java.Sun.COM, and visit our home page at `http://java.sun.com`, where more documentation and Java related discussion groups are available.

0.1 Grammar Notation

This manual uses context-free grammars to define the lexical and syntactic structure of a Java program. The lexical structure is defined by the grammar in §1, the syntactic structure in §5, §6, §7, §8, and §9. All the lexical and syntactic rules are collected together in §10.

Terminal symbols are shown in a `fixed width` font in the lexical and syntactic productions and throughout this reference manual. These are to appear in the program exactly as written. Note that the terminals in a lexical production are individual characters, whereas the terminals in the syntactic productions are tokens consisting of certain sequences of characters (§1.3).

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 expensions for the nonterminal then follow on succeeding lines. For example, the definition

> *Catch:*
>   `catch` `(` *Argument* `)` *Block*

states that the nonterminal *Catch* represents the token `catch` followed by a left parenthesis token followed by an *Argument* followed by a right parenthesis token followed by a *Block*. As another example, the 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*. (Notice that the definition of *ArgumentList* is recursive; the result is that, in principle, an *ArgumentList* may contain any positive number of arguments, so far as the grammar is concerned. Such recursive definitions of nonterminals are common.)

This example states that a *JumpStatement* may have any one of four forms; the last form, for example, consists of the token `throw` followed by an *Expression* followed by a semicolon. The subscripted suffix $_{opt}$, which may appear after a terminal or after a nonterminal, means that element is optional: the alternative containing the optional element actually has two cases, one that omits the optional element and one that includes it. In effect, the definition

> *JumpStatement:*
>   `break` $Identifier_{opt}$ `;`
>   `continue` $Identifier_{opt}$ `;`
>   `return` $Expression_{opt}$ `;`
>   `throw` *Expression* `;`

is merely a convenient abbreviation for

> *JumpStatement:*
>   `break ;`
>   `break` *Identifier* `;`
>   `continue ;`
>   `continue` *Identifier* `;`
>   `return ;`
>   `return` *Expression* `;`
>   `throw` *Expression* `;`

and the definition

MethodDeclaration:
  MethodModifiers$_{opt}$ *ResultType MethodDeclarator Throws*$_{opt}$ *MethodBody*

is merely a convenient abbreviation for

MethodDeclaration:
  ResultType MethodDeclarator Throws$_{opt}$ *MethodBody*
  MethodModifiers ResultType MethodDeclarator Throws$_{opt}$ *MethodBody*

which in turn is an abbreviation for

MethodDeclaration:
  ResultType MethodDeclarator MethodBody
  ResultType MethodDeclarator Throws MethodBody
  MethodModifiers ResultType MethodDeclarator MethodBody
  MethodModifiers ResultType MethodDeclarator Throws MethodBody

and so *MethodDeclaration* actually has four alternative forms.

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,

Digit: one of
```
0 1 2 3 4 5 6 7 8 9
```

is merely a convenient abbreviation for

Digit:
```
0
1
2
3
4
5
6
7
8
9
```

As an extension of this notation, when such a “one of” alternative in the lexical part of the grammar appears to be a token, it represents the sequence of characters that make up the token. For example, the definition

Keyword: one of
```
abstract boolean break byte
```

in the lexical part of the grammar would really mean

Keyword:
```
a b s t r a c t
b o o l e a n
b r e a k
b y t e
```

1 Lexical Structure

Java programs are written using the Unicode character encoding. (For information about Unicode, see *The Unicode Standard: Worldwide Character Encoding*, Version 1.0, Volume 1 ISBN 0-201-56788-1 and Volume 2 ISBN 0-201-60845-6, and the additional information about Unicode 1.1 at `ftp://unicode.org`.)

This chapter describes the translation of a raw Unicode character stream into a stream of Java tokens, using the following three translations, which are applied in turn:

1. A translation of the raw stream of Unicode characters, allowing any Unicode character to be input as an ASCII escape sequence, resulting in an escaped Unicode stream.
2. A translation of the escaped Unicode stream into a stream of input characters and line terminators.
3. A translation of the stream of Unicode characters and line terminators into a sequence of Java tokens.

In each of these translations the longest possible translation is chosen at each step, even if the result does not ultimately make a legal Java program, while another translation would. Thus the characters `a--b` are tokenized as `a`, `--`, `b`, which cannot become part of a grammatically correct Java program, even though the tokenization `a`, `-`, `-`, `b` could be part of a grammatically correct Java program.

On systems that do not support full Unicode, translations between Unicode and the native character encoding must be provided. For example, Unicode is effectively a superset of ASCII, and Java translation step 1, described just above, provides a simple way to encode any Unicode character, anywhere in the source code of a program, as an escape sequence of ASCII characters. Source code may thus be stored as ASCII files rather than as full Unicode source files. If each ASCII character in the source code file is simply mapped to the corresponding Unicode character as it fed to the Java compiler, translation step 1 will then reconstruct Unicode characters represented as escape sequences.

1.1 Unicode Escapes

All Java implementations first perform a transformation on the raw Unicode character input, translating the characters `\u` followed by four hexadecimal digits to the Unicode character whose code point is the indicated hexadecimal value. This transformation results in a sequence of escaped input characters.

EscapedInputCharacter:
  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 0 a b c d e f A B C D E F`

Note that `\`, `u`, and all the hexadecimal digits are ASCII characters.

A Unicode escape sequence may contain more than one occurrence of the letter u before the hexadecimal digits. (Programmers writing Java programmers are unlikely to need this feature; it is included to allow a simple automatic translation of Java source code from full Unicode to an ASCII file representation that is itself a correct Java program if naively mapped back to Unicode but from which the original Unicode file can be reconstructed exactly.)

If a \ is not followed by `u`, then it is treated as a *RawInputCharacter* and remains as part of the escaped Unicode stream. If a \ is followed by `u`, or more than one `u`, but the last `u` is not followed by four hexadecimal digits, then it is a compile-time error.

The character produced by a Unicode escape is not subject to rescanning. For example, the raw input `\u005cu005b` results in the four characters `\ u 5 b`, not the single character `Z` (which is Unicode character `005b`); while `005c` is indeed the Unicode value for `\`, the resulting `\` is not interpreted as the start of a further Unicode escape sequence.

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

1.2 Input Lines

The second translation step divides the sequence of escaped input characters into lines by recognizing lines as being terminated by the ASCII characters `CR LF` or `CR` or `LF`. (The idea is that either a carriage return `CR` or a line feed `LF` by itself can serve as a line terminator, but `CR` immediately followed by `LF` is counted as one line teminator, not two.) The result is a sequence of line terminators and input characters, which are the terminal symbols for the tokenization process in the third step.

LineTerminator:
  the ASCII `CR` *character followed by the ASCII* `LF` *character*
  the ASCII `CR` *character*
  the ASCII `LF` *character*

InputCharacter:
  EscapedInputCharacter, but not `CR` *and not* `LF`

This definition of what is a line determines any line numbers produced by a Java compiler or other Java system component. It also specifies the termination of the `//` form of comment (§1.4).

1.3 Tokens

The input characters resulting from escape processing and input line recognition are further processed by recognizing tokens and discarding comments and whitespace, thereby reducing the input to a sequence of tokens. This process is described by the following grammar:

Input:
  InputElements$_{opt}$

InputElements:
  InputElement
  InputElements InputElement

InputElement:
Comment
WhiteSpace
Token

WhiteSpace:
the ASCII `SP` *character*
the ASCII `HT` *character*
the ASCII `FF` *character*
LineTerminator

Token:
Keyword
Identifer
Literal
Separator
Operator

As usual, this translation works from left to right and, as usual, the longest possible match is chosen at each step.

Whitespace is defined as the ASCII space, horizontal tab, and form feed characters as well as line separators, previously recognized as `CR`, `LF`, or `CR LF`. As a special concession for compatibility with certain operating systems, the ASCII `SUB` character (`\u1a`) is also treated as whitespace if it is the last character in the escaped input stream.

Comments and white space serve to separate adjacent tokens that, if adjacent, might be tokenized in another manner. For example, the characters − and = in the input can form the operator token −= only if there is no intervening white space or comment.

1.4 Comments

A comment in a Java program begins with an occurrence of the characters `/*`, `/**`, or `//`.

Comment:
`/ *` *NotStar TraditionalCommentTail*
`/ * *` *DocCommentTail*
`/ /` *CharactersInLine*$_{opt}$ *LineTerminator*

TraditionalCommentTail:
`* /`
NotStar TraditionalCommentTail
`*` *NotSlash TraditionalCommentTail*

DocCommentTail:
`/`
`*` *DocCommentTail*
NotStarNotSlash TraditionalCommentTail

NotStar:
InputCharacter, but not `*`
LineTerminator

NotSlash:
InputCharacter, but not `/`
LineTerminator

NotStarNotSlash:
  InputCharacter, but not `*` *or* `/`
  LineTerminator

CharactersInLine:
  InputCharacter
  CharactersInLine InputCharacter

These three styles of comments are:

`/* text */` All the text from `/*` to `*/` is ignored.

`/** text */` The enclosed text is used in automatically generated documentation of the following declaration. The default format for the content of the comments is described in ???

`// text` All the text from `//` to the end of the line is ignored.

The grammar implies the following:

- Comments do not nest.
- Comments do not occur within string and character literals.
- `/*` and `*/` have no special meaning in `//` comments.
- `//` has no special meaning in comments that begin with `/*` or `/**`.

As a result, the text

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

is a single complete comment.

1.5 Keywords

The following character sequences, formed from ASCII letters, constitute special tokens that are reserved for use as keywords.

Keyword: one of

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

The keywords `byvalue`, `cast`, `const`, `future`, `generic`, `goto`, `inner`, `operator`, `outer`, `rest`, and `var` are reserved but not used in Java 1.0.

Note that `true` and `false` look like keywords but technically are Boolean literals (§1.7.3).

1.6 Identifiers

An identifier is an unlimited length sequence of Unicode letters and digits, the first of which must be a letter. An identifier must not have the same spelling (code point sequence) as a keyword.

Identifier:
  UnicodeLetter
  Identifier UnicodeLetter
  Identifier UnicodeDigit

A Unicode character is a digit if its Unicode name contains the word "DIGIT", as listed on pages 391–393 of *The Unicode Standard, Version 1.0*, Volume 1 (see §1.1 for the ISBN); this is precisely the characters in the following ranges:

`\u0030-\u0039`	`0-9`	ISO-LATIN-1 digits
`\u0660-\u0669`		Arabic-Indic digits
`\u06f0-\u06f9`		Eastern Arabic-Indic digits
`\u0966-\u096f`		Devanagari digits
`\u09e6-\u09ef`		Bengali digits
`\u0a66-\u0a6f`		Gurmukhi digits
`\u0ae6-\u0aef`		Gujarati digits
`\u0b66-\u0b6f`		Oriya digits
`\u0be7-\u0bef`		Tamil digits
`\u0c66-\u0c6f`		Telugu digits
`\u0ce6-\u0cef`		Kannada digits
`\u0d66-\u0d6f`		Malayalam digits
`\u0e50-\u0e59`		Thai digits
`\u0ed0-\u0ed9`		Lao digits
`\u1040-\u1049`		Tibetan digits

A Unicode character is a letter if it falls in one of the following ranges and is not a digit:

`\u0024`	`$`	dollar sign (for historical reasons)
`\u0041-\u005a`	`A-Z`	Latin capital letters
`\u005f`	`_`	underscore (for historical reasons)
`\u0061-\u007a`	`a-z`	Latin small letters
`\u00c0-\u00d6`		various Latin letters with diacritics
`\u00d8-\u00f6`		various Latin letters with diacritics
`\u00f8-\u00ff`		various Latin letters with diacritics
`\u0100-\u1fff`		other non-CJK alphabets and symbols
`\u3040-\u318f`		Hiragana, Katakana, Bopomofo, and Hangul
`\u3300-\u337f`		CJK squared words
`\u3400-\u3d2d`		Korean Hangul Symbols
`\u4e00-\u9fff`		Han (Chinese, Japanese, Korean)
`\uf900-\ufaff`		Han compatibility

Two identifiers are the same only if they are identical, that is, have the same Unicode code point for each letter or digit. This means, in particular, that the identifiers consisting of the single letters Latin capital `A` (`\u0041`), Latin small `a` (`\u0061`), Greek capital `A` (`\u0391`), and Cyrillic small `a` (`\u0430`) are all distinct.

This means that composite characters are distinct from the decomposed characters. For example, a LATIN CAPITAL LETTER A GRAVE `\u00c0` could be considered to be the same as a LATIN CAPITAL LETTER A `\u0041` followed by a NON-SPACING-GRAVE `\u0300` when sorting, but these are distinct in Java. See *The Unicode Standard*, Volume 1, pages 626–627, about sorting, and pages 412ff about decomposition.

1.7 Literals

A literal is the source representation of a value of a primitive or `String` type:

Literal:
IntegerLiteral
FloatingPointLiteral
BooleanLiteral
CharacterLiteral
StringLiteral

See §3 for a description of primitive types.

1.7.1 *Integer Literals*

Integer literals may be expressed in decimal (base 10), hexadecimal (base 16), or octal (base 8) notation, using characters from the ASCII character set portion of Unicode:

IntegerLiteral:
DecimalLiteral IntegerTypeSuffix$_{opt}$
HexLiteral IntegerTypeSuffix$_{opt}$
OctalLiteral IntegerTypeSuffix$_{opt}$

An integer literal is of type `long` if it is suffixed with an `L` or `l`; otherwise it is of type `int`.

IntegerTypeSuffix: one of
`l L`

A decimal literal consists of a digit from `1` to `9`, optionally followed by one or more digits from `0` to `9`, and represents a positive integer:

DecimalLiteral:
NonZeroDigit Digits$_{opt}$

Digits:
Digit
Digits Digit

Digit:
`0`
NonZeroDigit

NonZeroDigit: one of
`1 2 3 4 5 6 7 8 9`

A hexadecimal literal consists of a leading `0x` or `0X` followed by one or more hexadecimal digits and can represent a positive, zero, or negative number. Hexadecimal digits with values 10 through 15 are represented by the letters `a` through `f` or `A` through `F`, respectively; each letter used as a hexadecimal digit may be uppercase or lowercase.

HexLiteral:
`0x` *HexDigit*
`0X` *HexDigit*
HexLiteral HexDigit

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

An octal literal consists of a digit `0` optionally followed by zero or more of the digits `0` through `7` and can represent a positive, zero, or negative number.

OctalLiteral:
`0`
OctalLiteral OctalDigit

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

The largest decimal literal of type `int` is `2147483647` (`2^31-1`). The largest positive hexadecimal and octal literals of type `int` are `0x7fffffff` and `017777777777` respectively, both representing `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` each represent the decimal value `-1`.

It is a compile-time error for a decimal literal of type `int` to be larger than `2^31-1`, or for a hexadecimal or octal `int` literal to provide more than 32 bits. This means, in particular, that the largest negative `int` cannot be represented as a decimal literal, because "`-2147483648`" appearing in Java source code would be tokenized as the unary operator `-` followed by a putative decimal literal `2147483648`, but `2147483648` is not a valid decimal integer literal. Use the hexadecimal literal `0x80000000` instead.

Examples of `int` literals are:

```
0     2     0666      0xDadaCafe
```

The largest decimal literal of type `long` is `9223372036854775807L` (`2^63-1`). The largest positive octal and hexadecimal literals of type `long` are `0777777777777777777777L` and `0x7fffffffffffffffL` respectively; each represents `9223372036854775807L` (`2^63-1`). The most negative hexadecimal and octal literals of type `long` are `0xffffffffffffffffL` and `01777777777777777777777`, each of which represents the decimal value `-9223372036854775808` (`-2^63`). The most negative hexadecimal and octal literals of type `long` are `0x8000000000000000L` and `04000000000000000000000L` respectively, each of which represents the decimal value –9223372036854775808 (`-2^63`). The hexadecimal and octal literals `0xffffffffffffffffL` and `01777777777777777777777L` each represent the decimal value `-1L`.

It is a compile time error for a decimal literal of type `int` to be larger than `2^63-1` or for a hexadecimal or octal `long` literal to provide more than 64 bits. This means, in particular, that the largest negative `long` cannot be represented as a decimal literal, because "`-9223372036854775808L`" appearing in Java source code would be tokenized as the unary operator `-` followed by a putative decimal literal `9223372036854775808L`, but `9223372036854775808L` is not a valid decimal long integer literal. Use the hexadecimal literal `0x8000000000000000` instead.

Examples of `long` literals are:

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

1.7.2 *Floating-Point Literals*

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

It is required to have at least one digit, in either the whole number or the fraction part, and either a decimal point or an exponent. All other parts are optional.

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

> *FloatingPointLiteral :*
> *Digits* `.` *Digits*$_{opt}$ *ExponentPart*$_{opt}$ *FloatTypeSuffix*$_{opt}$
> `.` *Digits* *ExponentPart*$_{opt}$ *FloatTypeSuffix*$_{opt}$
> *Digits* *ExponentPart* *FloatTypeSuffix*$_{opt}$
>
> *ExponentPart:*
> *ExponentIndicator* *SignedInteger*$_{opt}$
>
> *ExponentIndicator: one of*
> `e E`
>
> *SignedInteger:*
> *Sign*$_{opt}$ *Digits*
>
> *Sign: one of*
> `+ -`
>
> *FloatTypeSuffix: one of*
> `f F d D`

It is a compile-time error for a non-zero floating-point literal to be too large, so that on rounded conversion to its internal representation it becomes an IEEE infinity, or nonzero but too small, so that on rounded conversion to its internal representation it becomes a zero.

The largest floating-point literal of type `float` is `3.40282347e+38f`. The smallest floating-point literal of type `float` is `1.40239846e-45f`.

The largest floating-point literal of type `double` is `1.79769313486231570e+308`. The smallest floating-point literal of type `double` is `4.94065645841246544e-324`.

Predefined constants representing the positive and negative infinities and Not-a-Number (NaN) values of both `float` and `double` types are defined in the standard classes `Float` and `Double`; see §16.6.

Examples of `float` literals:

```
1e1f      2.f       .3f       3.14f     6.02e+23f
```

Examples of `double` literals:

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

1.7.3 *Boolean Literals*

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

> *BooleanLiteral: one of*
> `true      false`

1.7.4 *Character Literals*

A literal of type `char` is expressed as a character or an escape sequence enclosed in single quotes. The escape sequences allow for the representation of some non-graphic characters as well as the single quote and the backslash in character and string literals.

