ON THIS PAGE

Lexical Structure

The *lexical structure* of Swift describes what sequence of characters form valid tokens of the language. These valid tokens form the lowest-level building blocks of the language and are used to describe the rest of the language in subsequent chapters. A token consists of an identifier, keyword, punctuation, literal, or operator.

In most cases, tokens are generated from the characters of a Swift source file by considering the longest possible substring from the input text, within the constraints of the grammar that are specified below. This behavior is referred to as *longest match* or *maximal munch*.

Whitespace and Comments

Whitespace has two uses: to separate tokens in the source file and to help determine whether an operator is a prefix or postfix (see Operators), but is otherwise ignored. The following characters are considered whitespace: space (U+0020), line feed (U+000A), carriage return (U+000D), horizontal tab (U+0009), vertical tab (U+000B), form feed (U+000C) and null (U+0000).

Comments are treated as whitespace by the compiler. Single line comments begin with `//` and continue until a line feed (U+000A) or carriage return (U+000D). Multiline comments begin with `/*` and end with `*/`. Nesting multiline comments is allowed, but the comment markers must be balanced.

Comments can contain additional formatting and markup, as described in Markup Formatting Reference.

GRAMMAR OF WHITESPACE

whitespace → whitespace-item whitespace$_{opt}$
whitespace-item → line-break
whitespace-item → comment
whitespace-item → multiline-comment
whitespace-item → U+0000, U+0009, U+000B, U+000C, or U+0020

line-break → U+000A
line-break → U+000D
line-break → U+000D followed by U+000A

comment → **//** comment-text line-break
multiline-comment → **/*** multiline-comment-text ***/**

comment-text → comment-text-item comment-text$_{opt}$
comment-text-item → Any Unicode scalar value except U+000A or U+000D

multiline-comment-text → multiline-comment-text-item multiline-comment-text$_{opt}$
multiline-comment-text-item → multiline-comment
multiline-comment-text-item → comment-text-item
multiline-comment-text-item → Any Unicode scalar value except **/*** or ***/**

Identifiers

Identifiers begin with an uppercase or lowercase letter A through Z, an underscore (_), a noncombining alphanumeric Unicode character in the Basic Multilingual Plane, or a character outside the Basic Multilingual Plane that isn't in a Private Use Area. After the first character, digits and combining Unicode characters are also allowed.

To use a reserved word as an identifier, put a backtick (`) before and after it. For example, `class` is not a valid identifier, but `` `class` `` is valid. The backticks aren't considered part of the identifier; `` `x` `` and `x` have the same meaning.

Inside a closure with no explicit parameter names, the parameters are implicitly named `$0`, `$1`, `$2`, and so on. These names are valid identifiers within the scope of the closure.

GRAMMAR OF AN IDENTIFIER

identifier → identifier-head identifier-characters$_{opt}$
identifier → **`** identifier-head identifier-characters$_{opt}$ **`**

identifier → implicit-parameter-name
identifier-list → identifier | identifier **,** identifier-list

identifier-head → Upper- or lowercase letter A through Z
identifier-head → **_**
identifier-head → U+00A8, U+00AA, U+00AD, U+00AF, U+00B2–U+00B5, or U+00B7–U+00BA
identifier-head → U+00BC–U+00BE, U+00C0–U+00D6, U+00D8–U+00F6, or U+00F8–U+00FF
identifier-head → U+0100–U+02FF, U+0370–U+167F, U+1681–U+180D, or U+180F–U+1DBF
identifier-head → U+1E00–U+1FFF
identifier-head → U+200B–U+200D, U+202A–U+202E, U+203F–U+2040, U+2054, or U+2060–U+206F
identifier-head → U+2070–U+20CF, U+2100–U+218F, U+2460–U+24FF, or U+2776–U+2793
identifier-head → U+2C00–U+2DFF or U+2E80–U+2FFF
identifier-head → U+3004–U+3007, U+3021–U+302F, U+3031–U+303F, or U+3040–U+D7FF
identifier-head → U+F900–U+FD3D, U+FD40–U+FDCF, U+FDF0–U+FE1F, or U+FE30–U+FE44
identifier-head → U+FE47–U+FFFD
identifier-head → U+10000–U+1FFFD, U+20000–U+2FFFD, U+30000–U+3FFFD, or U+40000–U+4FFFD
identifier-head → U+50000–U+5FFFD, U+60000–U+6FFFD, U+70000–U+7FFFD, or U+80000–U+8FFFD
identifier-head → U+90000–U+9FFFD, U+A0000–U+AFFFD, U+B0000–U+BFFFD, or U+C0000–U+CFFFD
identifier-head → U+D0000–U+DFFFD or U+E0000–U+EFFFD

