Table of Contents

3.1. Unicode
3.2. Lexical Translations
3.3. Unicode Escapes
3.4. Line Terminators
3.5. Input Elements and Tokens
3.6. White Space
3.7. Comments
3.8. Identifiers
3.9. Keywords
3.10. Literals
3.10.1. Integer Literals
3.10.2. Floating-Point Literals
3.10.3. Boolean Literals
3.10.4. Character Literals
3.10.5. String Literals
3.10.6. Text Blocks
3.10.7. Escape Sequences
3.10.8. The Null Literal
3.11. Separators
3.12. Operators

Chapter 3. Lexical Structure

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

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

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

3.1. Unicode

Programs are written using the Unicode character set (§1.7). Information about this character set and its associated character encodings may be found at `https://www.unicode.org/`.

The Java SE Platform tracks the Unicode Standard 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 JDK 1.1 used Unicode 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), Java SE 1.4 (to Unicode 3.0), Java SE 5.0 (to Unicode 4.0), Java SE 7 (to Unicode 6.0), Java SE 8 (to Unicode 6.2), Java SE 9 (to Unicode 8.0), Java SE 11 (to Unicode 10.0), Java SE 12 (to Unicode 11.0), Java SE 13 (to Unicode 12.1), and Java SE 15 (to Unicode 13.0).

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), and 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.

Some APIs of the Java SE Platform, primarily in the `Character` class, use 32-bit integers to represent code points as individual entities. The Java SE Platform provides methods to convert between 16-bit and 32-bit representations.

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

Except for comments (§3.7), identifiers (§3.8), and the contents of character literals, string literals, and text blocks (§3.10.4, §3.10.5, §3.10.6), 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 UTF-16 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 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. There are two exceptions to account for situations that need more granular translation: in step 1, for the processing of contiguous \ characters (§3.3), and in step 3, for the processing of contextual keywords and adjacent > characters (§3.5).

The input characters `a--b` are tokenized as `a`, `--`, and `b`, which is not part of any grammatically correct program, even though the tokenization `a`, `-`, `-`, `b` could be part of a grammatically correct program. The tokenization `a`, `-`, `-`, `b` can be realized with the input characters `a- -b` (with an ASCII SP character between the two `-` characters).

It might be supposed that the raw input `\\u1234` is translated to a `\` character and (following the "longest possible" rule) a Unicode escape of the form `\u1234`. In fact, the leading `\` character causes this raw input to be translated to seven distinct characters: `\` `\` `u` `1` `2` `3` `4`.

3.3. Unicode Escapes

A compiler for the Java programming language ("Java compiler") first recognizes Unicode escapes in its raw input, translating the ASCII characters \u followed by four hexadecimal digits to a *raw input character* which denotes the UTF-16 code unit (§3.1) for the indicated hexadecimal value. One Unicode escape can represent characters in the range U+0000 to U+FFFF; representing supplementary characters in the range U+010000 to U+10FFFF requires two consecutive Unicode escapes. All other characters in the compiler's raw input are recognized as raw input characters and passed unchanged.

This translation step results in a sequence of Unicode input characters, all of which are raw input characters (any Unicode escapes having been reduced to raw input characters).

```
UnicodeInputCharacter:
  UnicodeEscape
  RawInputCharacter

UnicodeEscape:
  \ UnicodeMarker HexDigit HexDigit HexDigit HexDigit

UnicodeMarker:
  u {u}

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

RawInputCharacter:
  any Unicode character
```

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

The *UnicodeInputCharacter* production is ambiguous because an ASCII \ character in the compiler's raw input could be reduced to either a *RawInputCharacter* or the \ of a *UnicodeEscape* (to be followed by an ASCII u). To avoid ambiguity, for each ASCII \ character in the compiler's raw input, input processing must consider the most recent raw input characters that resulted from this translation step:

- If the most recent raw input character in the result was itself translated from a Unicode escape in the compiler's raw input, then the ASCII \ character is eligible to begin a Unicode escape.

  For example, if the most recent raw input character in the result was a backslash that arose from a Unicode escape `\u005c` in the raw input, then an ASCII \ character appearing next in the raw input is eligible to begin another Unicode escape.

- Otherwise, consider how many backslashes appeared contiguously as raw input characters in the result, back to a non-backslash character or the start of the result. (It is immaterial whether any such backslash arose from an ASCII \ character in the compiler's raw input or from a Unicode escape `\u005c` in the compiler's raw input.) If this number is even, then the ASCII \ character is eligible to begin a Unicode escape; if the number is odd, then the ASCII \ character is not eligible to begin a Unicode escape.