CharacterLiteral:
    `'` *SingleCharacter* `'`
    `'` *Escape* `'`

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

Escape:

```
\ 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`

Note that 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 *Escape* to be other than a `'`; it is a compile-time error for a line terminator to appear after the first `'` and before the closing `'`. Note that, 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(§1.1), and the linefeed becomes a *LineTerminator* in step 2(§1.2), and so the character literal is not valid in step 3. Instead, one should write `'\n'`. Similarly, it is not correct to write '\u000d' for a character literal whose value is carriage return `CR`. Instead, one should write `'\r'`.

It is a compile-time error if the character following a backslash in an escape is not `b`, `t`, `n`, `f`, `r`, `"`, `'`, `\`, `0`, `1`, `2`, `3`, `4`, `5`, `6`, or `7`. (Recall that the Unicode escape `\u` is processed very early; see §1.1.)

Examples of `char` literals:

```
'a'      '\t'      '\\'      '\u15e'
```

1.7.5 String Literals

A string literal is zero or more characters enclosed in double quotes, and may use the escape sequences defined above:

StringLiteral:
    `"` *StringCharacters* `"`

StringCharacters:
  StringCharacter
  StringCharacters StringCharacter

StringCharacter:
  InputCharacter, but not `"` *or* `\`
  Escape

It is a compile-time error for a line separator to appear after the first `"` and before the closing `"`. In most situations a long string constant can be broken up into shorter pieces and written as an expression using the string concatenation operator + (§9.12.2).

Each string literal is a reference to an instance of class `String`; such literals cannot be modified.

Examples of string literals:

```
""                         // the empty string
"\""                       // string containing " alone
"This is a string"
"This is a " +             // actually a string-valued expression
     "two-line string"     // containing two string literals
```

1.8 Separators

The following characters are used in Java code as separators:

Separator: one of

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

1.9 Operators

The following characters and character combinations are defined as operators.

Operator: one of

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

2 Types and Values

There are four kinds of data types in Java: class types, interface types, array types, and primitive types. Every variable has an associated data type, sometimes called its "compile-time type" because its type can always be determined by the compiler, before the program is executed. There are two kinds of data values that can be stored in variables, passed as arguments, returned as values, and operated upon: *references* and *primitive values*. The value stored in a variable must be compatible with the compile-time type of the variable.

Type:
PrimitiveType
ClassType
InterfaceType
ArrayType

References are "pointers" to dynamically allocated objects. There are two kinds of dynamically allocated objects: class instances and arrays. Every object that is not an array is an instance of some particular class; this class is sometimes called the "run-time type" of the object. Every array also has a run-time type. If the value of a variable is a reference to an object, then the run-time type of the object must be compatible with the compile-time type of the variable.

There may be many references to the same object or to the same array. Objects may contain state information in field variables belonging to the object. If two variables contain references to the same object, it is possible to modify the state information through one reference to the object and then observe the altered state through another reference.

Primitive values are indivisible and do not share state with other primitive values. A variable whose (compile-time) type is a primitive type always holds a value of that exact primitive type. Such a value is not shared in any way with any other variable, so the value of the variable can be changed only by operations using that variable.

2.1 Primitive Types and Values

The primitive types available in every Java program are:

- the arithmetic types:
  - the integral types:
    - `byte`, whose values are 8-bit signed two's-complement integers
    - `short`, whose values are 16-bit signed two's-complement integers
    - `int`, whose values are 32-bit signed two's-complement integers
    - `long`, whose values are 64-bit signed two's-complement integers
  - the floating-point types:
    - `float`, whose values are 32-bit IEEE 754 floating-point numbers
    - `double`, whose values are 64-bit IEEE 754 floating-point numbers
- the character type `char`, whose values are 16-bit Unicode characters
- the `boolean` type, whose values are `true` and `false`

A primitive type is named by its reserved keyword:

PrimitiveType: one of
`boolean  char  byte  short  int  long  float  double`

2.1.1 *Integral Types and Values*

The primitive integral types are `byte`, `short`, `int`, and `long`, which are respectively 8-bit, 16-bit, 32-bit, and 64- bit signed two's-complement integers, and `char`, which is a 16-bit unsigned integer representing a Unicode code point.

The values of type `byte` are integers ranging from –256 to 255, inclusive.

The values of type `short` are integers ranging from –32768 to 32767, inclusive.

The values of type `int` are integers ranging from –2147483648 to 2147483647, inclusive.

The values of type `long` are integers ranging from –9223372036854775808 to 9223372036854775807, inclusive.

Any value of any integral type may be cast to any other arithmetic type.

Any value of any integral type may be cast to type `char`, and any character may be cast to any integer type.

There are no casts between integer types and the type `boolean`.

Java provides a number of operators that act on integer values:

- the basic equality operators = and `!=`
- the relational operators <, <=, >, and >=
- the unary operators + and –
- the additive and multiplicative operators `+`, `–`, `*`, `/`, and `%`
- the prefix and postfix increment/decrement operators ++ and `––`
- the signed and unsigned shift operators <<, >>, and >>>
- the unary bitwise logical negation operator ~
- the binary bitwise logical operators `&`, |, and `^`

If both operands are of integral type, the operation is considered an integer operation. If at least one of the operands is of type `long`, then the operation is carried out using 64-bit precision (any other operand that is not `long` is first widened, as if by a cast, to type `long`) and the result, if not `boolean`, is of type `long`. Otherwise, the operation is carried out using 32-bit precision (any other operand that is not `int` is first widened, as if by a cast, to type `int`) and the result, if not `boolean`, is of type `int`.

Note that while the built-in operators listed above always widen their operands so as to operate at 32-bit or 64-bit precision, values of integral type are *not* automatically widened when used as arguments in method calls. Individual defined methods may be coded so as to perform such widening, but the calling process itself does not do automatic widening.

The built-in integer operators produce only the low 32 bits or 64 bits of their two's-complement arithmetic result and do not indicate an overflow or underflow in any way.

Java throws an `ArithmeticException` if the right-hand operand to an integer divide operator `/` or integer remainder operator `%` is zero; this is the only case where an exception is generated by an operator on integral types.

2.1.2 *Floating-Point Types and Values*

The floating-point types are `float` and `double`, representing 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 Std. 754-1985 (IEEE, New York).

The floating-point value of type `float`, arranged from smallest to largest, are: negative infinity, negative finite values, negative zero, positive zero, positive finite values and positive infinity. There is also a special value Not-a-Number (NaN), which is used to represent the result of certain operations such as dividing zero by zero; most operations with NaN as an operand produce NaN as a result.

The finite nonzero values of type `float` are of the form $s \cdot m \cdot 2^e$, where s is +1 or –1, m is a positive integer less than 2^24, and e is an integer between –149 and 104, inclusive.

The finite nonzero values of type `double` are of the form $s \cdot m \cdot 2^e$, where s is +1 or –1, m is a positive integer less than 2^53, and e is an integer between –1045 and 1000, inclusive.

Positive zero and negative zero compare equal (`0.0 == -0.0` produces the result `true`) but there are other operations that can distinguish them; for example, `1.0/0.0` produces positive infinity but `1.0/-0.0` produces negative infinity.

Any value of any floating-point type may be cast to any other arithmetic type.

Any value of any floating-point type may be cast to type `char`, and any character may be cast to any floating-point type.

There are no casts between floating-point types and the type `boolean`.

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

- the basic equality operators = and `!=`
- the relational operators <, <=, >, and >=
- the unary operators + and `-`
- the additive and multiplicative operators `+`, `-`, `*`, `/`, and `%`
- the prefix and postfix increment/decrement operators ++ and `--`

If both operands are of floating-point type, or if one operand is of floating-point type and the other is of integral type, the operation is considered a floating-point operation. If at least one of the operands is of type `double`, then the operation is carried out using 64-bit floating-point arithmetic (any other operand that is not `double` is first cast to type `double`) and the result, if not `boolean`, is of type `double`. Otherwise, the operation is carried out using 32-bit floating-point arithmetic (any other operand that is not `float` is first cast to type `float`) and the result, if not `boolean`, is of type `float`.

Operators on floating-point numbers behave exactly as specified by IEEE 754. Java requires full support of IEEE 754 denormalized floating-point numbers.

Java requires that floating-point arithmetic behave as if every floating-point operator rounds its floating-point result to the result precision. Inexact results must be rounded to the nearest representable value; if two representable values are equally distant from the true mathematical result of the operation, the result is the value whose least-significant bit is 0. (This is the IEEE 754 "round to nearest"mode.) Note, however, that Java rounds towards zero when casting a floating value to an integer.

Java floating-point arithmetic produces no exceptions. An operation that overflows produce a signed infinity, an operation that underflows produce s a signed zero, and an operation that has no mathematically definite result produces NaN. Java uses gradual underflow.

While the usual relational operations apply to IEEE floating-point numbers, the presence of NaN can produce some surprises. NaN is unordered, so that the result of a <, <=, >, >=, or == comparison between a NaN and another value is always `false`; in particular, == produces `false` when both operands are NaN. The result of a `!=` comparison with a NaN is always `true`, even if the both operands are NaN.

2.1.3 *Character Types and Values*

2.1.4 *Boolean Types and Values*

The `boolean` type represents a 1-bit logical quantity with two possible values, indicated by the literals `true` and `false`.

There are no casts defined to or from `boolean`. (Note, however, that an integer `x` can be converted to a boolean, following the convention of the C programming language that treats `0` as `false` and every nonzero value as `true`, by the expression `x!=0`. Similarly, a boolean value `b` can be converted to a zero/one integer value by the expression `b?1:0`.)

Operations defined on `boolean` include the relational operators == and `!=`, the logical operators `!`, `&`, `|`, and `^`, and the short-circuit logical operators `&&` and `||`. The control flow in the `if`, `while`, `do`, and `for` statements, and which subexpression is to be chosen in the conditional `? :` operator, are controlled only by `boolean` truth values. For arithmetic types, an explicit comparison to zero is needed to turn a zero/non-zero condition of the value into a truth value; similarly, object references must be explicitly compared to `null` to produce usable truth values for use in these places.

2.2 Reference Types and Values

A variable of reference type can hold a reference to any object whose run-time type can be converted to the variable's compile-time type by assignment conversion.

ClassType:
  Name

InterfaceType:
  Name

ArrayType:
  Type `[]`

2.2.1 *Class Instances*

Created by `new`, have fields (variables and methods)

2.2.2 *Arrays*

Created by `new`, have components, which are variables; can have arrays of arrays; ultimate non-array components are elements; elements are variables whose type must be a class type, interface type, or primitive type.

2.2.3 *Class Types*

Variables of class type can hold references to subclasses. `Object` can refer to arrays.

2.2.4 *Array Types*

Variables of array type can hold references to arrays.

2.2.5 *Interface Types*

Variables of interface type can hold references to objects that implement the interface.

2.3 Standard Default Values

No variable in a Java program ever has an undefined value.

When a variable (such as an instance variable or an array component) is first created and no initial value is explicitly specified in the program, the variable is given the *standard default value* for its type:

- For type `byte`, the standard default value is zero, that is, the value of `(byte)0`.
- For type `short`, the standard default value is zero, that is, the value of `(short)0`.
- For type `int`, the standard default value is zero, that is, `0`.
- For type `long`, the standard default value is zero, that is, `0L`.
- For type `float`, the standard default value is positive zero, that is, `0.0f`.
- For type `double`, the standard default value is positive zero, that is, `0.0d`.
- For type `char`, the standard default value is the null character, that is, `'\u0000'`.
- For type `boolean`, the standard default value is `false`.
- For all reference types, the standard default value is `null`.

Note, however, that the Java compiler goes to some trouble to detect programs that use variables before they have been explicitly initialized or assigned. The automatic initialization of variables to standard default values is required to guarantee portability of Java code, but good Java programming style does not rely on it.

3 Conversions

In Java, there are four contexts for conversion: casting, assignment, method call, and arithmetic promotion. Casting is the most general context; if a conversion is permitted at all within Java, it can be achieved by casting. Assignment and method call allow only certain conversions; however, assignment and method call allow the same subset of the possible conversions, so it is convenient to speak of “assignment conversion” with the understanding that it applies also to method calls.

Arithmetic promotion is not a general feature of Java, but is a property of the specific definitions of the built-in arithmetic operations; there are two kinds of arithmetic promotion: unary arithmetic promotion and binary arithmetic promotion. (The analogous conversions in C are called “the usual unary conversions” and “the usual binary conversions”.) Please note that not all binary operators perform the binary arithmetic promotion.

Conversions never convert from a primitive value to a reference value or from a reference value to a primitive value.

3.1 Conversions on Primitive Values

A value of any primitive type may be “converted” to that same type. Of course, this results in no change to the value or its type.

The following type conversions are called *widening* 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 conversions do not lose information about the overall magnitude of a numeric value. Indeed, integer-to-integer and float-to-float widening conversions do not lose any information at all; the numeric value is preserved exactly. Conversion of an `int` or a `long` value to `float`, or of a `long` value to `double`, may *lose precision*, that is, may lose some of the least significant bits of the value; the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode.

According to this rule, a widening conversion of a signed integer 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.

The following type conversions are called *narrowing* 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.; they 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`. This may cause the resulting value not to have the same sign as 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`. This may cause the resulting value not to have the same sign as the input value.

A narrowing conversion of a floating-point number to an integral type `T` first truncates the floating-point value to an integer value (rounding toward zero). If this integer value can be represented as a value of type `T`, then that is the result of the conversion. Otherwise the value must be too small (a negative value of large magnitude) or too large (a positive value of large magnitude). If it is too small, the result of the conversion is the smallest representable value of type `T`; if it is too large, the result of the conversion is the largest representable value of type `T`. If the floating-point number if NaN, the result of the conversion is `0`.

A narrowing conversion from `double` to `float` behaves in accordance with IEEE 754. The result is correctly rounded using IEEE 754 round-to-nearest mode. A value of small magnitude may be converted to zero (positive or negative); a value of large magnitude may be converted to infinity (positive or negative); NaN is always converted to NaN.

A narrowing conversion of an integer to a floating-point type results in the closest possible value in the target format. The result is correctly rounded using IEEE 754 round-to-nearest mode.

Despite the fact that overflow, underflow, or loss of precision may occur, conversion among primitive types never results in a run-time exception.

3.2 Conversions on Reference Values

An object reference whose run-time type is `R` can be converted to a class type `C` if and only if `R` is `C` or a subclass of `C`.

An object reference whose run-time type is `R` can be converted to an interface type `I` if and only if `R` implements `I`. (Remember that if a class implements an interface, all its subclasses also automatically implement the interface, even if the subclass declarations do not mention the interface explicitly.)

An array reference can be converted to a class type `C` if and only if `C` is the class `Object`.

An array reference whose run-time type is `R[]` (an array whose components have type `R`) can be converted to an array type `T[]` (an array whose components have type `T`) if and only if either `R` and `T` are the same primitive type or `T` is a reference type and `R` can be converted to `T`.

3.3 Assignment Conversion

Assignment conversion, when applied to a variable and a value, converts the value to the type of the variable. Assignment permits only certain conversions to take place, namely those that require no run-time validity check and cannot lose information about numeric magnitude.

If the value of an expression of some compile-time type can be converted to the type of some variable, we say the expression (or its value) is *assignable* to the variable. If a type `S` can be converted to type `T` by assignment conversion, we say that `S` is *assignable* to `T`.

A value of any compile-time type can always be assigned to a variable of that same type. No conversion action need occur at run time, of course.

Consider this sequence of primitive types:

```
byte    short    int    long    float    double
```

Assignment conversion can convert any type in this series to any type that appears to its right. Furthermore, the same is true of this series:

```
        char    int    long    float    double
```

Such conversions are performed at run time as described in

The type `boolean` cannot be assigned to any other type.

A value of primitive type must not be assigned to a variable of reference type; similarly, a value of reference type must not be assigned to a variable of primitive type.

Assignment of a value of reference type to a variable of reference type requires no conversion action at run time. The basic principle is that the compiler must be able to prove from the compile-time type of the value that it can always be converted to the type of the variable. The detailed rules for assignment conversion of reference types are shown in Table 1.

	`T` is a class that is not `final`	`T` is a class that is `final`	`T` is an interface	`T` = `B[]`, an array with components of type `B`
`S` is a class that is not `final`	`T` must be a subclass of `S`	`T` must be a subclass of `S`	*compile-time error*	`S` must be `Object`
`S` is a class that is `final`	`T` must be the same class as `S`	`T` must be the same class as `S`	*compile-time error*	*compile-time error*
`S` is an interface	`T` must implement interface `S`	`T` must implement interface `S`	`T` must be a subinterface of `S`	*compile-time error*
`S` = `A[]`, an array with components of type `A`	*compile-time error*	*compile-time error*	*compile-time error*	*either* `A` and `B` are the same primitive type, *or* `A` is a reference type and `B` can be assigned to `A`

Table 1. Rules for permitted assignment conversion when assigning a reference value of type `T` to a variable of type `S`

3.4 Casting Conversion

Casting conversions are more general than assignment conversions. If a conversion is possible at all, a cast can do it.

A value of any compile-time type can always be cast to that same type. Such a cast has no run-time effect, of course, and serves only to indicate explicitly that the resulting value will be of the indicated type.

Casting can convert a value of any arithmetic type to any other arithmetic type.

The type `boolean` cannot be cast to any other type.

A value of primitive type cannot be cast to a reference type; similarly, a value of reference type cannot be cast to a primitive type.

Casting of a value of reference type to a variable of reference type may require a run-time validity check. The basic principle is that if the compiler is able to prove from the compile-time type of the value that it can always be converted to the type of the variable (that is, that assignment conversion applies), then no run-time check is required; otherwise, execution of the cast operator must verify at run time that the run-time type is compatible with the type of the variable (and if it is not compatible, an exception is thrown).

Some casts can be proven incorrect at compile time; such casts result in a compile-time error. The detailed rules for compile-time correctness of casting conversions on reference types are shown in Table 1.

	`T` is a class that is not `final`	`T` is a class that is `final`	`T` is an interface	`T` = `B[]`, an array with components of type `B`
`S` is a class that is not `final`	`T` must be a subclass of `S`, or `S` of `T`	`T` must be a subclass of `S`	*always correct at compile time*	`S` must be `Object`
`S` is a class that is `final`	`S` must be a subclass of `T`	`T` must be the same class as `S`	`S` must implement interface `T`	*compile-time error*
`S` is an interface	*always correct at compile time*	`T` must implement interface `S`	*always correct at compile time*	*compile-time error*
`S` = `A[]`, an array with components of type `A`	`T` must be `Object`	*compile-time error*	*compile-time error*	*either* `A` and `B` are the same primitive type, *or* `A` is a reference type and `B` can be cast to `A`

Table 2. Rules for permitted casting conversion when casting a reference value of type `T` to type `S`

3.5 Unary Arithmetic Promotion

When an operator applies unary arithmetic promotion to a single operand, the following rules apply, in order:

- If the operand is of type `byte` or `short`, it is converted to `int`.
- Otherwise it remains as is and is not converted.

3.6 Binary Arithmetic Promotion

When an operator applies binary arithmetic promotion to a pair of operands, the following rules apply, in order:

- 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`.

4 Names and Variables

4.1 Names

A name is an identifier that has been given meaning in a program by a declaration. A name denotes either:

- a package, which is introduced by a `package` statement (§5.1),
- a type, which is introduced by a `class` or `interface` declaration (§6),
- a field, which is a variable in a `class` or `interface` type (§6.2),
- a group of methods of a `class` or `interface` type (§6.4),
- a variable that is a formal parameter of a method (§6.6.4),
- a variable that is local to a block (§8.2), or
- a statement label (§8.5).

An expression is also said to have a denotation (§9.1), and can denote everything a name can denote as well as:

- a directory that is part of a package name,
- an array type,
- a value of a primitive type,
- a variable that is an element of an array,
- a reference to an object, or
- `null`, which is a reference to no object.

If a name or expression denotes a variable or a value of a primitive type, then the type of that variable or primitive value is called the type of the name or the expression.

4.2 Variables: Values and References

A variable is a typed storage location. A variable contains either a value of a primitive type (§1.7), (§3), or a reference to an object. An object is an instance of a class type (§6) or an instance of an array type (§7).

Variables have two main attributes: their type and their storage class(§4.3).

A variable's type is either a primitive type or an object type. An object type may be a class type, an interface type, or an array type.

A variable must always contain a value consistent with its type; in fact, Java is so designed that it is impossible for a variable to take on a value inconsistent with its type.

4.3 Storage Classes

The storage class determines the lifetime of a variable.

Local variables are declared and allocated within a block and are discarded on exit from the block. Method parameters are considered local variables.

Static variables are local to a class; they are allocated when the class is loaded and discarded when the class is unloaded.

Dynamic objects are instances of classes and arrays. They are allocated by the `new` expression (§9.8) and may be referenced by more than one variable. Automatic storage management techniques, such as garbage collection, are used to reclaim the storage used by dynamic objects. A class may declare a `finalize` method(§6.6) that will be called just before an instance of that class is discarded.

4.4 Name Spaces

Each name declared in a program is defined at a lexical level, and becomes part of a name space at that level. The name spaces in a Java program lexically nest as follows:

MCMANIS SUGGESTS DROPPING HOST FROM HOST SYSTEM. THINKS THIS SECTION IS MUDDY. HE SUGGESTS GETTING RID OF "HOST SYSTEM" AND CLEANING UP ACCESS CONTROL AT THIS LEVEL

0. Host system's package name space
    1. A compilation unit's name space
        2. A type's name space
            3. A method's parameter name space
                4. A local block's name space
                    5. A nested local block or `for`'s name space
                        . . .

The name spaces thus differ in the kinds of declarations they contain:

- a compilation unit's name space contains the type names declared in all compilation units of the package it belongs to and any package names or type names that are imported (§5.4),
- a type's name space contains its declared fields as well as any field names inherited from superclasses and interfaces (X.X),
- a method's parameter name space contains the formal parameters of the method(§6.4),
- a local block's name space contains local variables and labels declared in the block, and
- a for statement's name space contains any local variables declared in the initialization part of the for statement (§8.8.3).

Names introduced by import statements and local variable declarations must be declared before they are used. All other names are known throughout the name space in which they appear.

The host system package name space consists of the first component of the names of all of the packages that are available on the host system. It:

- always contains the `java` package name, used internally by the Java system,
- usually contain several all-upper-case ISO-LATIN-1 package names such as COM, EDU and FR; such names are reserved to be the first component of global package names, and should not otherwise be used,
- usually contains local packages whose initial names are not all upper-case ISO-LATIN-1 letters, which represent locally developed packages, and any other packages that have not been given globally unique names.

4.5 Name Resolution

When a name occurs in a Java program it is resolved by looking successively in the namespaces of each lexical level, looking from the highest nesting level to the lowest. Only the first match is considered.

**** Need a discussion of ambiguity when looking up field names; this can occur when the field variable appears in more than one interface, for example.

Let `F` be the class or interface whose definition contains the declaration of a field variable. Access to the field is controlled as follows:

- If `F` is the same as the class or interface in whose body the field access expression appears, then the access is allowed.
- Otherwise, if `F` is a class (not an interface) that is a superclass of the class in whose body the field access expression appears, then access is allowed only if the field is declared to be `protected` or `public`.
- Otherwise, if `F` is defined in the same package as the class or interface in whose body the field access expression appears, then the field access is allowed unless the field is declared `private`.
- Otherwise, the field access is allowed only if the field is declared `public`.

If a field access is not allowed, a compile-time error results.

4.6 External Access

Name may be used from outside the scope of their declaration as follows:

- for package names: if the host system permits access.
- for type names declared in a different package: if the host system permits access to the package and the type is declared `public` (§6.1.1), (§9.6).
- for field names in the same package: if the field is not declared `private`
- for field names in a different package: if the host system permits access to the package, the type is declared `public` (§6.1.1), and either
  - the field is declared `public` (§9.6), or
  - the field is declared `protected` (§4.6) and the use is from within the declaration of a subclass of the said type.

Access control is determined by the compile-time (static) types of objects. A subclass may not be declared `public`, yet may be available outside the package where it is declared if it has a `public` superclass, since it can, for example, be assigned to a variable of this public type. Invocation of a `public` method of this variable's declared (compile-time) type may invoke a method of the (non-`public`) subclass if this method overrode(§6.4.1) a method of the `public` superclass(§9.7).

4.7 Rules about Names

If it denotes a declared entity the entity is either:

- a package,
- a type (which is either a class or an interface),
- a field variable, which is either `final` or not and `static` or not,
- a set of fields which are a group of one or more methods with the same name,
- a variable which is an argument of a method,
- a local variable which is either `final` or not, or a
- statement label.

If it denotes an undeclared entity it is either:

- a directory which are part of a package name, and which may contain further directories and/or a package,
- a value of a primitive type, either a integer value of type `byte`, `short`, `int`, or `long`, a floating-point value of type `float` or `double`, where the integer and floating-point type are collectively called arithmetic types, a `boolean` value of `true` or `false`, or a `char` value which is a Unicode character (note that `char` is not an arithmetic type),
- an assignable variable including elements of arrays,
- a reference to an object, which is known to be an instance of a specific class, or of one of this classes subclasses,
- a reference to an object, which is known to be an instance of some class supporting a specific interface,
- a reference to an object which is known to be an array of some type T,
- `null`, which is a reference to no object, or
- `void`, which is the result of a method which returns no value.

5 Program Structure

5.1 Packages and Directories

Java source code is organized into `package`s that have hierarchical names.

Each component of a package name is an identifier. In a typical Java implementation, package name components may be identified with directory names in a hierarchical file system, wherein each directory can contain zero or more subdirectories and/or the compilation units of a single package.

> *PackageName:*
> *PackageNameComponent*
> *PackageName* `.` *PackageNameComponent*
>
> *PackageNameComponent:*
> *Identifier*

5.2 Globally Unique Package Names

Java packages that are to be widely used should be given globally unique package names. This will allow them to be easily installed and catalogued. Java specifies a convention for generating globally unique package names.

You form a globally unique 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. 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 name is always written in all-uppercase ASCII letters and should be `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`.

Package names whose first component does not consist entirely of uppercase ASCII letters are reserved for local use, with the sole exception of the predefined portions of the Java language and system, which use the name `java`.

If you need to get a new Internet Domain Name, you can get an application form from `ftp://ftp.internic.net` and submit the complete forms to `domreg@internic.net`. If you want to check what the currently registered domain names are, you can `telnet` to `rs.internic.net` and use the `whois` facility.

5.3 Locating Packages on a Host System

In a typical hosted implementation of Java package names are transformed into a pathname, by concatenating the components of the package name, placing a file name separator between them.

Thus on UNIX systems, where the file name separator is `/`, the package name:

```
jag.fun.scrabble
```

is transformed to the directory name:

```
jag/fun/scrabble
```

and

```
COM.Sun.sunsoft.DOE
```

`is mapped to the directory name:`

```
COM/Sun/sunsoft/DOE
```

On UNIX, the CLASSPATH environment variable then provides a list of directories that provide roots for a search for a directory with this name.

If a package name component or class name contains a character which may not appear in a host file system's ordinary directory name, for example a Unicode character on a system which has only ASCII file names, then the character should be escaped by using a `@` character followed by one to four hexadecimal digits giving the Unicode code point of the escaped character, as in the `\uxxxx` escape (§1.1), so that:

```
COM.Sun.java.java2\u23b
```

is mapped to the directory name:

```
COM/Sun/java/java2@23b
```

5.4 Compilation Units

Each `package` consists of a number of compilation units.

CompilationUnit :
    PackageStatement$_{opt}$ *ImportStatements*$_{opt}$ *TypeDeclarations*$_{opt}$

PackageStatement:
    `package` *PackageName* `;`

TypeDeclarations:
    TypeDeclaration
    TypeDeclarations TypeDeclaration

TypeDeclaration:
    ClassDeclaration
    InterfaceDeclaration

If a compilation unit has no `package` statement, the unit is placed in a default package, which has no name. This is used on many systems to easily write fragments of Java code in the current directory in the file system.

A compilation unit declares zero or more types, at most one of which is declared `public`. This restriction makes it easy for the compiler and runtime system to find a named class within a package: if a type is `public`, its source code for its type `bar` would typically be found in a file `bar.java`, and the object code for the Java Virtual Machine in the file `bar.class`.

5.5 Compilation Unit Name Space

A compilation unit creates a name space which contains imported type names, imported package names, and all the type names declared in all the compilation units of the package. The imported and declared names must all be distinct.

5.6 Standard Imports

Each compilation unit automatically imports each of the type names defined in the predefined package `java.lang`, such as `Int`, `Float`, `Object`, `String`, and `NullPointerException`. The full specification of `java.lang` is given in §16.

5.7 The Import Statement

An `import` statement causes an name to denote a package or type which is declared elsewhere:

ImportStatement:
  PackageImportStatement
  TypeImportStatement
  TypeImportOnDemandStatement

PackageImportStatement:
  `import` *PackageName*

TypeImportStatement:
  `import` *PackageName* `.` *Identifier* `;`

TypeImportOnDemandStatement:
  `import` *PackageName* `. * ;`

A package import statement causes the named package to be known by the name of its last component. So after:

```
import java.io;
```

the types in `java.io` are known both as `io.`*name* and as `java.io.`*name*.

A type import statement causes the single `public` type from the named package to be available, thus

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

causes the name `Vector` to be interchangeable with the full name `java.util.Vector`.

It is a compile-time error for either a package or a type import to attempt to declare a name which is already declared by another import or as a type name in this package.

A type import on demand statement causes `public` types declared in the named package to be made ready to be imported as needed. Whenever Java is looking up a name in a package name space and the name is not found Java will look to see if the name is declared and `public` in a type imported by a type import on demand statement, and automatically import the name to the compilation unit's name space if it is found there.

It is a compile-time error for an undefined type name to be used and then found to be declared `public` in two or more packages which are being imported on demand.

The Java compiler and run-time keep track of packages and types within packages by their true names and are not fooled by having multiple ways of naming packages as introduced by `import`.

6 *Class and Interface Type Declarations*

A `class` declaration introduces a new reference type with an implementation that is derived from the implementation of another class called its *immediate superclass*; we say that a class *extends* its immediate superclass because it may provide additional implementation details. The single implementation inheritance Java provides for classes supports code reuse. Every object is an *instance* of some class.

The *superclass* relationship is the reflexive transitive closure of the *immediate superclass* relationship. Thus class `A` is a superclass of class `C` if and only if at least one of the following is true:

- `A` is the same as `C.`
- `A` is the immediate superclass of `C.`
- There is some class `B` such that `A` is a superclass of `B` and `B` is a superclass of `C`.

Class `A` is a *proper superclass* of class `B` if and only if `A` is a superclass of `B` but is not `B`.

Class `B` is an *immediate subclass* of class `A` if and only if class `A` is the immediate superclass of class `B`. Class `B` is a *subclass* of class `A` if and only if class `A` is a superclass of class `B`. Class `B` is a *proper subclass* of class `A` if and only if class `A` is a proper superclass of class `B`.

A variable whose declared type is a class type `C` may have as its value a reference to an object that is an instance of `C` or of any subclass of `C`.

An `interface` declaration introduces a new reference type that *specifies* a set of method signatures and some associated named constants, but does not specify an implementation. An interface may be declared to be an *immediate extension* of one or more other interfaces, meaning that it implicitly specifies all the method signatures and named constants of the interfaces it extends, perhaps adding method signatures or named constants of its own.

The *extension* relationship is the reflexive transitive closure of the *immediate extension* relationship. Thus interface `K` is an extension of interface `I` if and only if at least one of the following is true:

- `K` is the same as `I`.
- `K` is an immediate extension of `I`.
- There is some class `J` such that `K` is an extension of `J` and `J` is an extension of `I`.

Interface `K` is a *proper extension* of interface `I` if and only if `K` is a extension of `I` but is not `I`.

A class may be declared to *implement* one or more interfaces, meaning that any instance of the class implements all the method signatures specified by the interface. This (multiple) interface inheritance allows objects to support (multiple) common behaviors without sharing any implementation. If a class is declared to implement an interface, then all its subclasses (including the class itself) are implicitly considered to be declared to implement all interfaces (including the interface itself) of which that interface is an extension

A variable whose declared type is an interface type may have as its value a reference to an object that is an instance of any class that is declared to implement the specified interface. (It is not sufficient that the class happen to implement all the method signatures specified by the interface; the class or one of its superclasses must actually be declared to implement the interface.)

6.1 Class Declarations

A `class` declaration introduces a new reference type and specifies part or all of its implementation.

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

The *Identifier* is the name of the class; the fully qualified name of the class is *P*`.`*Identifier* where *P* is the fully qualified name of the package of the compilation unit in which the class is declared. The class declaration has a body that may contain field definitions. A class declaration may optionally have modifiers, specify its immediate superclass, and specify interfaces that it implements.

6.1.1 Class Modifiers

ClassModifiers:
ClassModifier
ClassModifiers ClassModifier

ClassModifier: one of
`abstract final public`

A class that is declared `abstract` may have `abstract` methods (§6.6.2). It is a compile-time error for a class containing an `abstract` method not to be declared `abstract`.

It is impossible to create an instance of an `abstract` class: you cannot create an instance of an `abstract` class with `new` (§9.8) or with the `newInstance` method of class `Class` (§16.1).

A class that is declared `final` can have no immediate subclasses, and therefore no proper subclasses, because it may not appear in the `extends` clause of another `class` declaration.

A class that is declared `public` can be accessed from other packages, either directly (§4.6) or in an import statement (§5.7). If a class lacks the `public` modifier, use of the class is limited to the package in which it is declared. At most one public class may be declared in each compilation unit (§5.4). A compilation unit may not contain both a public class and a public interface.

It is a compile-time error for a class to be declared both `final` and `abstract`.

6.1.2 Superclass Specification

The optional `extends` clause specifies the immediate superclass of the class being declared.

Super:
`extends` *TypeName*

The *TypeName* must name an accessible class that is not `final`. If the `extends` clause is omitted from a class declaration, then the class has the class `Object` (§16.1) as its immediate superclass. (Thus all classes are ultimately derived from this single root class, `Object`, forming a class hierarchy.)

It is a compile-time error for there to be a circularity that causes a class to directly or indirectly extend itself. For example, it is not permitted for `A` to be the immediate superclass of `B` and for `B` also to be a superclass of `A`.

6.1.3 *Implemented Interfaces*

The optional `implements` clause lists interfaces implemented by the class being declared.

Interfaces:
    `implements` *TypeNameList*

If the class `C` being declared is not `abstract`, every method signature that is declared in any of these interfaces must be defined by some superclass of `C` (possibly `C` itself).

6.1.4 *Class Body*

The class body consists of a (possibly empty) list of field declarations:

ClassBody:
    `{` *FieldDeclarations*$_{opt}$ `}`

FieldDeclarations:
    FieldDeclaration
    FieldDeclarations FieldDeclaration

Field declarations introduce new variables and methods to the class; some special methods are called constructors.

6.1.5 *Class Name Space*

A class introduces a new name space, built from inherited and declared field names.

The class inherits from its immediate superclass all the field declarations in the name space of the superclass, except that:

- Fields that are declared `private` are not inherited.
- Constructors are not inherited.
- If a name is declared as a field variable in the class being declared, no field variable of the same name is inherited from the superclass; a field variable declaration is said to *shadow* any field variable declaration in a superclass. This means the scoping rules for variables are different from the scoping rules for methods, because variables are shadowed rather than overridden (§6.4.9).

To these inherited fields are added the newly declared field declarations of the class itself.

6.1.6 *Multiple Declarations of a Single Name*

A variable may have the same name as a variable in the name space of its superclass, in which case the variable in the namespace of the superclass is not inherited but is said to be shadowed. (The shadowed variable can be accessed using the keyword `super` (§6.4.6) (§9.4) or the superclass's type name in a field access expression (§9.6).)

It is a compile-time error for a class to declare a method with the same name as a declared or inherited variable.

It is a compile-time error for a class to declare a variable to have the same name as a declared or inherited method.

It is a compile-time error for a class to declare two or more variables with the same name.

It is a compile-time error to declare two methods with the same name that take the same number of arguments and take the same declared type of argument in each argument position. More than one method with the same name may be declared, provided that, for any two methods of the same name, they either accept different numbers of arguments or take arguments of different declared type in at least one argument position. (Such polymorphism is called method overloading and is described in (§6.4.8).)

It is a compile-time error to declare two constructors that take the same number of arguments and take the same declared type of argument in each argument position. More than one constructor may be declared, provided that, for any two of the constructors, they either accept different numbers of arguments or take arguments of different declared type in at least one argument position. (Such polymorphism is called constructor overloading and is described in (§6.4.8).)

6.1.7 Visibility of Field and Class Names

Every field name is visible throughout the body of the class `C` in which the field is declared. It is also visible throughout the body of every subclass of `C` that inherits the name of the field (§6.1.5).

Every class type name is visible throughout the compilation unit in which the class is declared.

As an example, the following code is all correct:

```
class A {
     void a() {
          f.set(42);     // forward reference to f is okay
     }
     B f;                // forward reference to B is okay
}

class B {
     void set(long n) {
          this.n = n;    // See text below
     }
     long n;
}
```

In the assignment `this.n = n;` the first occurrence of `n` (after `this.`) is, in effect, a forward reference to the field named `n` declared two lines later; the second occurrence of `n` refers to the method parameter declared one line before the assignment.

6.2 Field Declarations

Fields are variables and methods. Constructors are methods of a special kind. Static initializers are used along with the initializers in the declarators of static field variables to define an implicit static class initialization method.

FieldDeclaration =
  FieldVariableDeclaration
  MethodDeclaration
  ConstructorDeclaration
  StaticInitializer

6.2.1 Field Access

Every field other than a static initializer may be declared to be `public`, `protected`, or `private` to control access to the declared entity. (This control allows the programmer to hide details of the implementation of an abstraction from the users of the abstraction.)

A `public` field is accessible anywhere the class name is accessible.

A `protected` field is throughout the package that contains the class in which the field is declared, and is also also accessible (unless shadowed) within the body of any subclass of that class.

A `private` field is accessible only within the class body in which the field is declared.

If none of the keywords `public`, `protected`, or `private` is specified, the field is throughout the package that contains the class in which the field is declared but is not accessible within the body of any subclass of that class if the subclass is declared in another package.

It is a compile-time error to mention more than one of `public`, `protected`, or `private` in a single field declaration. It is a compile-time error to mention the same modifier more than once in a single field declaration.

6.3 Variable Declarations

FieldVariableDeclaration:
  VariableModifiers$_{opt}$ *Type VariableDeclarators*

More than one field variable may be declared in a single field variable declaration by writing more than one declarator. The specified *Type* (§2) and modifiers apply to all the declarators in the declaration.

6.3.1 Variable Modifiers

VariableModifiers:
  VariableModifier
  VariableModifiers VariableModifier

VariableModifier: one of
```
public  protected  private
static  final  transient  volatile
```

The field access modifiers `public`, `protected`, and `private` are discussed in (§6.2.1).

A field variable that is not declared `static` is called an instance variable; there is actually a distinct variable known by that name associated with every instance of the class or its subclasses. Whenever a new instance of a class is allocated, a new variable associated with that instance is created for every field variable declared in that class or any of its superclasses. (Note that this is true even of shadowed field variables; a new variable is created and can be accessed, though not simply by its name alone.)

If a field variable is declared `static`, there is exactly one variable of that name, no matter how many instances (possibly zero) of the class are created. A static field variable is sometimes called a class variable because it is regarded as belonging to the class itself rather than to instances of the class.

A variable declared `final` must be assigned a value by including a variable initializer in its declarator. Any other attempt to assign to the variable results in a compile-time error.

Variables may be marked `transient` to indicate to low-level parts of the Java virtual machine that they are not part of the persistent state of an object. The `transient` attribute will to be used to implement some functions in later versions of the Java system. It is a compile-time error if a `transient` variable is also declared `final` or `static`.

A variable declared `volatile` is known to be modified asynchronously. The compiler arranges to use such variables carefully so that unsynchronized accesses to volatile variables are observed in some global total order. This means that variables which are declared `volatile` are reloaded from and stored to memory at each use, in a way that is coherent throughout a multiprocessor.

6.3.2 Variable Declarators

VariableDeclarator:
  DeclaratorName
  DeclaratorName = *VariableInitializer*

DeclaratorName:
  Identifier
  DeclaratorName `[]`

6.3.3 Variable Initializers

VariableInitializer:
  Expression
  `{` *ArrayInitializers*$_{opt}$ `,`$_{opt}$ `}`

ArrayInitializers:
  VariableInitializer
  ArrayInitializers `,`

If a variable declarator contains a variable initializer, then it behaves exactly as if it were an assignment (§9.20) to the declared variable. See also §7.3, which describes the treatment of array initializers.

If the declarator is for a `static` field variable, the variable initializer is computed and the assignment performed once, when the class is loaded (§6.7).

If the declarator is for an instance variable, the variable initializer is computed and the assignment performed as part of the execution of certain constructors for the class in which the instance variable is declared (§6.5.4).

If the declarator is for a local variable, the variable initializer is computed and the assignment performed as part of the execution of the variable declaration statement.

Examples of variable declarations:

```
int x, y;
float z = 1.0;
java.lang.String foo = "foo";
Object o = foo;
Exception e = new Exception();
double trouble[] = new double[27];
```

6.4 Method Declarations

A method is a chunk of executable code that can be invoked, possibly passing it certain values as arguments. Every method definition belongs to some class and must appear within the body of the class definition.

MethodDeclaration:
    MethodModifiers$_{opt}$ ResultType MethodDeclarator Throws$_{opt}$ MethodBody

6.4.1 Method Modifiers

MethodModifiers:
    MethodModifier
    MethodModifiers MethodModifier

MethodModifier: one of
```
public protected private
static
abstract final native synchronized
```

The field access modifiers `public`, `protected`, and `private` are discussed in (§6.2.1).

A method that is not declared `static` is called an instance method. Such a method can be invoked only relative to some particular object that is an instance of the method's class or one of its subclasses.

A method that is declared `static` is sometimes called a class method because it is regarded as belonging to the class itself rather than operating within instances of the class. A `static` method may refer to other fields and methods of the class by name only if they are also `static`.

Every `static` method is implicitly `final`. It is permitted but not required for the modifier `final` to appear redundantly along with the modifier `static` in a method declaration. Note that because every `static` method is implicitly `final`, it is not possible to override a `static` method.

Methods which are `static` are called class methods. Restrictions on `static` methods:

- Static methods can refer to other fields and methods of the class only if they are also `static`.
- A `static` method is implicitly `final`, so no overriding occurs on `static` methods.

A method can be declared `abstract`, in which case no implementation is provided for the method. The method declaration contains no body (a semicolon appears instead of a block). The declaration of an `abstract` method (call it `m`) must appear within an `abstract` class (call it `A`); otherwise a compile-time error results. Such a declaration merely defines the calling signature and return type for `m`. Every subclass of `A` that is not `abstract` must provide an implementation for `m`. To be precise, for every subclass `C` of the `abstract` class `A`, if `C` is not `abstract` then there must be some class `B` such that (1) `B` is a superclass of `C` (possibly `C` itself); (2) `B` is a subclass of `A`; (3) `B` is not `abstract`; and (4) `B` overrides `m`, thereby providing an implementation for `m` visible to `C`.

A `private` method cannot also be declared `abstract` (it is impossible to override a private method, so such a method could never be used).

A `static` method cannot also be declared `abstract` (a static method is implicitly final, so it is impossible to override a private method, so such a method could never be used).

A method that is declared `final` cannot be overridden; it is a compile-time error to attempt to override a `final` method. A `private` method is effectively `final`, even if not explicitly declared `final`, as are all methods declared in a `final` class, even if the methods are not explicitly declared `final`. In both these cases, it s permitted but not required for the modifier `final` to appear redundantly in such a method declaration. Note that if a method is `final` or effectively `final`, an optimizing compiler may be able to "inline" the method, that is, replace a call to the method with the code in its body.

A method can be declared `native`, in which case the method is implemented in a platform-dependent way, for example, in C or assembly language. Because the implmentation is not provided by Java Language code, the method declaration contains no body (a semicolon appears instead of a block). Native methods are otherwise like normal Java methods; they are inherited, may be `static` or not, may be `final` or not, may override or be overridden by non-native methods, and so on.

A `synchronized` method will acquire a monitor lock before it executes; the lock is per class if the method is `static`, per object otherwise.

6.4.2 *Result Type*

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.

A method that returns an array may be declared with the empty bracket pairs preceding the method name (as part of the result type) or following the argument list (as would be expected by a programmer accustomed to the declarator syntax of the C programming language), or with some bracket pairs in each place (cf. §7.2).

ResultType:
    Type
    `void`

MethodDeclarator:
    DeclaratorName (ParameterList$_{opt}$ *)*
    MethodDeclarator `[]`

6.4.3 *Parameter List*

The formal parameters of a method, if any, are specified by a list of comma-separated parameter specifiers. Each parameter specifier consists of a type and a name; as in other places in the Java Language, if the the type of the parameter is to be an array type, the empty bracket pairs may appear preceding the name (as part of the type specifier) or following the parameter name (as would be expected by a programmer accustomed to the declarator syntax of the C programming language), or with some bracket pairs in each place (cf. §7.2).

If a method has no parameters, only an empty pair of parentheses appears in the method declaration.

ParameterList:
    Parameter
    ParameterList `,` *Parameter*

Parameter:
    Type DeclaratorName

DeclaratorName:
  Identifier
  DeclaratorName `[]`

The parameters are local variables of the method, in the method name space, but declared outside the method's body.When the method is called, these local variables are freshly instantiated for the call; the values of the actual argument expressions are assigned to the fresh parameter variables before execution of the body of the method.

6.4.4 *Throws*

A method must declare any normal exceptions that can result from its execution:

Throws:
  `throws` *TypeNameList*

TypeNameList:
  TypeName
  TypeNameList `,` *TypeName*

If a method declaration contains a `throws` clause, it is a compile-time error if an exception can be thrown from the body of the method whose compile-time type is not assignable (§3.3) to either `Error`, `RunTimeException`, or one of the types mentioned in the `throws` clause. It is a compile-time error if any name in the `throws` clause does not name an accessible type that is a assignable to the type `Throwable` (perhaps `Throwable` itself).

If a method declaration does not contain a `throws` clause, it is a compile-time error if a normal exception can be thrown from the body of the method.

A method that overrides another method may not be declared to throw more exceptions than the overridden method. More precisely, if B is a subclass of A, a method declaration in B overrides a method declaration in A, and B has a throws clause, then

6.4.5 *The Body of a Method*

MethodBody:
  Block
  `;`

If a method is `abstract` or `native`, then its *MethodBody* must be a semicolon.

In all other cases, the *MethodBody* must be a block. If the method that is not `abstract` or `native` needs no executable code, then an empty block `{ }` should be used.

6.4.6 *Using* `this`, `super` *and Superclass Type Names*

Within the definition of an instance method, (one that is not `static`), the keyword `this` represents the current object. For example, an object may need to pass itself as an argument to another object's method:

```
class MyClass extends {
     void aMethod(OtherClass obj) {
          ...
          obj.Method(this);
          ...
     }
}
```

The `this` keyword is a reference to the current object; its type is the class containing the currently executing method.

Anytime a method refers to its own instance variables or methods, an implicit "`this.`" is in front of each reference:

```
class Foo {
     int a;
     ...
}
class Bar extends Foo {
     int a, b, c;
            ...
     void myPrint() {
          print(a + "\n");          // a == "this.a"
          print(super.a);           // Foo's a
          print(Foo.a);             // also Foo's a
     }
     ...
}
```

The `super` keyword is a reference to the superclass, i.e. equivalent to `((Foo)this)` in the example above.

A superclass's name may be used in a field access expression (§9.6) to access the superclass's fields and methods, usually because they are hidden or overridden.

6.4.7 *Using Superclass Names*

The names of the superclasses of the current type may also be used to access instance (non-static) methods and variables:

```
class A {
     Object x;
}
class B extends A {
     float x;
}
class C extends B {
     char x;
     void m() {
          char cx = x;        // C's x is a char
          float bx = B.x;     // B's x is a float
                              // ... super.x would also work here
          Object ax = A.x;    // A's x is a object
     }
}
```

6.4.8 *Method Overloading*

Java allows overloading (polymorphic) method declarations: there can be more than one method with the same name visible within a class provided the methods differ in the number of parameters or in the types of the parameters.

When a method is to be called, the number of actual arguments and the compile-time types of the actual arguments are used at compile time to determine which method definition will actually be invoked. If there is a possibility that the method may be overridden, a dynamic method dispatch may also be used at run time. See §9.7 for a complete description of compile-time method selection and run-time dispatch.

6.4.9 *Method Overriding*

Java allows overriding method declarations: a class may inherit from one of its superclasses a method with a certain name and a certain number of parameters with certain types and yet also declare a method of the same name with the same number of parameters, with corresponding parameters having the same type.

The access modifier of the overriding method must provide at least as much access as the overridden method:

- If the declaration of the overridden method does not contain any of the modifiers `public`, `protected`, or `private`, then the overriding method must not be `private`.
- If the overridden method is `protected`, then the overriding method must be `public` or `protected`.
- If the overridden method is `public`, then the overriding method must be `public`.

The return type of an overriding method must be assignable (§3.3) to the return type of the overridden method.

An overridden method can be invoked within a class containing the overriding method by using the `super` keyword in a method call:

```
setThermostat(...)         // refers to the overriding method
super.setThermostat(...)   // refers to the overridden method
```

<u>This example is incomplete... it should also talk about using the superclass type name to get around the override.</u>

A `private` method is not inherited and hence is not available to be overridden. It is permitted for a proper subclass of a class containing a `private` method to declare a method of the same name with the same number of parameters, with corresponding parameters having the same type; but this is not considered to be a case of overriding, so the method in the proper subclass need obey the restrictions imposed by overriding; for example, the return type of the method in the subclass need not bear any relation to the return type of the private method.

It is *not* permitted for two method declarations within the *same* class to have the same name, the same number of parameters, and corresponding parameters of the same type; this situation is a compile-time error.

6.5 Constructor Method Declarations

A constructor is a special kind of method that is used to initialize a newly created object. Constructors have a special declaration syntax and a special invocation syntax.

ConstructorDeclaration:
  ConstructorModifier$_{opt}$ *ConstructorDeclarator* *Throws*$_{opt}$ *ConstructorBody*

ConstructorDeclarator:
  Typename `(` *ParameterList*$_{opt}$ `)`

The *TypeName* in the *ConstructorDeclarator* must name the class that contains the constructor declaration. A constructor has no separate name of its own. (At the level of the Java Virtual Machine, every constructor has the special name `<init>`. This name is supplied by the Java compiler. Because it is not a valid identifier, this name cannot be used directly by a Java programmer.)

A constructor has no separately declared result type. For the purpose of using `return` statements, the return type of a constructor is implicitly `void`.

If a class contains no constructor declarations, then a default constructor is implicitly and automatically provided. The default constructor takes no arguments; it simply calls the superclass constructor `super()` with no arguments and then performs instance variable initialization. (As a special case, the default constructor for class `Object` does not invoke `super()` because `Object` has no supeclass.) It is a compile-time error if the superclass does not have a constructor that takes no arguments.

Unlike ordinary methods, constructors are not inherited by subclasses (§6.1.5).

6.5.1 *Constructor Modifiers*

ConstructorModifier: one of
  `public protected private`

The field access modifiers `public`, `protected`, and `private` are discussed in (§6.2.1). Note that constructors are not referenced directly by name, but through the use of allocation expressions (§9.8) or the `newInstance` method of class `Class`; the access restrictions indicated by field access modifiers (or their absence) apply to these indirect means of reference.

Note that a class can be designed to prevent code outside the class definition 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`.

6.5.2 *Parameter List*

The parameter list for a constructor is identical in structure and behavior to the parameter list for an ordinary method (§6.4.3).

6.5.3 *Throws*

The throws clause for a constructor is identical in structure and behavior to the throws clause list for an ordinary method (§6.4.4).

6.5.4 *The Body of a Constructor*

ConstructorBody:
  `{` *ExplicitConstructorCallStatement*$_{opt}$ *BlockBody* `}`

ExplicitConstructorCallStatement:
  `this (` *ArgumentList*$_{opt}$ `) ;`
  `super (` *ArgumentList*$_{opt}$ `) ;`

The first statement of a constructor may be an explicit call to another constructor of the same class, written as `this` followed by a parenthesized argument list, or an explicit call to a constructor of the immediate superclass, written as `super` followed by a parenthesized argument list (this is one of the two places in the Java language where the keyword `super` has a special meaning and cannot be replaced by a cast of `this` to the type of the immediate superclass; see also §9.7). Note that an explicit constructor call statement may appear *only* as the first statement of a constructor body and nowhere else.

If an explicit constructor call is not present and the constructor begin defined is not for class `Object`, then the constructor body is implicitly assumed to begin with the statement "`super();`", that is, a call to the superclass constructor without arguments. Therefore every constructor for every class except `Object` effectively begins with a call to some other constructor, either for the same class or for its immediate superclass.

An explicit constructor call statement may not contain references to instance variables of the object being created.

A call `super(...);` to a superclass constructor, whether it actually appears as an explicit constructor call statement or is implicitly assumed, performs an additional implicit action after a normal return of control from the superclass constructor: all the instance variables that have initializers are initialized at that time. More precisely:

> for every instance variable declared in the class containing the call,
> taken in the order in which the instance variables appear in the class declaration:
> if that variable has an initializer *and*
> either the initialization expression is not a constant expression *or*
> its value is not the standard default value for the variable,
> then the initialization expression is executed and
> its value is assigned to the instance variable.

A call `this(...);` to another constructor in the same class does not perform this additional implicit action.

Taking all these rules into account, a simple inductive argument shows that when an object of any given class type is created, constructors for all the superclasses of that class will be called; the body of the constructor for Object will be executed first, and in general each constructor will be executed only after the constructors for its superclasses have been executed. All the instance variables of the object will be initialized; each initializer will be executed exactly once per object creation; when an initialization expression is executed, all instance variables declared in superclasses and all instance variables preceding it in the same class declaration will already have been initialized.

