CHAPTER 3

Lexical Structure

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

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

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

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

3.1 Unicode

Programs are written using the Unicode character set. Information about this character set and its associated character encodings may be found at:

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

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

Versions of the Java programming language prior to 1.1 used Unicode version 1.1.5. Upgrades to newer versions of the Unicode Standard occurred in JDK 1.1 (to Unicode 2.0), JDK 1.1.7 (to Unicode 2.1), J2SE 1.4 (to Unicode 3.0), and J2SE 5.0 (to Unicode 4.0).

The Unicode standard was originally designed as a fixed-width 16-bit character encoding. It has since been changed to allow for characters whose representation requires more than 16 bits. The range of legal code points is now U+0000 to U+10FFFF, using the hexadecimal *U+n notation*. Characters whose code points are greater than U+FFFF are called supplementary characters. To represent the complete range of characters using only 16-bit units, the Unicode standard defines an encoding called UTF-16. In this encoding, supplementary characters are represented as pairs of 16-bit code units, the first from the high-surrogates range, (U+D800 to U+DBFF), the second from the low-surrogates range (U+DC00 to U+DFFF). For characters in the range U+0000 to U+FFFF, the values of code points and UTF-16 code units are the same.

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

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

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

3.2 Lexical Translations

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

1. A translation of Unicode escapes (§3.3) in the raw stream of Unicode characters to the corresponding Unicode character. A Unicode escape of the form `\u`*`xxxx`*, where *`xxxx`* is a hexadecimal value, represents the UTF-16 code unit whose encoding is *`xxxx`*. This translation step allows any program to be expressed using only ASCII characters.

2. A translation of the Unicode stream resulting from step 1 into a stream of input characters and line terminators (§3.4).

3. A translation of the stream of input characters and line terminators resulting from step 2 into a sequence of input elements (§3.5) which, after white space (§3.6) and comments (§3.7) are discarded, comprise the tokens (§3.5) that are the terminal symbols of the syntactic grammar (§2.3).

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

3.3 Unicode Escapes

Implementations first recognize *Unicode escapes* in their input, translating the ASCII characters `\u` followed by four hexadecimal digits to the UTF-16 code unit (§3.1) with the indicated hexadecimal value, and passing all other characters unchanged. Representing supplementary characters requires two consecutive Unicode escapes. This translation step results in a sequence of Unicode input characters:

UnicodeInputCharacter:
  UnicodeEscape
  RawInputCharacter

UnicodeEscape:
  `\` *UnicodeMarker HexDigit HexDigit HexDigit HexDigit*

UnicodeMarker:
  `u`
  UnicodeMarker `u`

RawInputCharacter:
  any Unicode character

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

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

In addition to the processing implied by the grammar, for each raw input character that is a backslash `\`, input processing must consider how many other `\` characters contiguously precede it, separating it from a non-`\` character or the start of the input stream. If this number is even, then the `\` is eligible to begin a Unicode escape; if the number is odd, then the `\` is not eligible to begin a Unicode escape. For example, the raw input `"\\u2297=\u2297"` results in the eleven characters `" \ \ u 2 2 9 7 = ⊗ "` (`\u2297` is the Unicode encoding of the character "⊗").

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

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

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

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

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

3.4 Line Terminators

Implementations next divide the sequence of Unicode input characters into lines by recognizing *line terminators*. This definition of lines determines the line numbers produced by a Java compiler or other system component. It also specifies the termination of the `//` form of a comment (§3.7).

LineTerminator:
    the ASCII LF character, also known as “newline”
    the ASCII CR character, also known as “return”
    the ASCII CR character followed by the ASCII LF character

InputCharacter:
    UnicodeInputCharacter but not CR or LF

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

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

3.5 Input Elements and Tokens

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

This process is specified by the following productions:

Input:
    InputElements$_{opt}$ *Sub*$_{opt}$

InputElements:
    InputElement
    InputElements InputElement

InputElement:
    WhiteSpace
    Comment
    Token

Token:
    Identifier
    Keyword
    Literal
    Separator
    Operator

Sub:
    the ASCII SUB character, also known as “control-Z”

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

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

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

For example, in this simple piece of code:

```
class Empty {
}
```

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

3.6 White Space

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

WhiteSpace:
    the ASCII SP character, also known as “space”
    the ASCII HT character, also known as “horizontal tab”
    the ASCII FF character, also known as “form feed”
    LineTerminator

3.7 Comments

There are two kinds of *comments*:

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

These comments are formally specified by the following productions:

Comment:
    TraditionalComment
    EndOfLineComment

TraditionalComment:
    `/ *` *CommentTail*

EndOfLineComment:
  `/ /` *CharactersInLine*$_{opt}$

CommentTail:
  `*` *CommentTailStar*
  NotStar CommentTail

CommentTailStar:
  `/`
  `*` *CommentTailStar*
  NotStarNotSlash CommentTail

NotStar:
  InputCharacter but not `*`
  LineTerminator

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

CharactersInLine:
  InputCharacter
  CharactersInLine InputCharacter

These productions imply all of the following properties:

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

As a result, the text:

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

is a single complete comment.

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

3.8 Identifiers

An *identifier* is an unlimited-length sequence of *Java letters* and *Java digits*, the first of which must be a Java letter. An identifier cannot have the same spelling (Unicode character sequence) as a keyword (§3.9), boolean literal (§3.10.3), or the null literal (§3.10.7).

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

IdentifierChars:
  JavaLetter
  IdentifierChars JavaLetterOrDigit

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

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

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

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

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

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

Two identifiers are the same only if they are identical, that is, have the same Unicode character for each letter or digit.

Identifiers that have the same external appearance may yet be different. For example, the identifiers consisting of the single letters LATIN CAPITAL LETTER A (`A`, `\u0041`), LATIN SMALL LETTER A (`a`, `\u0061`), GREEK CAPITAL LETTER ALPHA (`A`, `\u0391`), CYRILLIC SMALL LETTER A (`a`, `\u0430`) and MATHEMATICAL BOLD ITALIC SMALL A (`a`, `\ud835\udc82`) are all different.

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

```
String    i3    αρετη    MAX_VALUE    isLetterOrDigit
```

3.9 Keywords

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

Keyword: one of

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

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

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

3.10 Literals

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

Literal:
- *IntegerLiteral*
- *FloatingPointLiteral*
- *BooleanLiteral*
- *CharacterLiteral*
- *StringLiteral*
- *NullLiteral*

3.10.1 Integer Literals

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

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

IntegerLiteral:
  DecimalIntegerLiteral
  HexIntegerLiteral
  OctalIntegerLiteral

DecimalIntegerLiteral:
  DecimalNumeral IntegerTypeSuffix$_{opt}$

HexIntegerLiteral:
  HexNumeral IntegerTypeSuffix$_{opt}$

OctalIntegerLiteral:
  OctalNumeral IntegerTypeSuffix$_{opt}$

IntegerTypeSuffix: one of
  `l L`

An integer literal is of type `long` if it is suffixed with an ASCII letter `L` or `l` (ell); otherwise it is of type `int` (§4.2.1). The suffix `L` is preferred, because the letter `l` (ell) is often hard to distinguish from the digit `1` (one).

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

DecimalNumeral:
  `0`
  NonZeroDigit Digits$_{opt}$

Digits:
  Digit
  Digits Digit

Digit:
  `0`
  NonZeroDigit

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

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

HexNumeral:
`0 x` *HexDigits*
`0 X` *HexDigits*

HexDigits:
HexDigit
HexDigit HexDigits

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

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

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

OctalNumeral:
`0` *OctalDigits*

OctalDigits:
OctalDigit
OctalDigit OctalDigits

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

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

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

The largest positive hexadecimal and octal literals of type `int` are `0x7fffffff` and `017777777777`, respectively, which equal `2147483647` ($2^{31}-1$). The most negative hexadecimal and octal literals of type `int` are `0x80000000` and `020000000000`, respectively, each of which represents the decimal value `–2147483648` (-2^{31}). The hexadecimal and octal literals `0xffffffff` and `037777777777`, respectively, represent the decimal value `-1`.

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

Examples of `int` literals:

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

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

The largest positive hexadecimal and octal literals of type `long` are `0x7fffffffffffffffL` and `0777777777777777777777L`, respectively, which equal `9223372036854775807L` ($2^{63} - 1$). The literals `0x8000000000000000L` and `01000000000000000000000L` are the most negative `long` hexadecimal and octal literals, respectively. Each has the decimal value `–9223372036854775808L` (-2^{63}). The hexadecimal and octal literals `0xffffffffffffffffL` and `01777777777777777777777L`, respectively, represent the decimal value `-1L`.