  For example, the raw input `"\\u2122=\u2122"` results in the eleven characters " \ \ u 2 1 2 2 = ™ " because while the second ASCII \ character in the raw input is not eligible to begin a Unicode escape, the third ASCII \ character is eligible, and `\u2122` 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 a backslash. It does not result in the character Z, which is Unicode value `005a`, because the backslash that resulted from processing the Unicode escape `\u005c` is not interpreted as the start of a further Unicode escape.*
>
> *Note that `\u005cu005a` cannot be written in a string literal to denote the six characters `\ u 0 0 5 a`. This is because the first two characters resulting from translation, \ and u, are interpreted in a string literal as an illegal escape sequence (§3.10.7).*
>
> *Fortunately, the rule about contiguous backslash characters helps programmers to craft raw inputs that denote Unicode escapes in a string literal. Denoting the six characters `\ u 0 0 5 a` in a string literal simply requires another \ to be placed adjacent to the existing \, such as `"\\u005a is Z"`. This works because the second \ in the raw input `\\u005a` is not eligible to begin a Unicode escape, so the first \ and the second \ are preserved as raw input characters, as are the next five characters `u 0 0 5 a`. The two \ characters are subsequently interpreted in a string literal as the escape sequence for a backslash, resulting in a string with the desired six characters `\ u 0 0 5 a`. Without the rule, the raw input `\\u005a` would be processed as a raw input character \ followed by a Unicode escape `\u005a` which becomes a raw input character Z; this would be unhelpful because `\Z` is an illegal escape sequence in a string literal. (Note that the rule translates `\u005c\u005c` to `\\` because the translation of the first Unicode escape to a raw input character \ does not prevent the translation of the second Unicode escape to another raw input character \.)*
>
> *The rule also allows programmers to craft raw inputs that denote escape sequences in a string literal. For example, the raw input `\\\u006e` results in the three characters `\ \ n` because the first \ and the second \ are preserved as raw input characters, while the third \ is eligible to begin a Unicode escape and thus `\u006e` is translated to a raw input character n. The three characters `\ \ n` are subsequently interpreted in a string literal as `\ n` which denotes the escape sequence for a linefeed. (Note that `\\\u006e` may be written as `\u005c\u005c\u006e` because each Unicode escape `\u005c` is translated to a raw input character \ and so the remaining raw input `\u006e` is preceded by an even number of backslashes and processed as the Unicode escape for n.)*

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

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

A Java compiler should use the `\uxxxx` notation as an output format to display Unicode characters when a suitable font is not available.

3.4. Line Terminators

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

```
LineTerminator:
  the ASCII LF character, also known as "newline"
  the ASCII CR character, also known as "return"
  the ASCII CR character followed by the ASCII LF character

InputCharacter:
  UnicodeInputCharacter but not CR or LF
```

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

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

The lines defined by line terminators may determine the line numbers produced by a Java compiler.

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 Unicode escape processing (§3.3) and then input line recognition (§3.4) are reduced to a sequence of *input elements*.

```
Input:
  {InputElement} [Sub]

InputElement:
  WhiteSpace
  Comment
  Token

Token:
  Identifier
  Keyword
  Literal
  Separator
  Operator

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

Those input elements that are not white space or comments are *tokens*. The tokens are the terminal symbols of the syntactic grammar (§2.3).

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 input characters `-` and `=` can form the operator token `-=` (§3.12) only if there is no intervening white space or comment. As another example, the ten input characters `staticvoid` form a single identifier token while the eleven input characters `static void` (with an ASCII SP character between `c` and `v`) form a pair of keyword tokens, `static` and `void`, separated by white space.

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.

The *Input* production is ambiguous, meaning that for some sequences of input characters, there is more than one way to reduce the input characters to input elements (that is, to tokenize the input characters). Ambiguities are resolved as follows:

- A sequence of input characters that could be reduced to either an identifier token or a literal token is always reduced to a literal token.
- A sequence of input characters that could be reduced to either an identifier token or a reserved keyword token (§3.9) is always reduced to a reserved keyword token.
- A sequence of input characters that could be reduced to either a contextual keyword token or to other (non-keyword) tokens is reduced according to context, as specified in §3.9.
- If the input character `>` appears in a type context (§4.11), that is, as part of a *Type* or an *UnannType* in the syntactic grammar (§4.1, §8.3), it is always reduced to the numerical comparison operator `>`, even when it could be combined with an adjacent `>` character to form a different operator.

  Without this rule for `>` characters, two consecutive `>` brackets in a type such as `List<List<String>>` would be tokenized as the signed right shift operator `>>`, while three consecutive `>` brackets in a type such as `List<List<List<String>>>` would be tokenized as the unsigned right shift operator `>>>`. Worse, the tokenization of four or more consecutive `>` brackets in a type such as `List<List<List<List<String>>>>` would be ambiguous, as various combinations of `>`, `>>`, and `>>>` tokens could represent the `>>>>` characters.

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, 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 character, horizontal tab character, form feed character, and line terminator characters (§3.4).

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

3.7. Comments

There are two kinds of comments:

- `/* text */`

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

- `// text`

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

```
Comment:
  TraditionalComment
  EndOfLineComment

TraditionalComment:
  / * CommentTail

CommentTail:
  * CommentTailStar
  NotStar CommentTail

CommentTailStar:
  /
  * CommentTailStar
  NotStarNotSlash CommentTail

NotStar:
  InputCharacter but not *
  LineTerminator

NotStarNotSlash:
  InputCharacter but not * or /
  LineTerminator

EndOfLineComment:
  / / {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 following text is a single complete comment:

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

The lexical grammar implies that comments do not occur within character literals, string literals, or text blocks (§3.10.4, §3.10.5, §3.10.6).