```
class ColoredPoint {
     double x, y;
     Color c = blue;
     Point new(float xVal, float yVal) {
          // implicit super() call here.
          // implicit assignment of blue to c here.
          x = xVal; y = yVal;
     }
     Point new() {             // default constructor
          this(0.0, 1.0);      // default value
     }
}
```

It is a compile-time error for instance variable initializations to have a forward dependency. For example, the following code:

```
class Z {
     int i = j + 2;
     int j = 4;
}
```

results in a compile-time error.

6.5.5 *Object Creation*

An object can be created by an allocation expression (§9.8), which performs these steps:

- A new object is created of the specified type. As the new object is created, all its instance variables are initialized to their standard default values (§2.3).
- The appropriate constructor for the newly created object is invoked on whatever actual arguments appear in the allocation expression. For a use of the `newInstance` method, the constructor that takes no arguments is invoked.
- After the constructor has returned, a reference to the newly created and initialized object is the value of the allocation expression.

An object can also be created by a call to the `newInstance` method of class `Class` (§16.1), which performs these steps:

- A new object is created of the type represented by the class object for which the `newInstance` method was invoked. As the new object is created, all its instance variables are initialized to their standard default values (§2.3).
- The constructor for the newly created object is invoked with no actual arguments.
- After the constructor has returned, a reference to the newly created and initialized object is returned as the value of the call to the `newInstance` method. The compile-time type of this reference will be `Object`, which is the declared return type of the `newInstance` method, but its run-time type will be the type represented by the class object for which the `newInstance` method was invoked.

6.6 Automatic Storage Management and Finalization

When an object is no longer referenced, this may be detected by the automatic storage management of the Java system. Automatic storage management typically makes use of so-called "garbage collector" algorithms. Once it has been determined that an object is no longer referenced, the storage it occupies may be reclaimed immediately and recycled for other use—unless the dynamic object has a finalizer.

A class may request *finalization* of its instances by implementing a non-static method named `finalize` that takes no arguments and returns no value:

```
void finalize();
```

Such a method is called a *finalizer*. This method must not be declared with any method modifiers (§6.4).

When an object is no longer referenced, but has a finalizer, the Java system will (eventually) call the finalizer before reclaiming the storage occupied by the object.

After an object has been finalized, no further reclamation action is taken until the automatic storage management determines again that it is no longer referenced. This is necessary because the finalizer may have *resurrected* the object by making it accessible once again, perhaps by storing a reference to the object into some accessible variable. The finalizer is never called more than once for each object, so an object can be resurrected at most once.

When an object is no longer referenced, but has a finalizer, but the finalizer has already been called for that object, the storage occupied by the object may be reclaimed immediately and recycled for other use.

If an uncaught exception occurs during the finalization, the exception is ignored. The finalizer will not be called again for that object.

The Java language makes no guarantees about when or in what order objects will be finalized.

The Java language makes no guarantees about which thread will invoke the finalizer for any given object. It is guaranteed, however, that the thread that invokes the finalizer will not be holding any user-visible synchronization locks when the finalizer is called.

The purpose of finalizers is to provide a chance free up resources (such as file descriptors or operating system graphics contexts) that are owned by objects but cannot be accessed directly and cannot be freed automatically by the automatic storage management. Simply reclaiming an object's memory by garbage collection would not guarantee that these resources would be reclaimed.

6.7 Class Loading and Initialization

6.7.1 Class Loading

A class is loaded when it is needed, either because it is implicitly needed by another class, or because its is explicitly requested to be loaded using a `classLoader` (§16.4) or the `Class.forName` method of the class `Class` (§16.1). This is sometimes called dynamic loading.

When a class is loaded, storage is allocated for its static variables. A class object (an instance of the class `Class`) is also allocated to represent the class. The class is then initialized.

(At the level of the Java Virtual Machine, a class is initialized by invoking its class initialization method with no arguments. The class initialization method has the special name `<clinit>`. This name is supplied by the Java compiler. Because it is not a valid identifier, this name cannot be used directly by a Java programmer.)

6.7.2 Static Variable Initialization

The `static` variables of a class may be initialized by initializers in their declarations or by one or more static initializers, or both.

StaticInitializer:
    `static` *Block*

A static initializer is simply some code that is executed when the class containing it is loaded. Static initializers and variable initializers are executed in the order in which they appear in the class declaration. For example, when the class

```
class Z {
     static int a = 1;
     static double b;
     static {
          a++;
          c = 7;
     }
     static int c = 2;
     static Window d = new Window();
     static { b = Math.cos(Math.PI/4.0); }
}
```

is loaded, the following initialization steps occur in the order shown:

- The variable `a` is set to `1`.
- The first static initializer is executed, incrementing `a` to `2` and setting `c` to `7`.
- The variable c is then set to `2` (the value `7` is lost).

- A new `Window` is allocated and assigned to variable `d`.
- The variable `b` is set to the value of the expression `Math.cos(Math.PI/4.0)`.

It is a compile-time error for static variable initializations to have a forward dependency. For example, the following code:

```
class Z {
     static int i = j + 2;
     static int j = 4;
}
```

results in a compile-time error.

It is a compile-time error for static initializers or initializers for static variables to contain references to instance variables of the class in whose declaration they appear.

The static initializer code may call static methods of the class being loaded and use other classes that have already been loaded, but it is a run-time error for there to be a circularity, i.e. a situation where a class A needs class B to have been loaded to run its static initializer and vice-versa. If this mutual dependency is detected at compile-time a compile-time error results, if it is detected at run-time a `ClassCircularityException` is thrown.

6.8 Interface Declarations

An interface declares a type consisting of a set of methods and constants without specifying its implementation.

Java programs can use interfaces to make it unnecessary for related classes to share a common abstract superclass or to add methods to `Object`. This provides the power of multiple interface inheritance to classes without the messiness of multiple implementation inheritance.

InterfaceDeclaration:
InterfaceModifiers$_{opt}$ `interface` *Identifier* *ExtendsInterfaces*$_{opt}$ *InterfaceBody*

6.8.1 Interface Modifiers

InterfaceModifiers:
InterfaceModifier
InterfaceModifiers InterfaceModifier

InterfaceModifier: one of
`public abstract`

An interface that is declared `public` can be accessed from other packages, either directly (§4.6) or in an import statement (§5.7). If an interface lacks the `public` modifier, use of the interface is limited to the package in which it is declared. At most one public interface may be declared in each compilation unit (§5.4). A compilation unit may not contain both a public interface and a public class.

Every interface is implicitly `abstract`. It is permitted but not required to specify the `abstract` modifier.

6.8.2 Subinterfaces and the `extends` *Clause*

If an `extends` clause is provided then the interface being declared extends each of the other named interfaces and therefore implicitly includes the methods and constants (unless shadowed) of each of the other named interfaces. Any class that `implements` the declared interface is also considered to implement all the interfaces that this interface extends.

ExtendsInterfaces:
  `extends` *TypeName*
  ExtendsInterfaces `,` *TypeName*

Each *TypeName* in the extends clause of an interface declaration must name an interface.

It is a compile-time error for there to be a circularity that causes an interface to directly or indirectly extend itself.

Note that there is no analogue of the class `Object` for interfaces; that is, while every class is an extension of class `Object`, there is no single interface of which all interfaces are extensions.

6.8.3 *Body of an Interface*

The body of an interface is much like the body (§6.1.4) of an abstract class (§6.1.1).

InterfaceBody:
  `{` *FieldDeclarations* `}`

However, the body of an interface may not contain constructor declarations (§6.5) or static initializers (§6.7.2).

6.8.4 *Variable Declarations in Interfaces*

Every field variable in the body of an interface is implicitly `static` and `final`. It is permitted but not required to specify the `static` modifier, the `final` modifier, or both `static` and `final` for such variables. Every field variable in the body of an interface must have an initializer and the initialization expression must be a constant expression (§9.22).

Every variable declaration in the body of a `public` interface is implicitly `public`. It is permitted but not required to specify the `public` modifier for such methods.

A variable declaration in an interface body may not include any of the modifiers `synchronized`, `transient`, or `volatile`.

6.8.5 *Method Declarations in Interfaces*

Every method declaration in the body of an interface is implicitly `abstract`. Its body must be represented by a semicolon, not a block. It is permitted but not required to specify the `abstract` modifier for such methods.

Every method declaration in the body of a `public` interface is implicitly `public`. It is permitted but not required to specify the `public` modifier for such methods.

A method declaration in an interface body may not include any of the modifiers `final`, `native`, `static`, or `synchronized`.

6.9 A `class` and `interface` Example

```
interface Storing {
     void freezeDry(Stream s);
     void reconstitute(Stream s);
}
```

```
class Image implements Storing {
     ...
     void freezeDry(Stream s) {
          // JPEG compress image before storing
          ...
     }
     void reconstitute (Stream s) {
          // JPEG decompress image before reading
          ...
     }
}
class StorageManager {
     Stream stream;
     ...
     // Storing is the interface name
     void pickle(Storing obj) {
          obj.freezeDry(stream);
     }
}
```

The `StorageManager` class requires that the argument to `pickle` implement the `Storing` interface but can make no other assumption about how `obj` is implemented.

7 Arrays

Java arrays are objects, are dynamically allocated, and may be assigned to variables of type `Object`. All methods of class `Object` may be invoked on an array.

Java arrays are single-dimensional. An array is an object that contains a number of variables. (This number may be zero.) These variables have no names; instead they are referenced using nonnegative integer 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 integers 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*`[]`.

There are no multi-dimensional arrays as such. However, the component type of an array may itself be an array type. The subarrays themselves have components, of course, and so on. 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. (Note that there is one situation in which an element of an array can be an array: if the element type is `Object`, then some or all of the elements may be arrays, because every array can be cast to class `Object`.)

Like all objects in Java, arrays must be explicitly allocated. However, there are two different ways to allocate an array. The `new` operator may be used in the usual way to allocate a fresh array. In addition, a special "array initializer" syntax may be used on the right-hand side of the = in a declaration of an array variable.

7.1 Array Types

An array type is notated (§2.2.4) by writing the name of the element type followed by some number of empty pairs of square brackets `[]`. The number of bracket pairs indicates the depth of array nesting.

Array types may be used in declarations and in casts.

7.2 Declarations of Array-valued Variables

Declaring a variable of array type does not allocate an array object and therefore does not 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 may allocate an array, a reference to which then becomes the initial value of the variable.

Here are some examples of declarations of array variables that do not allocate an array:

```
int[]     ai;            // array of integer
short[][] as;            // array of array of short;
Object[]  ao,            // array of object
          otherAo;       // array of object
short     s,             // scalar short
          as[],          // array of short
          aas[][];       // array of array of short
```

Here are some examples of declarations of array variables that allocate 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", };
```

Note that `[]` may appear as part of the 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[];
```

Programmers may prefer to avoid putting some brackets in the declaration type and some in the declarator as a matter of style.

7.3 Array Initialization

An array may be allocated by using an array initializer in place of an initialization expression in a declarator. This is written as a pair of braces ("{}") enclosing a comma-separated list of expressions. The length of the constructed array will equal the number of expressions. Each expression specifies a value for one array component. Each expression must be assignment-compatible with the array's component type. 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.

ArrayInitializer:
  `{` *ElementInitializers*$_{opt}$ `,`$_{opt}$ `}`
ElementInitializers:
  Element
  ElementInitializers `,` *Element*
Element:
  Expr
  ArrayInitializer

A redundant trailing comma may appear after the last expression in an array initializer.

7.4 Array Length

An array's length is not part of its type. As a consequence, over the course of time a single variable of array type may contain references to arrays of different lengths.

Every array has a `.length` field, which is a `final` variable; it may be examined but may not be changed by assignment. Once an array object is allocated, its length never changes. If it it desired to make an array variable refer to an array of different length, it is necessary to allocate or otherwise identify another array of the desired size and then assign a reference to that other array to the variable.

All array accesses are checked at run-time; an attempt to use an index that is less than zero or not less than the length of the array causes an `ArrayIndexOutOfBoundsException` to be thrown (§9.5).

7.5 Array Indexing

Arrays may be indexed by `int` values (§9.5); `short`, `byte`, or `char` values may also be used as they are subjected to unary arithmetic promotion (§3.5) and become `int` values. Arrays may not be indexed by `long` values.

7.6 Array Allocation and Reclamation

Array use example:

```
/*
 * Method to return an n-by-m array of bytes with a
 * given initial value.
 */
byte[][] makeByteArray( int n, int m, byte initialValue )
{
     byte[][] newArray = new byte[n][m];
     for( int i=0 ; i < newArray.length ; i++ )
          for( int j=0 ; j < newArray[i].length ; j++ )
               newArray[i][j] = initialValue;
     return newArray;
```

Array use example:

```
/*
 * Method to return a triangular array of bytes with a
 * given initial value. Element [i][j] exists only if j < i.
 */
byte[][] makeTriangularByteArray( int n, byte initialValue )
{
     byte[][] newArray = new byte[n][];
     for( int i=0 ; i < newArray.length ; i++ )
          newArray[i] = new byte[i];
          for( int j=0 ; j < newArray[i].length ; j++ )
               newArray[i][j] = initialValue;
     return newArray;
}
```

Array use example:

```
/*
 * Method to return a triangular array of integers filled
 * with Pascal's triangle. Element [i][j] exists only if j <= i
 * and equals i!/(j!(i-j)!).
 */
int[][] makePascalTriangle( int n )
{
     byte[][] result = new byte[n][];
     for( int i=0 ; i < result.length ; i++ )
          result[i] = new byte[i+1];
          result[i][0] = 1;
          for( int j=1 ; j < i; j++ )
               result[i][j] = result[i-1][j-1] + result[i-1][j];
          result[i][i] = 1;
     return result;
}
```

7.7 Arrays versus Strings

An array of `char` is not a `String`. Note that a String does not have assignable components, whereas the character components of an array of characters can be assigned to.

Neither `Strings` nor arrays of `char` are automatically terminated by `'\u0000'` (the `NUL` character). In this respect Java differs from C.

8 Blocks and Statements

Except as described, statements are executed in sequence. Statements are executed for their effect, and do not have values.

Java requires that variables be clearly initialized before use. We (will soon) give, in this section and in the next section on Expressions, the algorithm which a Java compiler is required to use to determine if a variable has been clearly initialized. A compiler may not vary from this algorithm, because that would affect what Java programs are legal.

8.1 Blocks

The body of a method and the body of a static initializer are both blocks, which are a sequence of local variable declarations and statements.

Block:
　　`{` *LocalVariableDeclarationsAndStatements* `}`
LocalVariableDeclarationsAndStatements:
　　LocalVariableDeclarationOrStatement
　　LocalVariableDeclarationsAndStatements LocalVariableDeclarationOrStatement
LocalVariableDeclarationOrStatement:
　　LocalVariableDeclarationStatement
　　Statement

8.2 Local Variable Declarations

A local variable declaration statement introduces a new identifier into a block; it has the form:

LocalVariableDeclarationStatement:
　　TypeSpecifier VariableDeclarators

The identifier is not allowed to already be declared as a local variable or label, or as a variable which is a formal argument to a method or constructor.

If the identifier declared by this statement was previously declared as a field (variable or method) or type name, then the other declaration is hidden for the remainder of the block, after which it resumes its force. The keyword `this` can be used to access a hidden field `x`, in an expression of the form `this.x`.

The initializations of the declared variables are done each time the local variable declaration statement is executed. It is a compile-time error if a variable is used when the compiler cannot determine whether the variable will be dynamically initialized before use, using the standard algorithm (to be) described here.

8.3 Statements

Statements fall into several groups:

Statement:
EmptyStatement
LabeledStatement
ExpressionStatement `;`
SelectionStatement
IterationStatement
JumpStatement
SynchronizationStatement
ExceptionStatements

8.4 Empty Statement

An empty statement does nothing.

EmptyStatement:
`;`

8.5 Labeled Statements

Statements may carry label prefixes.

LabeledStatement:
Identifier `:` *Statement*
`case` *ConstantExpression* `:` *Statement*
`default` `:` *Statement*

The first form declares the identifier as the label of the statement, and has as its scope the current block. Labels used with the `continue` statement must be on iterations statements.

The identifier is not allowed to already be declared as a local variable or label, or as a variable which is a formal argument to a method or constructor.

If the identifier declared by this statement was previously declared as a field (variable or method) or type name, then the other declaration is hidden for the remainder of the block, after which it resumes its force.

Statement labels can be used only in labeled `break` and `continue` statements within this block.

The `case` labels and `default` labels may occur only in `switch` statements (§8.7).

8.6 Expression Statements

Most statements are expression statements, which have one of the forms:

ExpressionStatement:
Assignment `;`
PreIncrement `;`
PreDecrement `;`
PostIncrement `;`
PostDecrement `;`
MethodCall `;`
AllocationExpression `;`

All side effects from the expression are completed before the next statement is executed.

Unlike C and C++ Java restricts the forms of expressions which are valid statements to catch errors. The programmer can assign the value of any other expression to a variable to make such an expression into a statement.

It is legal for a result of a method which is not declared `void` to be ignored.

Java forbids the expression statement to begin with a cast.

8.7 Selection Statements

Selection statements choose one of several flows of control:

SelectionStatement:
`if (` *Expression* `)` *Statement*
`if (` *Expression* `)` *Statement* `else` *Statement*
`switch (` *Expression* `)` *Block*

8.7.1 The `if` *Statement*

In both forms of the `if` statement, the expression, which must have a `boolean` type, is evaluated, including all side effects. If it evaluates to `true` then the first substatement is executed. In the second form the second substatement is executed if the expression evaluates to `false`. The `else` ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if` in the same block.

8.7.2 The `switch` *Statement*

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression. The type of the expression must be `char`, `byte`, `short` or `int`.

The substatement controlled by a switch is a block. Any top-level statement within the block may be labeled with one or more `case` labels, and at most one top-level statement may be labeled with a `default` label.

The controlling expression and the case constants are converted to `int`.

No two of the (promoted) case constants associated with the same switch may have the same value; this applies whether the case is on `char` or an integral type.

When the `switch` statement is executed, its expression is evaluated, including all side effects, and compared with each case constant. If one of the case constants is equal to the value of the expression, control passes to the statement of the matched `case` label. If no case matches, and there is a `default` label, control passes to the labeled statement. If no case matches, and there is no `default`, then none of the substatements of the switch is executed.

The `case` or `default` labels in themselves do not alter the flow of control, which continues unimpeded across such labels. To exit from a `switch`, a `break` or other jump statement is typically used.

8.8 Iteration Statements

Iteration statements specify looping:

IterationStatement:
    `while ( ` *Expression* ` ) ` *Statement*
    `do ` *Statement* ` while ( ` *Expression* ` ) ;`
    `for ( ` *ForInit Expression$_{opt}$* ` ; ` *ForIncr$_{opt}$* ` ) ` *Statement*

ForInit:
    ExpressionStatements `;`
    LocalVariableDeclarationStatement

ForIncr:
    ExpressionStatements

ExpressionStatements:
    ExpressionStatement
    ExpressionStatements `,` *ExpressionStatement*

Note that the *ForInit* part ends with a semicolon.

8.8.1 *The* `while` *Statement*

In the `while` statement the substatement is executed repeatedly until the value of the expression, which must be of type `boolean`, becomes `false`. The test, including all side effect from evaluation of the expression, takes place before each execution of the substatement. The substatement may be executed zero times.

8.8.2 *The* `do` *Statement*

In the `do` statement the substatement is executed repeatedly until the value of the expression, which must be of type `boolean`, becomes `false`. The test, including all side effect from evaluation of the expression, takes place after each execution of the substatement. The substatement is executed at least once.

8.8.3 *The* `for` *Statement*

The `for` statement

    `for ( ` *ForInit Expression$_{opt}$* ` ; ` *ForIncr$_{opt}$* ` ) ` *Statement*

is equivalent to