identifier-character → Digit 0 through 9
identifier-character → U+0300–U+036F, U+1DC0–U+1DFF, U+20D0–U+20FF, or U+FE20–U+FE2F
identifier-character → identifier-head
identifier-characters → identifier-character identifier-characters$_{opt}$

implicit-parameter-name → **$** decimal-digits

Keywords and Punctuation

The following keywords are reserved and can't be used as identifiers, unless they're escaped with backticks, as described above in Identifiers. Keywords other than `inout`, `var`, and `let` can be used as parameter names in a function declaration or function call without being escaped with backticks. When a member has the same name as a keyword, references to that member don't need to be escaped with backticks, except when there's ambiguity between referring to the member and using the keyword—for example, `self`, `Type`, and `Protocol` have special meaning in an explicit member expression, so they must be escaped with backticks in that context.

- Keywords used in declarations: `associatedtype`, `class`, `deinit`, `enum`, `extension`, `fileprivate`, `func`, `import`, `init`, `inout`, `internal`, `let`, `open`, `operator`, `private`, `protocol`, `public`, `static`, `struct`, `subscript`, `typealias`, and `var`.
- Keywords used in statements: `break`, `case`, `continue`, `default`, `defer`, `do`, `else`, `fallthrough`, `for`, `guard`, `if`, `in`, `repeat`, `return`, `switch`, `where`, and `while`.
- Keywords used in expressions and types: `as`, `Any`, `catch`, `false`, `is`, `nil`, `rethrows`, `super`, `self`, `Self`, `throw`, `throws`, `true`, and `try`.
- Keywords used in patterns: `_`.
- Keywords that begin with a number sign (`#`): `#available`, `#colorLiteral`, `#column`, `#else`, `#elseif`, `#endif`, `#error`, `#file`, `#fileLiteral`, `#function`, `#if`, `#imageLiteral`, `#line`, `#selector`, `#sourceLocation`, and `#warning`.
- Keywords reserved in particular contexts: `associativity`, `convenience`, `dynamic`, `didSet`, `final`, `get`, `infix`, `indirect`, `lazy`, `left`, `mutating`, `none`, `nonmutating`, `optional`, `override`, `postfix`, `precedence`, `prefix`, `Protocol`, `required`, `right`, `set`, `Type`, `unowned`, `weak`, and `willSet`. Outside the context in which they appear in the grammar, they can be used as identifiers.

The following tokens are reserved as punctuation and can't be used as custom operators: `(`, `)`, `{`, `}`, `[`, `]`, `.`, `,`, `:`, `;`, `=`, `@`, `#`, `&` (as a prefix operator), `->`, `` ` ``, `?`, and `!` (as a postfix operator).

Literals

A *literal* is the source code representation of a value of a type, such as a number or string.

The following are examples of literals:

```
42              // Integer literal
3.14159         // Floating-point literal
"Hello, world!" // String literal
true            // Boolean literal
```

A literal doesn't have a type on its own. Instead, a literal is parsed as having infinite precision and Swift's type inference attempts to infer a type for the literal. For example, in the declaration `let x: Int8 = 42`, Swift uses the explicit type annotation (`: Int8`) to infer that the type of the integer literal `42` is `Int8`. If there isn't suitable type information available, Swift infers that the literal's type is one of the default literal types defined in the Swift standard library. The default types are `Int` for integer literals, `Double` for floating-point literals, `String` for string literals, and `Bool` for Boolean literals. For example, in the declaration `let str = "Hello, world"`, the default inferred type of the string literal `"Hello, world"` is `String`.