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*.

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

IdentifierChars:
  JavaLetter {JavaLetterOrDigit}

JavaLetter:
  any Unicode character that is a "Java letter"

JavaLetterOrDigit:
  any Unicode character that is a "Java letter-or-digit"
```

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 dollar sign ($, or `\u0024`) and underscore (_, or `\u005f`). The dollar sign should be used only in mechanically generated source code or, rarely, to access pre-existing names on legacy systems. The underscore may be used in identifiers formed of two or more characters, but it cannot be used as a one-character identifier due to being a keyword.

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

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.

Two identifiers are the same only if, after ignoring characters that are ignorable, the identifiers have the same Unicode character for each letter or digit. An ignorable character is a character for which the method `Character.isIdentifierIgnorable(int)` returns true. 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 (Α, `\u0391`), CYRILLIC SMALL LETTER A (а, `\u0430`) and MATHEMATICAL BOLD

ITALIC SMALL A (a, \ud835\udc82) are all different.

Unicode composite characters are different from their canonical equivalent decomposed characters. For example, a LATIN CAPITAL LETTER A ACUTE (Á, \u00c1) is different from a LATIN CAPITAL LETTER A (A, \u0041) immediately followed by a NON-SPACING ACUTE (́, \u0301) in identifiers. See The Unicode Standard, Section 3.11 "Normalization Forms".

Examples of identifiers are:

- `String`
- `i3`
- `αρετη`
- `MAX_VALUE`
- `isLetterOrDigit`

An identifier never has the same spelling (Unicode character sequence) as a reserved keyword (§3.9), a boolean literal (§3.10.3) or the null literal (§3.10.8), due to the rules of tokenization (§3.5). However, an identifier may have the same spelling as a contextual keyword, because the tokenization of a sequence of input characters as an identifier or a contextual keyword depends on where the sequence appears in the program.

To facilitate the recognition of contextual keywords, the syntactic grammar (§2.3) sometimes disallows certain identifiers by defining a production to accept only a subset of identifiers. The subsets are as follows:

```
TypeIdentifier:
  Identifier but not permits, record, sealed, var, or yield

UnqualifiedMethodIdentifier:
  Identifier but not yield
```

TypeIdentifier is used in the declaration of classes, interfaces, and type parameters (§8.1, §9.1, §4.4), and when referring to types (§6.5). For example, the name of a class must be a TypeIdentifier, so it is illegal to declare a class named `permits`, `record`, `sealed`, `var`, or `yield`.

UnqualifiedMethodIdentifier is used when a method invocation expression refers to a method by its simple name (§6.5.7.1). Since the term `yield` is excluded from UnqualifiedMethodIdentifier, any invocation of a method named `yield` must be qualified, thus distinguishing the invocation from a `yield` statement (§14.21).

3.9. Keywords

51 character sequences, formed from ASCII characters, are reserved for use as keywords and cannot be used as identifiers (§3.8). Another 16 character sequences, also formed from ASCII characters, may be interpreted as keywords or as other tokens, depending on the context in which they appear.

```
Keyword:
  ReservedKeyword
  ContextualKeyword

ReservedKeyword:
  (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
  _ (underscore)

ContextualKeyword:
  (one of)