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

Examples of `long` literals:

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

3.10.2 Floating-Point Literals

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

A *floating-point literal* has the following parts: a whole-number part, a decimal or hexadecimal point (represented by an ASCII period character), a fractional part, an exponent, and a type suffix. A floating point number may be written either as a decimal value or as a hexadecimal value. For decimal literals, the exponent, if present, is indicated by the ASCII letter `e` or `E` followed by an optionally signed integer. For hexadecimal literals, the exponent is always required and is indicated by the ASCII letter `p` or `P` followed by an optionally signed integer.

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

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

FloatingPointLiteral:
  DecimalFloatingPointLiteral
  HexadecimalFloatingPointLiteral

DecimalFloatingPointLiteral:
  Digits `.` *Digits*$_{opt}$ *ExponentPart*$_{opt}$ *FloatTypeSuffix*$_{opt}$
  `.` *Digits* *ExponentPart*$_{opt}$ *FloatTypeSuffix*$_{opt}$
  Digits *ExponentPart* *FloatTypeSuffix*$_{opt}$
  Digits *ExponentPart*$_{opt}$ *FloatTypeSuffix*

ExponentPart:
  ExponentIndicator SignedInteger

ExponentIndicator: one of
  `e E`

SignedInteger:
  Sign$_{opt}$ *Digits*

Sign: one of
  `+ -`

FloatTypeSuffix: one of
  `f F d D`

HexadecimalFloatingPointLiteral:
  HexSignificand BinaryExponent FloatTypeSuffix$_{opt}$

HexSignificand:
  HexNumeral
  HexNumeral .
  0x HexDigits$_{opt}$ *. HexDigits*
  0X HexDigits$_{opt}$ *. HexDigits*

BinaryExponent:
  BinaryExponentIndicator SignedInteger

BinaryExponentIndicator:one of
  `p P`

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

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

The largest positive finite `float` literal is `3.4028235e38f`. The smallest positive finite nonzero literal of type `float` is `1.40e-45f`. The largest positive finite `double` literal is `1.7976931348623157e308`. The smallest positive finite nonzero literal of type `double` is `4.9e-324`.

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

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

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

Examples of `float` literals:

```
1e1f2.f.3f0f3.14f6.022137e+23f
```

Examples of `double` literals:

```
1e12..30.03.141e-9d1e137
```

Besides expressing floating-point values in decimal and hexadecimal, the method `intBitsToFloat` of class `Float` and method `longBitsToDouble` of class `Double` provide a way to express floating-point values in terms of hexadecimal or octal integer literals.For example, the value of:

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

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

3.10.3 Boolean Literals

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

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

BooleanLiteral: one of
```
true false
```

3.10.4 Character Literals

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

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

A character literal is always of type `char`.

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

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

The escape sequences are described in §3.10.6.

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

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

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

The following are examples of `char` literals:

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

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

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

3.10.5 String Literals

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

A string literal is always of type `String` (§4.3.3). A string literal always refers to the same instance (§4.3.1) of class `String`.

StringLiteral:
  `"` *StringCharacters*$_{opt}$ `"`

StringCharacters:
  StringCharacter
  StringCharacters StringCharacter

StringCharacter:
  InputCharacter but not `"` or `\`
  EscapeSequence

The escape sequences are described in §3.10.6.

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

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

The following are examples of string literals:

```
""                    // the empty string
"\""                  // a string containing " alone
"This is a string"    // a string containing 16 characters
"This is a " +        // actually a string-valued constant expression,
    "two-line string" //    formed from two string literals
```

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

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

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

```
package testPackage;

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

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

and the compilation unit:

```
package other;

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

produces the output:

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

This example illustrates six points:

- Literal strings within the same class (§8) in the same package (§7) represent references to the same `String` object (§4.3.1).
- Literal strings within different classes in the same package represent references to the same `String` object.
- Literal strings within different classes in different packages likewise represent references to the same `String` object.
- Strings computed by constant expressions (§15.28) are computed at compile time and then treated as if they were literals.
- Strings computed by concatenation at run time are newly created and therefore distinct.

The result of explicitly interning a computed string is the same string as any pre-existing literal string with the same contents.

3.10.6 Escape Sequences for Character and String Literals

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

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

OctalEscape:
    \ OctalDigit
    \ OctalDigit OctalDigit
    \ ZeroToThree OctalDigit OctalDigit

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

ZeroToThree: one of
    0 1 2 3
```

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

3.10.7 The Null Literal

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

```
NullLiteral:
    null
```

3.11 Separators

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

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

3.12 Operators

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

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

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

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

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

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

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

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