When specifying the type annotation for a literal value, the annotation's type must be a type that can be instantiated from that literal value. That is, the type must conform to one of the following Swift standard library protocols: `ExpressibleByIntegerLiteral` for integer literals, `ExpressibleByFloatLiteral` for floating-point literals, `ExpressibleByStringLiteral` for string literals, `ExpressibleByBooleanLiteral` for Boolean literals, `ExpressibleByUnicodeScalarLiteral` for string literals that contain only a single Unicode scalar, and `ExpressibleByExtendedGraphemeClusterLiteral` for string literals that contain only a single extended grapheme cluster. For example, `Int8` conforms to the `ExpressibleByIntegerLiteral` protocol, and therefore it can be used in the type annotation for the integer literal `42` in the declaration `let x: Int8 = 42`.

GRAMMAR OF A LITERAL

literal → numeric-literal | string-literal | boolean-literal | nil-literal

numeric-literal → **-**$_{opt}$ integer-literal | **-**$_{opt}$ floating-point-literal
boolean-literal → **true** | **false**
nil-literal → **nil**

Integer Literals

Integer literals represent integer values of unspecified precision. By default, integer literals are expressed in decimal; you can specify an alternate base using a prefix. Binary literals begin with `0b`, octal literals begin with `0o`, and hexadecimal literals begin with `0x`.

Decimal literals contain the digits `0` through `9`. Binary literals contain `0` and `1`, octal literals contain `0` through `7`, and hexadecimal literals contain `0` through `9` as well as `A` through `F` in upper- or lowercase.

Negative integers literals are expressed by prepending a minus sign (`–`) to an integer literal, as in `–42`.

Underscores (`_`) are allowed between digits for readability, but they're ignored and therefore don't affect the value of the literal. Integer literals can begin with leading zeros (`0`), but they're likewise ignored and don't affect the base or value of the literal.

Unless otherwise specified, the default inferred type of an integer literal is the Swift standard library type `Int`. The Swift standard library also defines types for various sizes of signed and unsigned integers, as described in Integers.

> GRAMMAR OF AN INTEGER LITERAL
>
> *integer-literal* → binary-literal
> *integer-literal* → octal-literal
> *integer-literal* → decimal-literal
> *integer-literal* → hexadecimal-literal
>
> *binary-literal* → **0b** binary-digit binary-literal-characters$_{opt}$
> *binary-digit* → Digit 0 or 1
> *binary-literal-character* → binary-digit | **_**
> *binary-literal-characters* → binary-literal-character binary-literal-characters$_{opt}$
>
> *octal-literal* → **0o** octal-digit octal-literal-characters$_{opt}$
> *octal-digit* → Digit 0 through 7
> *octal-literal-character* → octal-digit | **_**
> *octal-literal-characters* → octal-literal-character octal-literal-characters$_{opt}$
>
> *decimal-literal* → decimal-digit decimal-literal-characters$_{opt}$
> *decimal-digit* → Digit 0 through 9

decimal-digits → decimal-digit decimal-digits$_{opt}$
decimal-literal-character → decimal-digit | **_**
decimal-literal-characters → decimal-literal-character decimal-literal-characters$_{opt}$

hexadecimal-literal → **0x** hexadecimal-digit hexadecimal-literal-characters$_{opt}$
hexadecimal-digit → Digit 0 through 9, a through f, or A through F
hexadecimal-literal-character → hexadecimal-digit | **_**
hexadecimal-literal-characters → hexadecimal-literal-character hexadecimal-literal-characters$_{opt}$

Floating-Point Literals

Floating-point literals represent floating-point values of unspecified precision.

By default, floating-point literals are expressed in decimal (with no prefix), but they can also be expressed in hexadecimal (with a `0x` prefix).

Decimal floating-point literals consist of a sequence of decimal digits followed by either a decimal fraction, a decimal exponent, or both. The decimal fraction consists of a decimal point (`.`) followed by a sequence of decimal digits. The exponent consists of an upper- or lowercase `e` prefix followed by a sequence of decimal digits that indicates what power of 10 the value preceding the `e` is multiplied by. For example, `1.25e2` represents 1.25 x 10^2, which evaluates to `125.0`. Similarly, `1.25e-2` represents 1.25 x 10^{-2}, which evaluates to `0.0125`.