  exports      opens      requires     uses
  module       permits    sealed       var
  non-sealed   provides   to           with
  open         record     transitive   yield
```

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.

The keyword `strictfp` is obsolete and should not be used in new code.

The keyword _ (underscore) is reserved for possible future use in parameter declarations.

`true` and `false` are not keywords, but rather boolean literals (§3.10.3).

`null` is not a keyword, but rather the null literal (§3.10.8).

During the reduction of input characters to input elements (§3.5), a sequence of input characters that notionally matches a contextual keyword is reduced to a contextual keyword if and only if both of the following conditions hold:

1. The sequence is recognized as a terminal specified in a suitable context of the syntactic grammar (§2.3), as follows:
   - For `module` and `open`, when recognized as a terminal in a *ModuleDeclaration* (§7.7).

- For `exports`, `opens`, `provides`, `requires`, `to`, `uses`, and `with`, when recognized as a terminal in a *ModuleDirective*.
- For `transitive`, when recognized as a terminal in a *RequiresModifier*.

  For example, recognizing the sequence `requires transitive ;` does not make use of RequiresModifier, so the term `transitive` is reduced here to an identifier and not a contextual keyword.
- For `var`, when recognized as a terminal in a *LocalVariableType* (§14.4) or a *LambdaParameterType* (§15.27.1).

  In other contexts, attempting to use `var` as an identifier will cause an error, because `var` is not a TypeIdentifier (§3.8).
- For `yield`, when recognized as a terminal in a *YieldStatement* (§14.21).

  In other contexts, attempting to use the `yield` as an identifier will cause an error, because `yield` is neither a TypeIdentifier nor a UnqualifiedMethodIdentifier.
- For `record`, when recognized as a terminal in a *RecordDeclaration* (§8.10).
- For `non-sealed`, `permits`, and `sealed`, when recognized as a terminal in a *NormalClassDeclaration* (§8.1) or a *NormalInterfaceDeclaration* (§9.1).

2. The sequence is not immediately preceded or immediately followed by an input character that matches *JavaLetterOrDigit*.

In general, accidentally omitting white space in source code will cause a sequence of input characters to be tokenized as an identifier, due to the "longest possible translation" rule (§3.2). For example, the sequence of twelve input characters `p u b l i c s t a t i c` is always tokenized as the identifier `publicstatic`, rather than as the reserved keywords `public` and `static`. If two tokens are intended, they must be separated by white space or a comment.

The rule above works in tandem with the "longest possible translation" rule to produce an intuitive result in contexts where contextual keywords may appear. For example, the sequence of eleven input characters `v a r f i l e n a m e` is usually tokenized as the identifier `varfilename`, but in a local variable declaration, the first three input characters are tentatively recognized as the contextual keyword `var` by the first condition of the rule above. However, it would be confusing to overlook the lack of white space in the sequence by recognizing the next eight input characters as the identifier `filename`. (This would mean that the sequence undergoes different tokenization in different contexts: an identifier in most contexts, but a contextual keyword and an identifier in local variable declarations.) Accordingly, the second condition prevents recognition of the contextual keyword `var` on the grounds that the immediately following input character `f` is a JavaLetterOrDigit. The sequence `v a r f i l e n a m e` is therefore tokenized as the identifier `varfilename` in a local variable declaration.

As another example of the careful recognition of contextual keywords, consider the sequence of 15 input characters `n o n - s e a l e d c l a s s`. This sequence is usually translated to three tokens - the identifier `non`, the operator `-`, and the identifier `sealedclass` - but in a normal class declaration, where the first condition holds, the first ten input characters are tentatively recognized as the contextual keyword `non-sealed`. To avoid translating the sequence to two keyword tokens (`non-sealed` and `class`) rather than three non-keyword tokens, and to avoid rewarding the programmer for omitting white space before `class`, the second condition prevents recognition of the contextual keyword. The sequence `n o n - s e a l e d c l a s s` is therefore tokenized as three tokens in a class declaration.

In the rule above, the first condition depends on details of the syntactic grammar, but a compiler for the Java programming language can implement the rule without fully parsing the input program. For example, a heuristic could be used to track the contextual state of the tokenizer, as long as the heuristic guarantees that valid uses of contextual keywords are tokenized as keywords, and valid uses of identifiers are tokenized as identifiers. Alternatively, a compiler could always tokenize a contextual keyword as an identifier, leaving it to a later phase to recognize special uses of these identifiers.

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
  TextBlock
  NullLiteral
```

3.10.1. Integer Literals

An *integer literal* may be expressed in decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2).

```
IntegerLiteral:
  DecimalIntegerLiteral
  HexIntegerLiteral
  OctalIntegerLiteral
  BinaryIntegerLiteral

DecimalIntegerLiteral:
  DecimalNumeral [IntegerTypeSuffix]

HexIntegerLiteral:
  HexNumeral [IntegerTypeSuffix]

OctalIntegerLiteral:
  OctalNumeral [IntegerTypeSuffix]

BinaryIntegerLiteral:
  BinaryNumeral [IntegerTypeSuffix]

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).

Underscores are allowed as separators between digits that denote the integer.

In a hexadecimal or binary literal, the integer is only denoted by the digits after the `0x` or `0b` characters and before any type suffix. Therefore, underscores may not appear immediately after `0x` or `0b`, or after the last digit in the numeral.

In a decimal or octal literal, the integer is denoted by *all* the digits in the literal before any type suffix. Therefore, underscores may not appear before the first digit or after the last digit in the numeral. Underscores may appear after the initial `0` in an octal numeral (since `0` is a digit that denotes part of the integer) and after the initial non-zero digit in a non-zero decimal literal.

A decimal numeral is either the single ASCII digit `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` interspersed with underscores, representing a positive integer.

```
DecimalNumeral:
  0
  NonZeroDigit [Digits]
  NonZeroDigit Underscores Digits

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

Digits:
  Digit
  Digit [DigitsAndUnderscores] Digit

Digit:
  0
  NonZeroDigit

DigitsAndUnderscores:
  DigitOrUnderscore {DigitOrUnderscore}

DigitOrUnderscore:
  Digit
  _

Underscores:
  _ {_}
```

A hexadecimal numeral consists of the leading ASCII characters `0x` or `0X` followed by one or more ASCII hexadecimal digits interspersed with underscores, 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 [HexDigitsAndUnderscores] 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

HexDigitsAndUnderscores:
  HexDigitOrUnderscore {HexDigitOrUnderscore}

HexDigitOrUnderscore:
  HexDigit
  _
```

The HexDigit production above comes from §3.3.