```
ForInit
while ( Expression_opt ) {
        Statement
        ForIncr ;
}
```

except that a `continue` in Statement will execute *ForIncr* before re-evaluating *Expression.* Thus the first statement specifies initialization for the loop; the first expression specifies a test, made before each iteration, such that the loop is exited when the expression becomes `false`; the second expression often specifies incrementing that is done after each iteration.

Either or both of the expressions may be omitted. A missing *Expression* makes the implied `while` equivalent to `while(true)`.

8.9 Jump Statements

Jump statements transfer control unconditionally:

JumpStatement:
`break` *Identifier*$_{opt}$ `;`
`continue` *Identifier*$_{opt}$ `;`
`return` *Expression*$_{opt}$ `;`
`throw` *Expression* `;`

In any case where a jump statement causes control to bypass a `finally` part of a `try` statement, the non-local control transfer pauses while the `finally` part is executed, and continues if the `finally` part finishes normally (§8.10.2).

8.9.1 *The* `break` *Statement*

An unlabeled `break` statement transfers control to the end of the enclosing iteration (`for`, `do`, `while`), or `switch` statement. If an identifier is provided, it must be the label of an arbitrary enclosing statement. Control passes to the statement following the terminated statement, after executing any required `finally` clauses, provided the `finally` clauses all complete normally.

8.9.2 *The* `continue` *Statement*

The `continue` statement may occur only in an iteration statement and causes control to pass to the loop-continuation point of an iteration statement, breaking out of the statement governed by the iteration but not out of the iteration itself. If the optional identifier is provided, then it must be a label of an enclosing iteration statement, otherwise, the nearest enclosing looping statement is continued.

If control passes any `finally` clauses they are executed before continuing at the continuation point, and control ultimately reaches the continuation point only if all such `finally` clauses complete normally.

More precisely, in each of the statements:

```
outer:                 outer:                  outer:
while (foo) {          do {                    for (;;) {
    // ...                 // ...                  // ...
    //continue here        //continue here         //continue here
}                      } while (foo);          }
```

a `continue` not contained in an enclosing iteration statement continues at the `continue here` point. A `continue` giving the label `outer` would continue at the `continue here` point (and specifically not fall in at the top of the iteration as a `goto` statement would in C or C++.)

8.9.3 *The* `return` *Statement*

A method, constructor, or static initializer returns to its caller by the `return` statement. If this causes control to pass any `finally` clauses they are executed before the `return` occurs, and the `return` continues to operate only as long as all of the `finally` clauses complete normally.

A return statement with an expression can be used only in methods that are declared to return a value, that is methods which are not declared `void`. If required, the expression is converted, as in an assignment to a variable which has as its type the return type of the function.

A return statement without an expression can be used in methods that are declared to not have a result type, constructors and static initializers.

8.9.4 *The* `throw` *Statement*

A `throw` statement signals a run-time exception. Its argument must be an object type, and is conventionally a subclass of `Exception`.

Normal execution is suspended while a suitable exception handler is sought for the exception. Each enclosing statement which is not a `try` is terminated, and any `finally` clauses that are passed by are executed. The exception propagation continues until a `catch` clause is found whose formal argument has a type which is a superclass of the type of the argument expression. Processing then continues as described in §8.10.2.

8.10 Guarding Statements

Guarding statements establish conditions or contexts during the execution of a substatement:

GuardingStatement:
  `synchronized (` *Expression* `)` *Statement*
  `try` *Block Finally*
  `try` *Block Catches*
  `try` *Block Catches Finally*
Catches:
  Catch
  Catches Catch
Catch:
  `catch (` *Argument* `)` *Block*
Finally:
  `finally` *Block*

8.10.1 *The* `synchronized` *Statement*

A `synchronized` statement establishes a critical expression. The value of the expression must be a reference to an object (which may be an array).

The `synchronized` statement acquires the (single) lock associated with the object, waits for the lock to be free if necessary, executes the governed statement, and then releases the lock.

8.10.2 *The* `try` *statement*

A try statement executes the block in the `try` part, which is the scope of the exception handlers established by any `catch` clauses.

If an exception occurs during execution of the statement in the `try` part which is not handled by a nested handler, then the exception will cause termination of the execution of the `try` part.

Any `catch` clauses associated with the `try` will then be examined. Each catch clause has a single formal argument of class or interface type, and will handle any exception which can legally be assigned to this argument. This allows subclasses of type `Exception` to define categories of exceptions in a natural way.

Exception handler types are compared in order: the first `catch` clause supporting a legal assignment accepts the exception, receiving the object which is associated with the exception in the actual variable which is its argument. This variable has as its scope the block of the `catch`. When the `catch` block completes execution, execution continues with the `finally` part, if any, or with the next statement in order after the `try`.

A `finally` clause is used to ensure that the block governed by `finally` is executed after the statement governed by `try` and `catch`, no matter how control leaves the `try` or `catch` part.

After the `finally` code is executed, control transfers out of the `try` statement. Normally, the control transfer destination is that determined by the event which caused the `try` statement to be terminated: fall-through, the execution of a `break`, `continue`, or `return`, or the propagation of an exception. But if the `finally` code executes a jump statement causing another unconditional control transfer outside of its block or causes another uncaught exception to be thrown, then the original jump statement is abandoned, and the new unconditional control transfer or exception is processed.

8.11 Unreachable Statements

It is a compile-time error if a statement cannot be executed because it is unreachable. The precise meaning of this remark will be explained in a future version of this document.

9 Expressions

9.1 Value of an Expression

Expressions are used in Java to indicate variables and to compute values. The execution of an expression produces one of three results:

- a value
- a variable (in C this would be called an *lvalue*)
- nothing (the expression is `void`)

An expression produces nothing if and only if it is a method call that invokes a method whose return type is `void`. Such an expression can be used only as an expression statement, because every other context in which an expression can appear requires the expression to produce a value or a variable. An expression statement that is a method call may also call a method whose return type is not void; the value returned by the method is quietly discarded.

An expression that produces a value may not appear as the left-hand operand of any assignment operator (§9.20) or as the operand of a ++ or −− operator (§9.9.1, §9.7.1). These contexts require an expression that produces a variable.

All other contexts where an expression may appear require a value, but the expression may produce either a variable or a value; if the expression produces a variable, then the value of that variable is used, and we simply speak of the value of the expression.

The execution of an expression can also produce side effects, because expressions may contain embedded method calls as well as embedded assignment, ++, and −− operators.

Each expression occurs in the declaration of some type which is being declared, either in its static initializer, in a constructor declaration, or in the code for a method.

9.2 Type of an Expression

Every expression has a compile-time type. The rules for determining the type of an expression are explained separately below for each kind of expression. The value of an expression will always be compatible with the compile-time type of the expression, just as the value stored in a variable will always be compatible with the compile-time type of the variable.

If the compile-time type of an expression is a class type `C`, then the value of the expression will be either `null` or a reference to an instance of some subclass `R` of `C` (which may be `C` itself). If the compile-time type of an expression is an interface type `I`, then the value of the expression will be either `null` or a reference to an instance of some class `R` that implements the interface `I`. If `C` is `Object`, then the value of the expression may also be an array of some array type `R`. In any of these cases, if value of the expression is not null then we say that `R` is the run-time type of the value. If the compile-time type of an expression is a primitive type, then the run-time type of the value is the same as the compile-time type of the expression.

Note that an expression whose compile-time type is a class type `F` that was declared `final` is guaranteed to produce a value whose run-time type is `F`, because `final` types have no subclasses.

These are the only places in the Java language where the run-time type of a value affects the course of execution in a manner that cannot be deduced from the compile-time type:

- In a method or constructor call(§9.7). The particular method is used for a call `o.m(...)` is chosen based on the methods which are part of the class or interface which is the static type of `o`. The run-time type of `o` participates because a subclass may override(§6.4.9) a specific method already defined in a parent class so that this overriding method is called first; this method may or may not choose to further call the original overridden `m` method.
- In a narrowing cast (§9.9.2). The value of an expression may be cast to a type that is narrower than the compile-time type of the expression; this requires a run-time check that throws an error if the run-time type of the value is not compatible with the narrower type.
- With `instanceof` (§9.14). An expression whose compile-time type is a class, interface, or array type may be tested using `instanceof` to find out whether the run-time type of its value is compatible with some narrower type.
- Assigning to an array component of reference type. Such an assignment may require a narrowing conversion at run time and so may require a run-time check.
- In a `catch` clause, where an exception is caught only if the run-time type of the exception is `instanceof` the formal argument type (§8.10.2).

Thus a Java run-time type error can occur only in these situations:

- In a narrowing cast, the value's run-time type is not compatible with the cast type.
- In an assignment to an array component of reference type, the run-time type of the value to be assigned is not compatible with the array component type.
- An exception is not caught.

9.3 Evaluation Order

Java guarantees that the operands to operators appear to be evaluated from left-to-right. Specifically:

- The left-hand operand of a binary operator appears to be fully executed before any part of the right-hand operand is executed. 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.
- In an array reference, the expression to the left of the brackets appears to be fully executed before any part of the expression within the brackets is executed. For example, in the (admittedly monstrous) expression `a[(a=b)[3]]`, the expression `a` is fully executed 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 executed. This old array is then subscripted by a value that is element `3` of another array copied from `b` into `a`.
- In a method call for an object, there is an expression whose value is an object; this expression appears to the left of the dot, method name, and left parenthesis of the method call. This expression appears to be fully executed before any part of any argument expression within the parentheses is executed.
- In a method call or allocation expression, there may be one or more argument expressions within the parentheses, separated by commas. Each argument expression appears to be fully executed before any part of any argument expression to its right.
- In an allocation expression, there may be one or more dimension expressions, each within brackets. Each dimension expression appears to be fully executed before any part of any dimension expression to its right.

It is not necesssarily recommended that Java code rely crucially on this specification; code is usually clearer when each expression contains at most one side effect, as its outermost operation. These rules are imposed principally to promote portability of Java programs, no matter how they are coded.

Java also guarantees that every operand of an operator appears to be fully executed before any part of the operation itself is performed. In particular, the operands of an increment, decrement, or compound assignment operator appear to have been fully executed before the compound assignment operator fetches the value of the variable to be updated. For example, in the compound assignment operation `a+=(a=3)`, the resulting value of `a` is guaranteed to be `6`, because the assignment of `3` to a occurs before the += operation fetches `a` in order to add its right-hand operand to it. (Note that this example therefore behaves slightly differently from `a=a+(a=3)`, where the old value of `a`—the value of the left-hand operand of the + operation—must be fetched and remembered before the assignment of `3` to `a` occurs. Note also that both these examples have undefined behavior in C, according to the ANSI/ISO standard.)

Java implementations therefore must respect the order of execution 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, 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. Note also that 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 certain large values of `x` for which the first expression will produce infinity (because of overflow) but the second expression will produce a finite result.

In contrast, integer addition and multiplication *are* provably associative in Java; for example `a+b+c` 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.

9.4 Primary Expressions

Primary expressions include names, literals, expressions in parentheses, allocation expressions, array references, field references, and method calls.

PrimaryExpression:
  Name
  NotJustName

NotJustName:
  AllocationExpression
  ComplexPrimary

ComplexPrimary:
  Literal
  `(` *Expression* `)`
  ArrayAccess
  FieldAccess
  MethodCall

Name:
  QualifiedName
  `this`
  `super`
  `null`

QualifiedName:
  Identifier
  QualifiedName `.` *Identifier*

A name may be a simple identifier or a qualified identifier. When used as an expression, such a name must be the name of a variable.

If a simple name refers to a local variable or method parameter, then if the variable is `final`, the result of the expression is the value of the specified variable; but if the variable is not `final`, the result of the expression is the variable itself. This distinction matters because it implies that only non-final variables may appear as the left-hand operand of an assignment operator (§9.20). In either case, the compile-time type of the expression is the declared type of the variable.

If a simple name does not refer to a local variable or method parameter, it may indicate a field access (§9.6.2). A qualified name cannot refer to a local variable or method parameter, but may indicate a field access (§9.6.3).

A literal (§1.7) denotes either a primitive value or a reference to an object that is an instance of class `String`.

A parenthesized expression is a primary expression that has the same value and compile-time type as the contained expression.

9.4.1 `this` *and* `super`

The keywords `this` and `super` may be used only within the body of a non-static method. They have the same value, which is a reference to the object for which the method was invoked; but they have different compile-time types.

The compile-time type of `this` is the class (call it `C`) within which the method body appears. The compile-time type of `super` is the immediate superclass of `C`, as indicated in the `extends` clause of the definition of `C`. The keyword `super` may not appear within the class `Object`, which has no superclass. The run-time type of the value, of course, may be `C` or any subclass of `C`, unless the class `C` is `final` (and therefore has no proper subclasses), in which case the run-time type is necessarily `C`.

There are two situations, involving method invocation (§9.7) and constructor invocation(§6.5.4), in which the keyword `super` plays a special role. In all other situations, the keyword `super` is entirely equivalent to a cast (§9.10.7) of the keyword `this` to the type of the immediate superclass.

9.4.2 `null`

The keyword `null` denotes a privileged polymorphic value representing the absence of a reference. Its compile-time type is, in effect, a subtype of every reference type.

9.5 Array Access

ArrayAccess:
    Name `[` *Expression* `]`
    ComplexPrimary `[` *Expression* `]`*r*

A primary expression followed by an index expression in square brackets is an array access. The compile-time type of the primary expression must be an array type (call it `T[]`, an array whose components are of type `T`); its value will then be either `null` or a reference to an array. The index expression undergoes unary arithmetic promotion (§3.5); the promoted type must be `int`.

If, at run-time, the value of the primary expression is `null`, a `NullPointerException` is thrown.

If, at run-time, the value of the primary expression is not `null`, but the value of the index expression is less than zero, or greater than or equal to the length of the array, an `ArrayIndexOutOfBoundsException` is thrown.

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.

Note that, for syntactic reasons, the primary expression in an array access cannot be an unparenthesized allocation expression.

9.6 Field Access

Fields of an object, array, class, or interface may be accessed in several ways.

In all cases, if the field is `final`, the result of the field access is the value of the specified field; if the field is not `final`, the result of the field access is a variable, namely the specified field itself. The compile-time type of the result is the declared type of the field.

9.6.1 Field Access through an Object or Array Reference

FieldAccess:
    PrimaryExpression `.` *Identifier*

A primary expression followed by a dot followed by an identifier indicates field access.

The compile-time type of the primary expression must be a reference type `T`. The identifier is resolved as a field variable (§4.5) within type `T`, and must be the name of a field variable of the class, interface, or array type, or a compile-time error results. (Note that an array type has exactly one named field variable: `length` (§7.4).)

If the field is `static`, the field access refers to a field variable associated with the class or interface whose definition contains the declaration of the field. If the field is not `static`, the field access must occur within the definition of a method that is not `static`, and it refers to a field variable within the current object (as declared in some class that is necessarily `T` or one of its superclasses).

9.6.2 *Field Access through a Simple Name*

If a primary expression is an identifier that does not name a local variable, then it is resolved as a field variable (§4.5) within the class whose definition contains the primary expression. In effect, a simple name *xxx* is treated as if it has been written `this.`*xxx*.

If the field is `static`, the field access refers to a field associated with the class or interface whose definition contains the declaration of the field. If the field is not `static`, the field access must occur within the definition of a method that is not `static`, and it refers to a field within the current object (as declared in the class whose definition contains the primary expression or one of its superclasses).

9.6.3 *Field Access through a Qualified Name*

If a primary expression is a qualified name of the form

TypeName `.` *Identifier*

where the *TypeName* is itself a simple or qualified name that names a class or interface, then it the *Identifier* is resolved as a field variable (§4.5) within the specified class or interface.

If the field is `static`, the field access refers to a field associated with the class or interface whose definition contains the declaration of the field. If the field is not `static`, the field access must occur within the definition of a method that is not `static`, and the indicated type must be the class in whose body the field access appears, or one of its superclasses; the field access refers to a field within the current object; in this case, the qualified name *TypeName*`.`*Identifier* is treated as if it had been written `((`*TypeName*`)this).`*Identifier*.

9.7 Method Calls

MethodCall:
    MethodAccess `(` *ArgumentList*$_{opt}$ `)`

MethodAccess:
    Name
    PrimaryExpression `.` *Identifier*

ArgumentList:
    Expression
    ArgumentList `,` *Expression*

A method call is a method access followed by parentheses that surround a possibly empty, comma-separated list of expressions, called the arguments. A method access has the same form as a field access but must refer to a field that is a method rather than a variable. Resolving a method name at compile time is more complicated than resolving a field variable because of the possibility of method overloading. Invoking a method at run time is also more complicated than accessing a field variable because of the possibility of method overriding.

For a method call to be correct and unambiguous there must be a method definition at compile time that is both applicable and most specific.

A method definition is applicable to a method call if all these requirements are satisfied::

- The declared name of the method is the same as the field name in the method call.

- The method definition is accessible from the method call by the rules of name resolution (§4.5).
- The number of parameters in the method definition equals the number of arguments in the method call, and
- Each actual argument in the method call is assignable (§3.3) to the corresponding parameter as declared in the method definition.

A method `m`, declared in class `T` with `n` parameters having types `T1, ..., Tn` is more specific than another method, also named `m` but declared in class `U` with `n` parameters having types `U1, ..., Un`, if and only if `T` is assignable to `U` and `Tj` is assignable to `Uj` for all `j` from `1` to `n`. This implies, by the way, that if we declare variables