Hexadecimal floating-point literals consist of a `0x` prefix, followed by an optional hexadecimal fraction, followed by a hexadecimal exponent. The hexadecimal fraction consists of a decimal point followed by a sequence of hexadecimal digits. The exponent consists of an upper- or lowercase `p` prefix followed by a sequence of decimal digits that indicates what power of 2 the value preceding the `p` is multiplied by. For example, `0xFp2` represents 15 x 2^2, which evaluates to `60`. Similarly, `0xFp-2` represents 15 x 2^{-2}, which evaluates to `3.75`.

Negative floating-point literals are expressed by prepending a minus sign (–) to a floating-point literal, as in `-42.5`.

Underscores (_) are allowed between digits for readability, but they're ignored and therefore don't affect the value of the literal. Floating-point literals can begin with leading zeros (`0`), but they're likewise ignored and don't affect the base or value of the literal.

Unless otherwise specified, the default inferred type of a floating-point literal is the Swift standard library type `Double`, which represents a 64-bit floating-point number. The Swift standard library also defines a `Float` type, which represents a 32-bit floating-point number.

GRAMMAR OF A FLOATING-POINT LITERAL

floating-point-literal → decimal-literal decimal-fraction$_{opt}$ decimal-exponent$_{opt}$
floating-point-literal → hexadecimal-literal hexadecimal-fraction$_{opt}$ hexadecimal-exponent

decimal-fraction → **.** decimal-literal
decimal-exponent → floating-point-e sign$_{opt}$ decimal-literal

hexadecimal-fraction → **.** hexadecimal-digit hexadecimal-literal-characters$_{opt}$
hexadecimal-exponent → floating-point-p sign$_{opt}$ decimal-literal

floating-point-e → **e** | **E**
floating-point-p → **p** | **P**
sign → **+** | **–**

String Literals

A string literal is a sequence of characters surrounded by quotation marks. A single-line string literal is surrounded by double quotation marks and has the following form:

```
"characters"
```

String literals can't contain an unescaped double quotation mark ("), an unescaped backslash (\), a carriage return, or a line feed.

A multiline string literal is surrounded by three double quotation marks and has the following form:

```
"""
characters
"""
```

Unlike a single-line string literal, a multiline string literal can contain unescaped double quotation marks ("), carriage returns, and line feeds. It can't contain three unescaped double quotation marks next to each other.

The line break after the """ that begins the multiline string literal is not part of the string. The line break before the """ that ends the literal is also not part of the string. To make a multiline string literal that begins or ends with a line feed, write a blank line as its first or last line.

A multiline string literal can be indented using any combination of spaces and tabs; this indentation is not included in the string. The """ that ends the literal determines the indentation: Every nonblank line in the literal must begin with exactly the same indentation that appears before the closing """; there's no conversion between tabs and spaces. You can include additional spaces and tabs after that indentation; those spaces and tabs appear in the string.

Line breaks in a multiline string literal are normalized to use the line feed character. Even if your source file has a mix of carriage returns and line feeds, all of the line breaks in the string will be the same.

In a multiline string literal, writing a backslash (`\`) at the end of a line omits that line break from the string. Any whitespace between the backslash and the line break is also omitted. You can use this syntax to hard wrap a multiline string literal in your source code, without changing the value of the resulting string.

Special characters can be included in string literals of both the single-line and multiline forms using the following escape sequences:

- Null character (`\0`)
- Backslash (`\\`)
- Horizontal tab (`\t`)
- Line feed (`\n`)
- Carriage return (`\r`)
- Double quotation mark (`\"`)
- Single quotation mark (`\'`)
- Unicode scalar (`\u{`*n*`}`), where *n* is a hexadecimal number that has one to eight digits

The value of an expression can be inserted into a string literal by placing the expression in parentheses after a backslash (`\`). The interpolated expression can contain a string literal, but can't contain an unescaped backslash, a carriage return, or a line feed.

For example, all of the following string literals have the same value:

```
"1 2 3"
"1 2 \("3")"
"1 2 \(3)"
"1 2 \(1 + 2)"
let x = 3; "1 2 \(x)"
```

A string delimited by extended delimiters is a sequence of characters surrounded by quotation marks and a balanced set of one or more number signs (#). A string delimited by extended delimiters has the following forms:

```
#"characters"#

#"""
characters
"""#
```

Special characters in a string delimited by extended delimiters appear in the resulting string as normal characters rather than as special characters. You can use extended delimiters to create strings with characters that would ordinarily have a special effect such as generating a string interpolation, starting an escape sequence, or terminating the string.