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

```
OctalNumeral:
  0 OctalDigits
  0 Underscores OctalDigits

OctalDigits:
  OctalDigit
  OctalDigit [OctalDigitsAndUnderscores] OctalDigit

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

OctalDigitsAndUnderscores:
  OctalDigitOrUnderscore {OctalDigitOrUnderscore}

OctalDigitOrUnderscore:
  OctalDigit
  _
```

Note that octal numerals always consist of two or more digits, as `0` alone 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.

A binary numeral consists of the leading ASCII characters `0b` or `0B` followed by one or more of the ASCII digits `0` or `1` interspersed with underscores, and can represent a positive, zero, or negative integer.

```
BinaryNumeral:
  0 b BinaryDigits
  0 B BinaryDigits

BinaryDigits:
  BinaryDigit
  BinaryDigit [BinaryDigitsAndUnderscores] BinaryDigit

BinaryDigit:
  (one of)
  0 1

BinaryDigitsAndUnderscores:
  BinaryDigitOrUnderscore {BinaryDigitOrUnderscore}

BinaryDigitOrUnderscore:
  BinaryDigit
  _
```

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. The decimal literal `2147483648` may appear only as the operand of the unary minus operator `-` (§15.15.4).

It is a compile-time error if the decimal literal `2147483648` appears anywhere other than as the operand of the unary minus operator; or if a decimal literal of type `int` is larger than `2147483648` (2^{31}).

The largest positive hexadecimal, octal, and binary literals of type `int` - each of which represents the decimal value `2147483647` (2^{31}-1) - are respectively:

- `0x7fff_ffff`,
- `0177_7777_7777`, and
- `0b0111_1111_1111_1111_1111_1111_1111_1111`

The most negative hexadecimal, octal, and binary literals of type `int` - each of which represents the decimal value `-2147483648` (-2^{31}) - are respectively:

- `0x8000_0000`,
- `0200_0000_0000`, and
- `0b1000_0000_0000_0000_0000_0000_0000_0000`

The following hexadecimal, octal, and binary literals represent the decimal value `-1`:

- `0xffff_ffff`,
- `0377_7777_7777`, and
- `0b1111_1111_1111_1111_1111_1111_1111_1111`

It is a compile-time error if a hexadecimal, octal, or binary `int` literal does not fit in 32 bits.

The largest decimal literal of type `long` is `9223372036854775808L` (2^{63}).

All decimal literals from `0L` to `9223372036854775807L` may appear anywhere a `long` literal may appear. The decimal literal `9223372036854775808L` may appear only as the operand of the unary minus operator `-` (§15.15.4).

It is a compile-time error if the decimal literal `9223372036854775808L` appears anywhere other than as the operand of the unary minus operator; or if a decimal literal of type `long` is larger than `9223372036854775808L` (2^{63}).

The largest positive hexadecimal, octal, and binary literals of type `long` - each of which represents the decimal value `9223372036854775807L` (2^{63}-1) - are respectively:

- `0x7fff_ffff_ffff_ffffL`,
- `07_7777_7777_7777_7777_7777L`, and
- `0b0111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111L`

The most negative hexadecimal, octal, and binary literals of type `long` - each of which represents the decimal value `-9223372036854775808L` (-2^{63}) - are respectively:

- `0x8000_0000_0000_0000L`, and
- `010_0000_0000_0000_0000_0000L`, and
- `0b1000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000L`

The following hexadecimal, octal, and binary literals represent the decimal value `-1L`:

- `0xffff_ffff_ffff_ffffL`,
- `017_7777_7777_7777_7777_7777L`, and
- `0b1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111L`

It is a compile-time error if a hexadecimal, octal, or binary `long` literal does not fit in 64 bits.

Examples of `int` literals:

```
0    2    0372    0xDada_Cafe    1996    0x00_FF__00_FF
```

Examples of `long` literals:

```
0l    0777L    0x100000000L    2_147_483_648L    0xC0B0L
```

3.10.2. Floating-Point Literals

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

A floating-point literal may be expressed in decimal (base 10) or hexadecimal (base 16).

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. The exponent, if present, is indicated by the ASCII letter `e` or `E` followed by an optionally signed integer.

For hexadecimal floating-point literals, at least one digit is required (in either the whole number or the fraction part), and the exponent is mandatory, and the float type suffix is optional. The exponent is indicated by the ASCII letter `p` or `P` followed by an optionally signed integer.

Underscores are allowed as separators between digits that denote the whole-number part, and between digits that denote the fraction part, and between digits that denote the exponent.

```
FloatingPointLiteral:
  DecimalFloatingPointLiteral
  HexadecimalFloatingPointLiteral

DecimalFloatingPointLiteral:
  Digits . [Digits] [ExponentPart] [FloatTypeSuffix]
  . Digits [ExponentPart] [FloatTypeSuffix]
  Digits ExponentPart [FloatTypeSuffix]
  Digits [ExponentPart] FloatTypeSuffix

ExponentPart:
  ExponentIndicator SignedInteger

ExponentIndicator:
  (one of)
  e E

SignedInteger:
  [Sign] Digits

Sign:
  (one of)
  + -

FloatTypeSuffix:
  (one of)
  f F d D
```

```
HexadecimalFloatingPointLiteral:
  HexSignificand BinaryExponent [FloatTypeSuffix]

HexSignificand:
  HexNumeral [.]
  0 x [HexDigits] . HexDigits
  0 X [HexDigits] . HexDigits

BinaryExponent:
  BinaryExponentIndicator SignedInteger

BinaryExponentIndicator:
  (one of)
  p P
```

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`.