```
T t; T1 t1; ... Tn tn;
```

then for any values of these variables, the code

```
((U)t).m(t1,...,tn)
```

could invoke the second method without type errors. Of course, within the body of the first method, `this` has the type `T` and its parameters have types `T1, ..., Tn`. This leads to the simple and intuitive notion that a method defined in class `T` is more specific than a method of the same name defined in class `U` if it can call the second method simply by casting `this` to type `U` and passing all its parameters as arguments.

At compile time, there is some set of methods applicable to a method call. If this set is empty (there is no applicable method), a compile-time error results. Otherwise, there must be a single method definition in the set that is more specific than all others; if not, the method call is considered ambiguous, and a compile-time error results.

If there is a single most specific method definition, it is called the *compile-time definition* for the method call; its name and the compile-time types of the parameters in the definition constitute the *signature* for the method call. The declared return type for this method definition is used as the compile-time type of the method call.

At run time, the method invocation proceeds as follows. If the method access requires computing a reference value (which may be an implicit occurrence of `this`), that subexpression is executed first; if the method is not static, the resulting value is called the *target object* and will be available within the called method as the value of `this` and of `super`. Then the argument expressions are evaluated in order, from left to right, and their values are assigned to the parameters of the method (in a new activation frame). Finally, a method definition is located and actually invoked.

If the method is `static`, then it cannot be overridden (because every `static` method is implicitly `final`). The method definition that was determined to be most specific at compile time is the definition invoked at run time.

If the method is `private`, then it cannot be overridden. The method definition that was determined to be most specific at compile time is the definition invoked at run time.

If the *MethodAccess* appearing before the parenthesized argument list is of exactly the form `super.`*MethodName*, this is considered a request to run the method named *MethodName* that is visible in the namespace of the immediate superclass of the class within whose body the method call appears. Any overriding methods are bypassed; the method definition that was determined to be most specific at compile time is the definition invoked at run time. (This is the one of the two situations in the Java language where `super` is not equivalent to a cast of `this` to the type of its immediate superclass; see also §6.5.4.)

If the method is neither `static` nor `private` and the *MethodAccess* is not of the form `super.`*MethodName*, then *dynamic method lookup* occurs. The lookup process starts from the class that is the run-time type of the target object and from there works its way up the chain of superclasses (if the target object is an array, the lookup process starts, and ends, at the class `Object`). As soon as a class is found with a method definition that matches the signature for the method call determined at compile time, that method definition is invoked. The lookup process must succeed, because the definition located at compile time will be found if no overriding definition is found in some subclass.

The result of a method call is the value returned by the invoked method. If the declared return type of the method is `void`, then there is no result; a method call to such a method may appear only as a top-level expression (as an expression statement or in the header of a `for` statement).

9.8 Allocation Expressions

The `new` operator attempts to create an object or array of a specified type:

AllocationExpression:
    `new` *TypeName* `(` *ArgumentList*$_{opt}$ `)`
    `new` *TypeName* *DimExprs* *Dims*$_{opt}$

TypeName:
    TypeKeyword
    QualifiedName

TypeKeyword: one of
    `boolean char byte short int float long double`

ArgumentList:
    Expression
    ArgumentList `,` *Expression*

DimExprs:
    DimExpr
    DimExprs DimExpr

DimExpr:
    `[` *Expression* `]`

Dims:
    `[ ]`
    Dims `[ ]`

A `new` operator will raise an `OutOfMemoryException` if there is insufficient memory available.

9.8.1 Allocating New Objects

In the first form of allocation expression, the *TypeName* must name a class type that is not `abstract`. This class type is the compile-time type of the allocation expression.

The types of the arguments in the argument list, if any, are used to match against all the constructor methods, declared in the class type or any of its superclasses, using the matching rules for method calls (§9.7). As in method calls, a compile-time method matching error results if there is not a single constructor that is both applicable and most specific.

The value of the first form of `new` is a newly created object of the specified class type that has been initialized by first initializing every instance variable of the object to its standard default value (§2.3) and then invoking the constructor method for that object on the arguments.

9.8.2 *Allocating New Arrays*

The second form of new allocates a new array whose elements are of the type specified by the *TypeName*; in this case the *TypeName* may name any type, even an abstract type or primitive type. The compile-time type of the allocation expression is an array type that can be described by deleting the `new` keyword and every *DimExpr* expression from the allocation expression; for example, the compile-time type of the allocation expression

```
new double[3][3][]
```

is

```
double[][][]
```

The expression in each *DimExpr* undergoes unary arithmetic promotion (§3.5); the promoted type must be `int`. If, at run-time, the value of any *DimExpr* expression is less than zero, `ArrayNegativeSizeException` is thrown.

If a single *DimExpr* appears, a single-dimensional array is allocated of the specified length. Each component of the array is initialized to its standard default value (§2.3).

Multidimensional arrays are implemented as arrays of arrays. If `A` is an *N*-dimensional array whose elements are of type `T`, then `A[i]` is a reference to an (*N*–1)-dimensional array whose elements are of type `T`.

If an array allocation expression contains *N DimExpr* expressions, then it effectively executes a set of nested loops of depth *N*–1 to allocate the implied arrays of arrays. For example, the allocation:

```
float[][] matrix = new float[3][3];
```

is roughly equivalent to:

```
float[][] matrix = new float[3][];
for (int i = 0; i < matrix.length; ++i)
     matrix[i] = new float[3];
```

And

```
String[][][][][] fivedims = new String[6][8][10][12][];
```

is equivalent to:

```
String[][][][][] fivedims= new String[6][][][][];
for (int d1 = 0; d1 < fivedims.length; d1++) {
     fivedims[d1] = new String[8][][][];
     for (d2 = 0; d2 < fivedims[d1].length; d2++) {
          fivedims[d1][d2] = new String[10][][];
          for (d3 = 0; d3 < fivedims[d1][d2].length; d3++) {
               fivedims[d1][d2][d3] = new String[12][];
          }
     }
}
```

leaving the fifth dimension, which would be arrays containing the actual references to `String` objects, initialized only to `null`.

A multidimensional array need not have the same length arrays at each level; thus a triangular matrix may be allocated by:

```
float triang[][] = new float[100][];
for (int i = 0; i < triang.length; i++)
     triang[i] = new float[i+1];
```

There is, however, no way to get this effect with a single allocation expression.

9.9 Postfix Expressions

PostfixExpression:
  PrimaryExpression
  PostIncrement
  PostDecrement

PostIncrement:
  PrimaryExpression ++

PostDecrement:
  PrimaryExpression −−

9.9.1 Postfix Increment Operator ++

A primary expression followed by a ++ operator is a postfix increment expression. The primary expression must denote a variable of an arithmetic type. The compile-time type of the postfix increment expression is the type of the variable. The value `1`, converted to the type of the variable, is added to the value of the variable and stored back into the variable. The value of the postfix increment expression is the value of the variable *before* the new value is stored.

9.9.2 Postfix Decrement Operator −−

A primary expression followed by a −− operator is a postfix decrement expression. The primary expression must denote a variable of an arithmetic type. The compile-time type of the postfix decrement expression is the type of the variable. The value `1`, converted to the type of the variable, is subtracted from the value of the variable and stored back into the variable. The value of the postfix decrement expression is the value of the variable *before* the new value is stored.

9.10 Unary Operators

Expressions with unary operators group right-to-left:

UnaryExpression:
  PreIncrement
  PreDecrement
  `'+'` *UnaryExpression*
  `'-'` *UnaryExpression*
  UnaryExpressionNotPlusMinus

PreIncrement:
  ++ *PrimaryExpression*

PreDecrement:
  −− *PrimaryExpression*

UnaryExpressionNotPlusMinus:
PostfixExpression
`'~'` *UnaryExpression*
`'!'` *UnaryExpression*
CastExpression

CastExpression:
`(` *TypeKeyword* `)` *UnaryExpr*
`(` *TypeExpression* `)` *UnaryExpressionNotPlusMinus*

The grammar is a bit more complicated than one might expect in order to avoid syntactic problems with expressions such as `(p)-q` and `(p)--q`.

In the case of `(p)-q`, it is not evident whether this is a binary subtraction of `q` from `p` or a cast of a unary negation of `q`. It depends on whether or not `p` names a type or a variable. Because the Java built-in unary negation operation can return only values of primitive type, and because values of primitive type can be cast only to other values of primitve type, the Java language treats `(p)-q` as a cast of a unary negation if and only if `p` is the name of a primitive type. This permits such familiar constructions as `(short)-3` without requiring additional parentheses.

As for `(p)--q`, the difficulty for a simple grammar is that it is not clear whether `(p)` is a cast operator or an expression without looking *two* tokens to the right, to the token after the `--` operator. Because most automatic parser generators support only one-token lookahead, the Java language forbids this construction. One can always write `(p)(--q)` instead.

9.10.1 *Prefix Increment Operator ++*

A primary expression preceded by a ++ operator is a prefix increment expression. The primary expression must denote a variable of an arithmetic type. The compile-time type of the prefix increment expression is the type of the variable. The value `1`, converted to the type of the variable, is added to the value of the variable and stored back into the variable. The value of the prefix increment expression is the value of the variable *after* the new value is stored.

9.10.2 *Prefix Decrement Operator --*

A primary expression preceded by a `--` operator is a prefix decrement expression. The primary expression must denote a variable of an arithmetic type. The compile-time type of the prefix decrement expression is the type of the variable. The value `1`, converted to the type of the variable, is subtracted from the value of the variable and stored back into the variable. The value of the prefix decrement expression is the value of the variable *after* the new value is stored.

9.10.3 *Unary Plus Operator +*

The value of the operand of the unary + operator must be a primitive value of an arithmetic type. The operand undergoes unary arithmetic promotion . The compile-time type of a unary plus expression is the promoted type of the operand. The result is the promoted value of the operand.

9.10.4 Unary Minus Operator −

The value of the operand of the unary − operator must be a primitive value of an arithmetic type. The operand undergoes unary arithmetic promotion . The compile-time type of a unary plus expression is the promoted type of the operand. The result is the arithmetic negation of the promoted value of the operand.

For integer values, negation is the same as subtraction from zero. Because Java uses two's-complement representation for integers, and the range of two's-complement values is not symmetric, the negation of the maximum negative `int` or `long` results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown. Note that, 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` equals `+0.0`, but `−x` equals `−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.

9.10.5 Bitwise Complement Operator ~

The value of the operand of the unary ~ operator must be a primitive value of an integral type. The operand undergoes unary arithmetic promotion . The compile-time type of a unary bitwise-complement expression is the promoted type of the operand. The result is the bitwise complement of the promoted value of the operand. Note that for all integer values `x`, `~x` equals `(−x)−1`.

9.10.6 Logical Complement Operator `!`

The value of the operand of the `!` operator must be a primitive value of type `boolean`. The result is a value of type `boolean`. The result is `true` if the operand value is `false` and `false` if the operand value is `true`.

9.10.7 Casts

A unary expression that does not begin with +, −, ++, or −− and that is preceded by a cast operator (parentheses enclosing the name of a type) is called a cast, and causes conversion of the value of the expression to the named type. The compile-time type of the cast expression is the type named in the cast operator. The value of the cast expression is the value of the unary expression after conversion to the specified type.

Not all casts are permitted by the Java language; see §3.4. Some casts result in an error at compile time; for example, it is not permitted to cast a primitive value 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. Yet other casts cannot be proven always correct or always incorrect at compile time; such casts require a test at run time. An `IncompatibleTypeException` is thrown if a cast is found to be impermissible at run-time. ??? Or is it `ClassCastException` ?

9.11 Multiplicative Operators

The so-called "multiplicative operators" `*`, `/`, and `%` have the same precedence and are syntactically left-associative (they group left-to-right).

MultiplicativeExpression:
    UnaryExpression
    MultiplicativeExpression `*` *UnaryExpression*
    MultiplicativeExpression `/` *UnaryExpression*
    MultiplicativeExpression `%` *UnaryExpression*

Each operand of the multiplicative operators must be a value of primitive arithmetic type. Binary arithmetic promotion is performed on the operands (§3.6); the compile-time type of the multiplicative expression is the promoted type of the 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.

9.11.1 Multiplication Operator *

The binary `*` operator performs multiplication, producing the product of its operands. Multiplication is a commutative operation. Integer multiplication is associative, but floating-point multiplication is not always 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. 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 arithmetic:

- If either operand is NaN, the result is NaN.
- If neither operand is NaN, the sign of the result is positive if both operands have the same sign, 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, with the sign producing rule just given.
- In the remaining cases, where neither an infinity or NaN is involved, the product is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent, we say the operation underflows; the result is then a zero of appropriate sign. Note that the Java language requires support of gradual underflow as defined by IEEE 754.

Despite the fact that overflow, underflow, or loss of precision may occur, execution of a multiplication operator `*` never throws a run-time exception.

9.11.2 Division Operator /

The binary `/` operator performs division, producing the quotient of its operands. The left-hand operand is the dividend and the right-hand operand is the divisor.

Integer division rounds toward 0; that is, the quotient produced for integer operands n and d is an integer value q that is negative if and only if exactly one of n and d is negative and whose magnitude is as large as possible while satisfying $|d \cdot q| \leq |n|$. 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 governed by the rules of IEEE arithmetic:

- If either operand is NaN, the result is NaN.
- If neither operand is 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, with the sign producing rule just given.
- Division of a finite value by an infinity results in a signed zero, with the sign producing rule just given.
- Division of a zero by a zero results in NaN; division of zero by any other finite value by a zero results in a signed zero, with the sign producing rule just given.
- Division of a non-zero finite value by a zero results in a signed infinity, with the sign producing rule just given.
- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, the quotient is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent, we say the operation underflows; the result is then a zero of appropriate sign. Note that the Java language requires support of gradual underflow as defined by IEEE 754.

Despite the fact that overflow, underflow, division by zero, or loss of precision may occur, execution of a floating-point division operator / never throws a run-time exception.

9.11.3 *Remainder*

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.

Integer remainder produces a result value such that `(a/b)*b+(a%b)` is equal to `a`. Note that 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 is 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; moreoer, 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.

The result of a floating-point remainder operation as computed by the `%` operator is *not* the same as the so-called "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. The Java language defines % on floating-point operations to behave in a manner analogous to that of the Java integer remainder operator; this may be compared with the C library function `fmod`. The IEEE 754 remainder operation may be computed by the Java library routine `Math.IEEEremainder`.

The result of a Java floating-point remainder operation is governed by these rules:

- If either operand is NaN, the result is NaN.
- If neither operand is 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 a dividend n and 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.

Despite the fact that division by zero may occur, execution of a floating-point remainder operator `%` never throws a run-time exception. Note that overflow, underflow, or loss of precision cannot occur.

9.12 Additive Operators

The so-called "additive operators" + and − have the same precedence and are syntactically left-associative (they group left-to-right).

AdditiveExpression:
  MultiplicativeExpression
  AdditiveExpression + *MultiplicativeExpression*
  AdditiveExpression − *MultiplicativeExpression*

If either operand, or both, of a + operator has type `String`, the operation is string concatenation. If exactly one operand is of type `String`, the other is converted to type `String` before the concatenation is performed.

Otherwise, each operand of the additive operators must be a value of primitive arithmetic type. Binary arithmetic promotion is performed on the operands (§3.6); the compile-time type of the additive expression is the promoted type of the 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.

9.12.1 Addition and Subtraction Operators (+ and −) for Arithmetic Types

The binary + operator performs addition when applied to two operands of arithmetic type, producing the sum of its operands. Addition is a commutative operation. Integer addition is associative, but floating-point addition is not always 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 will not be the same as the sign of the mathematical sum of the two operand values.

The result of a floating-point addition is governed by the 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 saem sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent, we say the operation underflows; the result is then a zero of appropriate sign. Note that the Java language requires support of gradual underflow as defined by IEEE 754.

The binary − operator performs subtraction when applied to two operands of arithmetic 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, however, that for floating-point operands, subtraction from zero is not the same as negation, because if `x` is `+0.0`, then `0.0-x` equals `+0.0`, but `-x` equals `-0.0`. For integer values, subtraction from zero is the same as negation.)

Despite the fact that overflow, underflow, or loss of precision may occur, execution of an arithmetic additive operator never throws a run-time exception.

9.12.2 String Concatenation Operator +

If the value of either operand of + is a reference to an object of type `String`, then the + operator behaves as if it converts the other operand to a reference to a `String` object (if it is not already a reference to a `String` object), and returns a reference to an object of type `String` that is the concatenation of the two operand strings. (The qualification "as if" is present because an implementation may choose to perform the conversion and concatenation in one step so as to avoid allocating and then discarding an intermediate `String` object.)

A operand that is not a reference to a `String` is converted to a `String` according to the compile-time type of the operand:

- If `String`, but the value is `null`, then the literal string `"null"` is the result.
- If a reference type other than `String`:
  - If the value is `null`, then the literal string `"null"` is the result.
  - Otherwise, the `toString` method of the object is invoked with no arguments; this method returns a reference value of type `String`, which is used as the result of the conversion unless it is the null value, in which case the literal string `"null"` is the result. (The class `Object` defines such a `toString` method, so this method is always available.)
- If a primitive integral type, the value is converted to a string representing the value in decimal notation, preceded by a − sign if the value is negative. If the value is nonzero, the first digit is nonzero; if the value is zero, a single digit `0` is produced.
- If a primitive floating-point type, has floating-point type then this value is converted ??? **TO BE SPECIFIED: FORMAT, TYPE SUFFIX?, ... dtoa? %g?**
- If type `char`, then the operand value is converted to a `String` of length one containing the operand value as its single character.

- If type `boolean`, then the result is either the literal string `"true"` or the literal string `"false"`.

9.13 Shift Operators

The shift operators include the left shift <<, the signed right shift >>, and the 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:
    AdditivieExpression
    ShiftExpression << *AdditiveExpression*
    ShiftExpression >> *AdditiveExpression*
    ShiftExpression >>> *AdditiveExpression*

Each operand of a shift operator must be a value of primitive integral type. Binary arithmetic promotion (§3.6) is *not* performed on the operands; rather, unary arithmetic promotion (§3.5) is performed oneach operand separately, and the compile-time 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 `&` (§9.17) 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`, 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 `&` (§9.17) with the mask value `0x3f`. The shift distance actually used is therefore always in the range 0 to 63, inclusive.

The value of `n<<s` is `n` left-shifted `s` bit positions; this is equivalent (even if overflow occurs) to multiplication by two to the power `s`.

The value of `n>>s` is `n` right-shifted `s` bit positions with sign-extension. The resulting value is $\lfloor n/2^s \rfloor$ (For non-negative value 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, 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<<(k-s-1))`, where `k` is `32` if the type of the left-hand operand is `int` and `64` if its type is `long`.

9.14 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` *TypeSpecifier Dims*$_{opt}$

9.14.1 *Numerical Comparison Operators <, <=, >, and >=*

Each operand of a numerical comparison operator must be a value of primitive arithmetic type. Binary arithmetic promotion is performed on the operands (§3.6); the compile-time type of the comparison expression is `boolean`. 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.

Floating-point comparison is performed in accordance with IEEE 754:

- If either operand is NaN, 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`.

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 for 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`.

9.14.2 *Type Comparison Operator* `instanceof`

The compile-time type of the left-hand operand of the `instanceof` operator must be a class or interface type; otherwise a compile-time error occurs.

The second operand of the `instanceof` operator is not really an expression, but instead must specify a reference type.

The `instanceof` operator returns `false` if the first operand denotes `null`. (The rationale is that while `null` can be assigned to a variable of any reference type, it is not an object and therefore not an “instance” of a type.)

The `instanceof` operator returns `true` if the run-time type of the first operand allows it to represent an object of the second operand’s type, and `false` otherwise. Equivalently, if the (run-time type of the) first operand can be cast to the second type without raising a `ClassCastException` then `instanceof` is `true` else `false`. The prototypical use of `instanceof` is:

```
if (thermostat instanceof MeasuringDevice) {
     MeasuringDevice dev = (MeasuringDevice)thermostat;
     ...
}
```

Here we may know that `thermostat` is a `device` (a superclass of `MeasuringDevice`), but may not know if it is, more specifically, a `MeasuringDevice`. The `instanceof` operator protects us from the `ClassCastException` which would result if the `thermostat` could not represent a `MeasuringDevice`, i.e. could not be assigned to `dev`.

If there is no possibility that the `instanceof` can return true then a compile-time error results. This can occur, for example, in:

```
class A extends Object;
class B extends Object;
A a;
...
if (a instanceof B) {          // impossible and illegal
     B b = (B)a;               // always an exception
}
```

Given

```
T t;
if (t instanceof U) {
     U u = (U)t;
     ...
}
```

with T and U distinct, then

- if `U` is a class type, then `instanceof` is checking that the run-time type of t is a subclass of `U`, which can be true only when T is a superclass of U.
- if `U` is an interface type, then `instanceof` is checking that `U` is implemented by the run-time type of t or by a superclass of the run-time type of T. If this is known to be `true` and the check unnecessary, then this is a compile-time error. Since any class can implement an interface this can never be proven `false` until run-time.

9.15 Equality Operators

The relational operators are syntactically left-associative (they group left-to-right), but this fact is only slightly useful; for example, `a==b==c` parses as `(a==b)==c`, and because the type of `a==b` is always `boolean`, `c` must therefore be of type `boolean`.

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 arithmetic type, or two operands of boolean type, or two operands of reference type. In all cases, `a!=b` has the same result as `!(a==b)`.

9.15.1 *Numerical Equality Operators* `==` *and* `!=`

If one operand of an equality operator is a value of primitive arithmetic type, the other operand must also be a value of (possibly some other) primitive arithmetic type. Binary arithmetic promotion is performed on the operands (§3.6); the compile-time type of the equality expression is `boolean`. If the promoted type of the operands is `int` or `long`, then an integer equality test is performed; if this promoted type is `float` or `double`, then a floating-point equality test is performed. The numeric equality operators are commutative.

Floating-point equality testing is performed in accordance with IEEE 754:

- If either operand is NaN, the result of == is `false` but the result of `!=` is `true`. (Indeed, the test `x!=x` is true if and only if the value of `x` is NaN.)
- Positive zero and negative zero are considered equal. Therefore `-0.0==0.0` is `true`, for example.
- Otherwise, two distinct floating-point values are considered unequal. 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, and otherwise 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, and otherwise is `false`.

All other cases, including any equality comparisons involving `boolean` variables or values, result in compile-time errors.

9.15.2 *Boolean Equality Operators* `==` *and* `!=`

If one operand of an equality operator is a value of type `boolean`, the other operand must also be a value of type `boolean`. The compile-time type of the equality expression is `boolean`. The boolean equality operators are commutative and 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 `^` (§9.16.2) when applied to boolean operands.)

9.15.3 *Object Equality Operators* `==` *and* `!=`

If one operand of an equality operator is a value of a reference type, the other operand must also be a value of a reference type. The compile-time type of the equality expression is `boolean`. The object equality operators are commutative.

It is a compile-time error if it is impossible to convert the compile-time type of one operand to the compile-time type of the other by a casting conversion (§3.4). (The run-time values of the two operands would necessarily be unequal.)

The result of == is `true` if the operands are both `null` or both refer to the exact same object or array; otherwise the result is `false`.

The result of != is `false` if the operands are both `null` or both refer to the exact same object or array; otherwise the result is `true`.

Note that while == may be used to compare references of type `String`, the equality test determines whether or not the two operands refer to the same exact `String` object. The result will be `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 call `s.equals(t)`.

9.16 Bitwise and Logical Operators

The bitwise 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 operator is syntactically left-associative (each groups left-to-right). Each operator is both commutative and associative.

AndExpression:
  EqualityExpression
  AndExpression `&` *EqualityExpression*
ExclusiveOrExpression:
  AndExpression
  ExclusiveOrExpression `^` *AndExpression*
InclusiveOrExpression:
  ExclusiveOr
  InclusiveOrExpression | *ExclusiveOrExpression*

The equality operators may be used to combine two operands of integral type or two operands of boolean type.

9.16.1 *Integer Bitwise Operators* `&`, `^`, *and* |

If one operand of a `&`, `^`, or | operator is a value of primitive integral type, the other operand must also be a value of (possibly some other) primitive integral type. Binary arithmetic promotion is performed on the operands (§3.6); the compile-time type of the entire expression the promoted type of the operands.

If the operator is `&`, the result is the bitwise AND function of the operands.

If the operator is `^`, the result is the bitwise exclusive OR function of the operands.

If the operator is |, the result is the bitwise inclusive OR function of the operands.

9.16.2 *Boolean Logical Operators* `&`, `^`, *and* |

If one operand of a `&`, `^`, or | operator is of type `boolean`, the other operand must also be of type `boolean`. The compile-time type of the entire expression is then `boolean`.

For `&`, the result is `true` if both operand values are `true`; otherwise the result is `false`.

For `^`, the result is `true` if the operand values are different; otherwise the result is `false`.

For |, the result is `false` if both operand values are `false`; otherwise the result is `true`.

9.17 Conditional-And Operator `&&`

The `&&` operator is syntactically left-associative (it groups left-to-right). It is associative with respect to both side effects and result value. It is commutative with respect to result value but not with respect to whether side effects in its operand expressions will occur.

ConditionalAndExpression:
InclusiveOrExpression
ConditionalAndExpression `&&` *InclusiveOrExpression*

Each operand of `&&` must be of type `boolean`. The compile-time type of the result is `boolean`. The left-hand operand expression is executed first; if its value is `false`, the value of the conditional-and expression is `false` and the right-hand operand expression is not executed. If the value of the left-hand operand is `true`, then the right-hand expression is executed 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 executed conditionally rather than always.

9.18 Conditional-Or Operator `||`

The `||` operator is syntactically left-associative (it groups left-to-right). It is associative with respect to both side effects and result value. It is commutative with respect to result value but not with respect to whether side effects in its operand expressions will occur.

ConditionalOrExpression:
ConditionalAndExpression
ConditionalOrExpression `||` *ConditionalAndExpression*

Each operand of `||`must be of type `boolean`. The compile-time type of the result is `boolean`. The left-hand operand expression is executed first; if its value is `true`, the value of the conditional-or expression is `true` and the right-hand operand expression is not executed. If the value of the left-hand operand is `false`, then the right-hand expression is executed 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 executed conditionally rather than always.

9.19 Conditional Operator `? :`

The conditional operator is syntactically right-associative (they group 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 expresssions. The first expression must be of type `boolean`. The compile-time types of the second and third expressions must both be primitive arithmetic types, or must both be boolean, or must both be reference types. (It is not permitted for either the second or the third operand expression to have type `void`.)

- If the second and third operands have arithmetic type, then there are several cases:
  - If the operands have the same type, then that is the compile-time type of the conditional expression.

  - If one of the operands is of type `byte` and the other is of type `short`, then the compile-time type of the conditional expression is `short`. (Here the Java language differs from C and C++.)
  - If one of the operands is of type `T` where `T` is `byte`, `short`, or `char`, and the other operand is an integer constant expression whose value is representable in type `T`, then the compile-time type of the conditional expression is `T`. (Here the Java language differs from C and C++.)
  - Otherwise, binary arithmetic promotion (§3.6) is applied to their types to determine a common promoted type, which is the compile-time type of the conditional expression.
- If the second and third operands are of type `boolean`, then the compile-time type of the conditional expression is `boolean`.
- If the second and third operands are both `null`, then the result of the conditional expression is `null`.
- If one of the second and third operands is `null` and the type of the other is a reference type, then the compile-time type of the conditional expression is that reference type.
- If the compile-time types of the second and third operands are (possibly different) reference types, then it must be possible to convert one of the types to the other type (call this type `T`) by assignment conversion (§3.3); the compile-time type of the conditional expression is then `T`. It is a compile-time error if neither type can be assigned to the other type.

At run time, for each execution of the conditional expression, the first operand expression is executed first; its value is then used to choose one of the second and third operand expressions for execution.

- If the value of the first operand is `true`, the second operand expression is chosen.
- If the value of the first operand is `false`, the third operand expression is chosen.

The chosen operand expression is then executed and the resulting value is converted to the compile-time type of the conditional expression as deteremined by the rules stated above. The operand expression not chosen is not executed for that particular execution of the conditional expression.

9.20 Assignment Operators

There are many 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:
    UnaryExpression AssignmentOperator AssignmentExpression

AssignmentOperator: one of
    `=  *=  /=  %=  +=  -=  <<=  >>=  >>>=  &=  ^=  |=`

The first operand of an assignment operator must be a variable, which may be a named variable (such as a local variable or a field variable) or a computed variable (such as an array component). The compile-time type of the assignment expression is the type of the variable. The result of the assignment expression is the value of the variable after the assignment has occurred (but this result is not itself a variable—in this respect the Java language is like C but unlike C++).

Note that it is not possible to assign to a variable that has been declared `final`, because mentioning the name of the variable, on the left-hand side of an assignment operator or anywhere else, produces its value rather than the variable itself (§9.4, §9.6).

9.20.1 *Simple Assignment Operator =*

The simple assignment operator converts the value of its right-hand operand to the type of the left-hand variable and stores this converted value into the variable. It is a compile-time error if the right-hand operand cannot be converted to the type of the variable by assignment conversion (§3.3).

9.20.2 *Compound Assignment Operators*

All compound assignment operators require both operands to be of primitive type.

An expression of the form of the form `E1` *op*= `E2` is equivalent to `E1` = `E1` *op* `(E2)` except that `E1` is evaluated only once.

9.21 Expression

An expression is an assignment-expression:

Expression:
    AssignmentExpression

(Unlike C and C++, the Java language has no comma operator.)

9.22 Constant Expression

A constant expression is an expression of primitive type that is formed from literals of primitive type; final variables whose initialization values are constant expressions; casts to primitive types; the unary operators +, –, ~, and !; the binary operators *, /, %, +, -, <, <=, >, >=, ==, !=, &, ^, |, &&, and ||; and the ternary conditional operator `? :`.

+ - ! ~

Constant expressions are used in `interface` declarations and `case` labels in `switch` statements.

9.23 Unassigned Variables

It is a compile-time error if a variable might be referenced before it has definitely been assigned or initialized. The precise meaning of this remark will be explained in a future version of this document.

10 Collected Java Grammar

So far, only the lexical grammar is collected. The rest is awaiting a mechanical cross-check with a yacc grammar before the tedium of cut/paste...

10.1 Lexical Structure

Unicode escape sequences in the original *RawInputCharacter*s are translated as described in §1.7.4:

EscapedInputCharacter:
    UnicodeEscape
    RawInputCharacter
UnicodeEscape:
    `\ u` *HexDigit*
    `\ u` *HexDigit HexDigit*
    `\ u` *HexDigit HexDigit HexDigit*
    `\ u` *HexDigit HexDigit HexDigit HexDigit*
RawInputCharacter:
    Any Unicode Character
HexDigit: one of
    `0 1 2 3 4 5 6 7 8 9 0 a b c d e f A B C D E F`

The *EscapedInputCharacter*s are separated into normal input characters and line terminators, as described in §1.2:

LineTerminator:
    `CR`
    `LF`
    `CR LF`
InputCharacter:
    EscapedInputCharacter, but not `CR` *and not* `LF`

The sequence of LineTerminators and InputCharacters is reduced to a sequence of tokens, as described in §1.3:

Input:
    $InputElements_{opt}$
InputElements:
    InputElement
    InputElements InputElement
InputElement:
    Comment
    WhiteSpace
    Token
WhiteSpace: one of
    `SP HT FF` *LineTerminator*
Token:
    Keyword
    Identifer
    Literal
    Separator
    Operator

The Comments are formed as described in §1.4:

Comment:
    `/ *` *TraditionalCommentTail*
    `/ * *` *DocCommentTail*
    `/ /` *CharactersInLine*$_{opt}$ *LineTerminator*
TraditionalCommentTail:
    `* /`
    InputCharacter TraditionalCommentTail
    LineTerminator TraditionalCommentTail
DocCommentTail:
    `/`
    InputCharacter TraditionalCommentTail
    LineTerminator TraditionalCommentTail
    TraditionalCommentTail
CharactersInLine:
    InputCharacter
    CharactersInLine InputCharacter

The keyword tokens are formed as described in §1.5. Here we abuse the grammar notation and show multiple input characters without the normal spaces between them, using spaces to separate multiple results on a single line, avoiding many tedious lines like:

Keyword:
    `a b s t r a c t`
    `b o o l e a n`

instead more compactly writing:

Keyword: one of

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

The lexical structure of identifier tokens are formed as described in §1.6. The *UnicodeLetter* s and *UnicodeDigit*s are defined there. An identifier must not have the same spelling (code point sequence) as a keyword:

Identifier:
    UnicodeLetter
    Identifier UnicodeLetter
    Identifier UnicodeDigit

The literal tokens are described in §1.7:

Literal:
    IntegerLiteral
    FloatingPointLiteral
    BooleanLiteral
    CharacterLiteral
    StringLiteral

The integer literals are formed as described in §1.7.1:

IntegerLiteral:
    DecimalLiteral IntegerTypeSuffix$_{opt}$
    HexLiteral IntegerTypeSuffix$_{opt}$
    OctalLiteral IntegerTypeSuffix$_{opt}$

DecimalLiteral:
    NonZeroDigit Digits$_{opt}$
Digits:
    Digit
    Digits Digit
Digit:
    `0`
    NonZeroDigit
NonZeroDigit: one of
    `1 2 3 4 5 6 7 8 9`

HexLiteral:
    `0x` *HexDigit*
    `0X` *HexDigit*
    HexLiteral HexDigit
HexDigit: one of
    `0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

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

The floating-point literal tokens are formed as described in §1.7.2:

FloatingPointLiteral :
    Digits `.` *Digits*$_{opt}$ *ExponentPart*$_{opt}$ *FloatTypeSuffix*$_{opt}$
    `.` *Digits ExponentPart*$_{opt}$ *FloatTypeSuffix*$_{opt}$
    Digits ExponentPart FloatTypeSuffix$_{opt}$
ExponentPart:
    ExponentIndicator SignedInteger$_{opt}$
ExponentIndicator: one of
    `e E`
SignedInteger:
    Sign$_{opt}$ *Digits*
Sign: one of
    `+ -`
FloatTypeSuffix: one of
    `f F d D`

The `boolean` literal tokens are formed as described in §1.7.3:

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

The character literal tokens are formed as described in §1.6:

CharacterLiteral:
`'` *SingleCharacter* `'`
`'` *Escape* `'`
SingleCharacter:
InputCharacter, excluding: `'` `\`

The escape sequences for character and string literals are formed as described in §1.7.4:

Escape:
```
\ n                 // u+000c: linefeed LF
\ t                 // u+000b: horizontal tab HT
\ b                 // u+000a: backspace BS
\ r                 // u+000e: carriage return CR
\ f                 // u+000d: form feed FF
\ \                 // u+005c: backslash
\ '                 // u+0060: single quote
\ "                 // u+0022: double quote
OctalEscape         // u+0000 to u+00ff: 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`

The string literal tokens are formed as described in §1.7.5:

StringLiteral:
`"` *StringCharacters* `"`
StringCharacters:
StringCharacter
StringCharacters StringCharacter
StringCharacter:
InputCharacter, excluding: `"` `\`
Escape

The separator tokens are formed as described in §1.8:

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

The operator tokens are formed as described in §1.9; Here we once again abuse the grammar notation and show multiple input characters without the normal spaces between them, using spaces to separate multiple results on a single line, avoiding many tedious lines like:

Operator:
...
`= =`
`< =`
`> =`
...

and instead write:

Operator: one of

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