The following example shows a string literal and a string delimited by extended delimiters that create equivalent string values:

```
let string = #"\(x) \ " \u{2603}"#
let escaped = "\\(x) \\ \" \\u{2603}"
print(string)
// Prints "\(x) \ " \u{2603}"
print(string == escaped)
// Prints "true"
```

If you use more than one number sign to form a string delimited by extended delimiters, don't place whitespace in between the number signs:

```
print(###"Line 1\###nLine 2"###) // OK
print(# # #"Line 1\# # #nLine 2"# # #) // Error
```

Multiline string literals that you create using extended delimiters have the same indentation requirements as regular multiline string literals.

The default inferred type of a string literal is `String`. For more information about the `String` type, see Strings and Characters and `String`.

String literals that are concatenated by the `+` operator are concatenated at compile time. For example, the values of `textA` and `textB` in the example below are identical—no runtime concatenation is performed.

```
let textA = "Hello " + "world"
let textB = "Hello world"
```

GRAMMAR OF A STRING LITERAL

string-literal → static-string-literal | interpolated-string-literal

string-literal-opening-delimiter → extended-string-literal-delimiter$_{opt}$ **"**
string-literal-closing-delimiter → **"** extended-string-literal-delimiter$_{opt}$

static-string-literal → string-literal-opening-delimiter quoted-text$_{opt}$ string-literal-closing-delimiter
static-string-literal → multiline-string-literal-opening-delimiter multiline-quoted-text$_{opt}$ multiline-string-literal-closing-delimiter

multiline-string-literal-opening-delimiter → extended-string-literal-delimiter **"""**
multiline-string-literal-closing-delimiter → **"""** extended-string-literal-delimiter
extended-string-literal-delimiter → **#** extended-string-literal-delimiter$_{opt}$

quoted-text → quoted-text-item quoted-text$_{opt}$
quoted-text-item → escaped-character
quoted-text-item → Any Unicode scalar value except **"**, ****, U+000A, or U+000D

multiline-quoted-text → multiline-quoted-text-item multiline-quoted-text$_{opt}$
multiline-quoted-text-item → escaped-character
multiline-quoted-text-item → Any Unicode scalar value except ****
multiline-quoted-text-item → escaped-newline

interpolated-string-literal → string-literal-opening-delimiter interpolated-text$_{opt}$ string-literal-closing-delimiter
interpolated-string-literal → multiline-string-literal-opening-delimiter interpolated-text$_{opt}$ multiline-string-literal-closing-delimiter

interpolated-text → interpolated-text-item interpolated-text$_{opt}$
interpolated-text-item → **\(** expression **)** | quoted-text-item

multiline-interpolated-text → multiline-interpolated-text-item multiline-interpolated-text$_{opt}$
multiline-interpolated-text-item → **\(** expression **)** | multiline-quoted-text-item

escape-sequence → **** extended-string-literal-delimiter
escaped-character → escape-sequence **0** | escape-sequence **** | escape-sequence **t** | escape-sequence **n** | escape-sequence **r** | escape-sequence **"** | escape-sequence **'**
escaped-character → escape-sequence **u** **{** unicode-scalar-digits **}**
unicode-scalar-digits → Between one and eight hexadecimal digits

escaped-newline → escape-sequence whitespace$_{opt}$ line-break

Operators

The Swift standard library defines a number of operators for your use, many of which are discussed in Basic Operators and Advanced Operators. The present section describes which characters can be used to define custom operators.

Custom operators can begin with one of the ASCII characters `/`, `=`, `–`, `+`, `!`, `*`, `%`, `<`, `>`, `&`, `|`, `^`, `?`, or `~`, or one of the Unicode characters defined in the grammar below (which include characters from the *Mathematical Operators*, *Miscellaneous Symbols*, and *Dingbats* Unicode blocks, among others). After the first character, combining Unicode characters are also allowed.