The elements of the types `float` and `double` are those values that can be represented using the IEEE 754 binary32 and IEEE 754 binary64 floating-point formats, respectively (§4.2.3).

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 and smallest positive literals of type `float` are as follows:

- The largest positive finite `float` value is numerically equal to $(2 - 2^{-23}) \cdot 2^{127}$.

  The shortest decimal literal which rounds to this value is `3.4028235e38f`.

  A hexadecimal literal for this value is `0x1.fffffeP+127f`.

- The smallest positive finite non-zero `float` value is numerically equal to 2^{-149}.

  The shortest decimal literal which rounds to this value is `1.4e-45f`.

  Two hexadecimal literals for this value are `0x0.000002P-126f` and `0x1.0P-149f`.

The largest and smallest positive literals of type `double` are as follows:

- The largest positive finite `double` value is numerically equal to $(2 - 2^{-52}) \cdot 2^{1023}$.

The shortest decimal literal which rounds to this value is `1.7976931348623157e308`.

A hexadecimal literal for this value is `0x1.f_ffff_ffff_ffffP+1023`.

- The smallest positive finite non-zero `double` value is numerically equal to 2^{-1074}.

  The shortest decimal literal which rounds to this value is `4.9e-324`.

  Two hexadecimal literals for this value are `0x0.0_0000_0000_0001P-1022` and `0x1.0P-1074`.

It is a compile-time error if a non-zero 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`.

It is a compile-time error if a non-zero 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 non-zero floating-point literal has a small value that, on rounded conversion to its internal representation, becomes a non-zero subnormal number.

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

Examples of `float` literals:

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

Examples of `double` literals:

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

3.10.3. Boolean Literals

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

```
BooleanLiteral:
  (one of)
  true false
```

A boolean literal is always of type `boolean` (§4.2.5).

3.10.4. Character Literals

A *character literal* is expressed as a character or an escape sequence (§3.10.7), enclosed in ASCII single quotes. (The single-quote, or apostrophe, character is `\u0027`.)

```
CharacterLiteral:
  ' SingleCharacter '
  ' EscapeSequence '

SingleCharacter:
  InputCharacter but not ' or \
```

A character literal is always of type `char` (§4.2.1).

The *content* of a character literal is the *SingleCharacter* or the *EscapeSequence* which follows the opening `'`.

It is a compile-time error for the character following the content to be other than a `'`.

It is a compile-time error for a line terminator (§3.4) to appear after the opening `'` and before the closing `'`.

The characters CR and LF are never an InputCharacter; each is recognized as constituting a LineTerminator, so may not appear in a character literal, even in the escape sequence \ LineTerminator.

The *character represented a character literal* is the content of the character literal with any escape sequence interpreted, as if by execution of `String.translateEscapes` on the content.

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.

The following are examples of `char` literals:

- `'a'`
- `'%'`
- `'\t'`
- `'\\'`
- `'\''`
- `'\u03a9'`
- `'\uFFFF'`
- `'\177'`

- `'™'`

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), so the character literal is not valid in step 3. Instead, one should use the escape sequence `'\n'`. Similarly, it is not correct to write `'\u000d'` for a character literal whose value is carriage return (CR). Instead, use `'\r'`. Finally, it is not possible to write `'\u0027'` for a character literal containing an apostrophe (`'`).

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 such as newlines may be represented by escape sequences (§3.10.7).

```
StringLiteral:
  " {StringCharacter} "

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

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

The *content* of a string literal is the sequence of characters that begins immediately after the opening " and ends immediately before the matching closing ".

It is a compile-time error for a line terminator (§3.4) to appear after the opening " and before the matching closing ".

The characters CR and LF are never an InputCharacter; each is recognized as constituting a LineTerminator, so may not appear in a string literal, even in the escape sequence `\` LineTerminator.

The string represented by a string literal is the content of the string literal with every escape sequence interpreted, as if by execution of `String.translateEscapes` on the content.

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), so the string literal is not valid in step 3. Instead, one should use the escape sequence `"\n"`. Similarly, it is not correct to write `"\u000d"` for a string literal containing a single carriage return (CR). Instead, use `"\r"`. Finally, it is not possible to write `"\u0022"` for a string literal containing a double quotation mark (").

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).

At run time, a string literal is a reference to an instance of class `String` (§4.3.3) that denotes the string represented by the string literal.

Moreover, a string literal always refers to the *same* instance of class `String`. This is because string literals - or, more generally, strings that are the values of constant expressions (§15.29) - are "interned" so as to share unique instances, as if by execution of the method `String.intern` (§12.5).

Example 3.10.5-1. String Literals

The 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.println(hello == "Hello");
        System.out.println(Other.hello == hello);
        System.out.println(other.Other.hello == hello);
        System.out.println(hello == ("Hel"+"lo"));
        System.out.println(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 { public static String hello = "Hello"; }
```

produces the output:

```
true
true
true
true
```

```
false
true
```

This example illustrates six points:

- *String literals in the same class and package represent references to the same `String` object (§4.3.1).*
- *String literals in different classes in the same package represent references to the same `String` object.*
- *String literals in different classes in different packages likewise represent references to the same `String` object.*
- *Strings concatenated from constant expressions (§15.29) 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` object as any pre-existing string literal with the same contents.*

3.10.6. Text Blocks

A *text block* consists of zero or more characters enclosed by opening and closing delimiters. Characters may be represented by escape sequences (§3.10.7), but the newline and double quote characters that must be represented with escape sequences in a string literal (§3.10.5) may be represented directly in a text block.

```
TextBlock:
  " " " {TextBlockWhiteSpace} LineTerminator {TextBlockCharacter} " " "

TextBlockWhiteSpace:
  WhiteSpace but not LineTerminator

TextBlockCharacter:
  InputCharacter but not \
  EscapeSequence
  LineTerminator
```

The following productions from §3.3, §3.4, and §3.6 are shown here for convenience:

```
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

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

UnicodeInputCharacter:
  UnicodeEscape
  RawInputCharacter

UnicodeEscape:
  \ UnicodeMarker HexDigit HexDigit HexDigit HexDigit

RawInputCharacter:
  any Unicode character
```

A text block is always of type `String` (§4.3.3).

The *opening delimiter* is a sequence that starts with three double quote characters (`"""`), continues with zero or more space, tab, and form feed characters, and concludes with a line terminator.

The *closing delimiter* is a sequence of three double quote characters.

The *content* of a text block is the sequence of characters that begins immediately after the line terminator of the opening delimiter, and ends immediately before the first double quote of the closing delimiter.

Unlike in a string literal (§3.10.5), it is *not* a compile-time error for a line terminator to appear in the content of a text block.

Example 3.10.6-1. Text Blocks

When multi-line strings are desired, a text block is usually more readable than a concatenation of string literals. For example, compare these alternative representations of a snippet of HTML:

```
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello, world</p>\n" +
              "    </body>\n" +
              "</html>\n";

String html = """
              <html>
                  <body>
                      <p>Hello, world</p>
                  </body>
              </html>
```

```
            """;
```

The following are examples of text blocks:

```
class Test {
    public static void main(String[] args) {
        // The six characters w i n t e r
        String season = """
                        winter""";

        // The seven characters w i n t e r LF
        String period = """
                        winter
                        """;

        // The ten characters H i , SP " B o b " LF
        String greeting = """
                          Hi, "Bob"
                          """;

        // The eleven characters H i , LF SP " B o b " LF
        String salutation = """
                            Hi,
                             "Bob"
                            """;

        // The empty string (zero length)
        String empty = """
                       """;

        // The two characters " LF
        String quote = """
                       "
                       """;

        // The two characters \ LF
        String backslash = """
                           \\
                           """;
    }
}
```

Using the escape sequences \n and \" to represent a newline character and a double quote character, respectively, is permitted in a text block, though not usually necessary. The exception is where three consecutive double quote characters appear that are not intended to be the closing delimiter """ - in this case, it is necessary to escape at least one of the double quote characters in order to avoid mimicking the closing delimiter.

Example 3.10.6-2. Escape sequences in text blocks

In the following program, the value of the `story` variable would be less readable if individual double quote characters were escaped:

```
class Story1 {
    public static void main(String[] args) {
        String story = """
            "When I use a word," Humpty Dumpty said,
            in rather a scornful tone, "it means just what I
            choose it to mean - neither more nor less."
            "The question is," said Alice, "whether you
            can make words mean so many different things."
            "The question is," said Humpty Dumpty,
            "which is to be master - that's all."
        """;
    }
}
```

If the program is modified to place the closing delimiter on the last line of the content, then an error occurs because the first three consecutive double quote characters on the last line are translated (§3.2) into the closing delimiter """ and thus a stray double quote character remains:

```
class Story2 {
    public static void main(String[] args) {
        String story = """
            "When I use a word," Humpty Dumpty said,
            in rather a scornful tone, "it means just what I
            choose it to mean - neither more nor less."
            "The question is," said Alice, "whether you
            can make words mean so many different things."
            "The question is," said Humpty Dumpty,
            "which is to be master - that's all.""";  // error
    }
}
```

The error can be avoided by escaping the final double quote character in the content:

```
class Story3 {
    public static void main(String[] args) {
        String story = """
```

```
            "When I use a word," Humpty Dumpty said,
            in rather a scornful tone, "it means just what I
            choose it to mean - neither more nor less."
            "The question is," said Alice, "whether you
            can make words mean so many different things."
            "The question is," said Humpty Dumpty,
            "which is to be master - that's all.""";  // OK
    }
}
```