You can also define custom operators that begin with a dot (`.`). These operators can contain additional dots. For example, `.+.` is treated as a single operator. If an operator doesn't begin with a dot, it can't contain a dot elsewhere. For example, `+.+` is treated as the `+` operator followed by the `.+` operator.

Although you can define custom operators that contain a question mark (`?`), they can't consist of a single question mark character only. Additionally, although operators can contain an exclamation mark (`!`), postfix operators can't begin with either a question mark or an exclamation mark.

> NOTE
>
> The tokens `=`, `–>`, `//`, `/*`, `*/`, `.`, the prefix operators `<`, `&`, and `?`, the infix operator `?`, and the postfix operators `>`, `!`, and `?` are reserved. These tokens can't be overloaded, nor can they be used as custom operators.

The whitespace around an operator is used to determine whether an operator is used as a prefix operator, a postfix operator, or a binary operator. This behavior is summarized in the following rules:

- If an operator has whitespace around both sides or around neither side, it's treated as a binary operator. As an example, the `+++` operator in `a+++b` and `a +++ b` is treated as a binary operator.
- If an operator has whitespace on the left side only, it's treated as a prefix unary operator. As an example, the `+++` operator in `a +++b` is treated as a prefix unary operator.
- If an operator has whitespace on the right side only, it's treated as a postfix unary operator. As an example, the `+++` operator in `a+++ b` is treated as a postfix unary operator.
- If an operator has no whitespace on the left but is followed immediately by a dot (`.`), it's treated as a postfix unary operator. As an example, the `+++` operator in `a+++.b` is treated as a postfix unary operator (`a+++ .b` rather than `a +++ .b`).

For the purposes of these rules, the characters `(`, `[`, and `{` before an operator, the characters `)`, `]`, and `}` after an operator, and the characters `,`, `;`, and `:` are also considered whitespace.

There's one caveat to the rules above. If the `!` or `?` predefined operator has no whitespace on the left, it's treated as a postfix operator, regardless of whether it has whitespace on the right. To use the `?` as the optional-chaining operator, it must not have whitespace on the left. To use it in the ternary conditional (`? :`) operator, it must have whitespace around both sides.

In certain constructs, operators with a leading `<` or `>` may be split into two or more tokens. The remainder is treated the same way and may be split again. As a result, there's no need to use whitespace to disambiguate between the closing `>` characters in constructs like `Dictionary<String, Array<Int>>`. In this example, the closing `>` characters are not treated as a single token that may then be misinterpreted as a bit shift `>>` operator.

To learn how to define new, custom operators, see Custom Operators and Operator Declaration. To learn how to overload existing operators, see Operator Methods.

> GRAMMAR OF OPERATORS

operator → operator-head operator-characters$_{opt}$
operator → dot-operator-head dot-operator-characters

operator-head → **/** | **=** | **-** | **+** | **!** | ***** | **%** | **<** | **>** | **&** | **|** | **^** | **~** | **?**
operator-head → U+00A1–U+00A7
operator-head → U+00A9 or U+00AB
operator-head → U+00AC or U+00AE
operator-head → U+00B0–U+00B1
operator-head → U+00B6, U+00BB, U+00BF, U+00D7, or U+00F7
operator-head → U+2016–U+2017
operator-head → U+2020–U+2027
operator-head → U+2030–U+203E
operator-head → U+2041–U+2053
operator-head → U+2055–U+205E
operator-head → U+2190–U+23FF
operator-head → U+2500–U+2775
operator-head → U+2794–U+2BFF
operator-head → U+2E00–U+2E7F
operator-head → U+3001–U+3003
operator-head → U+3008–U+3020
operator-head → U+3030

operator-character → operator-head
operator-character → U+0300–U+036F
operator-character → U+1DC0–U+1DFF
operator-character → U+20D0–U+20FF
operator-character → U+FE00–U+FE0F
operator-character → U+FE20–U+FE2F
operator-character → U+E0100–U+E01EF
operator-characters → operator-character operator-characters$_{opt}$

dot-operator-head → **.**
dot-operator-character → **.** | operator-character
dot-operator-characters → dot-operator-character dot-operator-characters$_{opt}$

binary-operator → operator
prefix-operator → operator
postfix-operator → operator

〈 About the Language Reference Types 〉