If a text block is intended to denote another text block, then it is recommended to escape the first double quote character of the embedded opening and closing delimiters:

```
class Code {
    public static void main(String[] args) {
        String text = """
            The quick brown fox jumps over the lazy dog
        """;

        String code =
            """
            String text = \"""
                The quick brown fox jumps over the lazy dog
            \""";
            """;
    }
}
```

The string represented by a text block is *not* the literal sequence of characters in the content. Instead, the string represented by a text block is the result of applying the following transformations to the content, in order:

1. Line terminators are *normalized* to the ASCII LF character, as follows:
   - An ASCII CR character followed by an ASCII LF character is translated to an ASCII LF character.
   - An ASCII CR character is translated to an ASCII LF character.
2. Incidental white space is removed, as if by execution of `String.stripIndent` on the characters resulting from step 1.
3. Escape sequences are interpreted, as if by execution of `String.translateEscapes` on the characters resulting from step 2.

When this specification says that a text block *contains* a particular character or sequence of characters, or that a particular character or sequence of characters is *in* a text block, it means that the string represented by the text block (as opposed to the literal sequence of characters in the content) contains the character or sequence of characters.

Example 3.10.6-3. Order of transformations on text block content

Interpreting escape sequences last allows programmers to use `\n`, `\f`, and `\r` for vertical formatting of a string without affecting the normalization of line terminators, and to use `\b` and `\t` for horizontal formatting of a string without affecting the removal of incidental white space. For example, consider this text block that mentions the escape sequence `\r` (CR):

```
String html = """
              <html>\r
                  <body>\r
                      <p>Hello, world</p>\r
                  </body>\r
              </html>\r
              """;
```

The `\r` escape sequences are not interpreted until after the line terminators have been normalized to LF. Using Unicode escapes to visualize LF (`\u000A`) and CR (`\u000D`), and using `|` to visualize the left margin, the string represented by the text block is:

```
|<html>\u000D\u000A
|    <body>\u000D\u000A
|        <p>Hello, world</p>\u000D\u000A
|    </body>\u000D\u000A
|</html>\u000D\u000A
```

At run time, a text block is a reference to an instance of class `String` that denotes the string represented by the text block.

Moreover, a text block always refers to the *same* instance of class `String`. This is because the strings represented by text blocks - or, more generally, strings that are the values of constant expressions (§15.29) - are "interned" so as to share unique instances, as if by execution of the method `String.intern` (§12.5).

Example 3.10.6-4. Text blocks evaluate to `String`

Text blocks can be used wherever an expression of type `String` is allowed, such as in string concatenation (§15.18.1), in the invocation of methods on instances of `String`, and in annotations with `String` elements:

```
System.out.println("ab" + """
                           cde
                           """);

String cde = """
```

```
            abcde""".substring(2);

String math = """
              1+1 equals \
              """ + String.valueOf(2);

@Preconditions("""
    rate > 0 &&
    rate <= MAX_REFRESH_RATE
""")
public void setRefreshRate(int rate) { ... }
```

3.10.7. Escape Sequences

In character literals, string literals, and text blocks (§3.10.4, §3.10.5, §3.10.6), the *escape sequences* allow for the representation of some nongraphic characters without using Unicode escapes (§3.3), as well as the single quote, double quote, and backslash characters.

```
EscapeSequence:
  \ b (backspace BS, Unicode \u0008)
  \ s (space SP, Unicode \u0020)
  \ t (horizontal tab HT, Unicode \u0009)
  \ n (linefeed LF, Unicode \u000a)
  \ f (form feed FF, Unicode \u000c)
  \ r (carriage return CR, Unicode \u000d)
  \ LineTerminator (line continuation, no Unicode representation)
  \ " (double quote ", Unicode \u0022)
  \ ' (single quote ', Unicode \u0027)
  \ \ (backslash \, Unicode \u005c)
  OctalEscape (octal value, Unicode \u0000 to \u00ff)

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 OctalDigit production above comes from §3.10.1. Octal escapes are provided for compatibility with C, but can express only Unicode values \u0000 through \u00FF, so Unicode escapes are usually preferred.

It is a compile-time error if the character following a backslash in an escape sequence is not a *LineTerminator* or an ASCII `b`, `s`, `t`, `n`, `f`, `r`, `"`, `'`, `\`, `0`, `1`, `2`, `3`, `4`, `5`, `6`, or `7`.

An escape sequence in the content of a character literal, string literal, or text block is *interpreted* by replacing its `\` and trailing character(s) with the single character denoted by the Unicode escape in the *EscapeSequence* grammar. The line continuation escape sequence has no corresponding Unicode escape, so is interpreted by replacing it with nothing.

The line continuation escape sequence can appear in a text block, but cannot appear in a character literal or a string literal because each disallows a *LineTerminator*.

3.10.8. The Null Literal

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

```
NullLiteral:
  null
```

A null literal is always of the null type (§4.1).

3.11. Separators

Twelve tokens, formed from ASCII characters, are the *separators* (punctuators).

```
Separator:
  (one of)

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

3.12. Operators

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

Operator:
(one of)

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