Basic Operators

On This Page

An *operator* is a special symbol or phrase that you use to check, change, or combine values. For example, the addition operator (+) adds two numbers, as in `let i = 1 + 2`, and the logical AND operator (&&) combines two Boolean values, as in `if enteredDoorCode && passedRetinaScan`.

Swift supports most standard C operators and improves several capabilities to eliminate common coding errors. The assignment operator (=) doesn't return a value, to prevent it from being mistakenly used when the equal to operator (==) is intended. Arithmetic operators (+, –, *, /, % and so forth) detect and disallow value overflow, to avoid unexpected results when working with numbers that become larger or smaller than the allowed value range of the type that stores them. You can opt in to value overflow behavior by using Swift's overflow operators, as described in Overflow Operators.

Swift also provides range operators that aren't found in C, such as `a..<b` and `a...b`, as a shortcut for expressing a range of values.

This chapter describes the common operators in Swift. Advanced Operators covers Swift's advanced operators, and describes how to define your own custom operators and implement the standard operators for your own custom types.

Terminology

Operators are unary, binary, or ternary:

- *Unary* operators operate on a single target (such as –a). Unary *prefix* operators appear immediately before their target (such as `!b`), and unary *postfix* operators appear immediately after their target (such as `c!`).
- *Binary* operators operate on two targets (such as `2 + 3`) and are *infix* because they appear in between their two targets.
- *Ternary* operators operate on three targets. Like C, Swift has only one ternary operator, the ternary conditional operator (`a ? b : c`).

The values that operators affect are *operands*. In the expression `1 + 2`, the + symbol is a binary operator and its two operands are the values `1` and `2`.

Assignment Operator

The *assignment operator* (`a = b`) initializes or updates the value of `a` with the value of `b`:

```
let b = 10
var a = 5
a = b
// a is now equal to 10
```

If the right side of the assignment is a tuple with multiple values, its elements can be decomposed into multiple constants or variables at once:

```
let (x, y) = (1, 2)
// x is equal to 1, and y is equal to 2
```

Unlike the assignment operator in C and Objective-C, the assignment operator in Swift does not itself return a value. The following statement is not valid:

```
if x = y {
    // This is not valid, because x = y does not return a value.
```

```
3 }
```

This feature prevents the assignment operator (=) from being used by accident when the equal to operator (==) is actually intended. By making `if x = y` invalid, Swift helps you to avoid these kinds of errors in your code.

Arithmetic Operators

Swift supports the four standard *arithmetic operators* for all number types:

- Addition (+)
- Subtraction (–)
- Multiplication (*)
- Division (/)

```
1 + 2       // equals 3
5 - 3       // equals 2
2 * 3       // equals 6
10.0 / 2.5  // equals 4.0
```

Unlike the arithmetic operators in C and Objective-C, the Swift arithmetic operators don't allow values to overflow by default. You can opt in to value overflow behavior by using Swift's overflow operators (such as `a &+ b`). See Overflow Operators.

The addition operator is also supported for `String` concatenation:

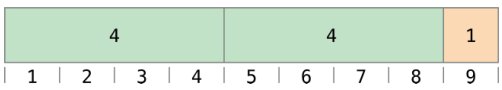
```
"hello, " + "world"  // equals "hello, world"
```

Remainder Operator

The *remainder operator* (`a % b`) works out how many multiples of `b` will fit inside `a` and returns the value that is left over (known as the *remainder*).

> NOTE
>
> The remainder operator (%) is also known as a *modulo operator* in other languages. However, its behavior in Swift for negative numbers means that, strictly speaking, it's a remainder rather than a modulo operation.

Here's how the remainder operator works. To calculate `9 % 4`, you first work out how many 4s will fit inside 9:



You can fit two 4s inside 9, and the remainder is 1 (shown in orange).

In Swift, this would be written as:

```
9 % 4    // equals 1
```

To determine the answer for `a % b`, the `%` operator calculates the following equation and returns `remainder` as its output:

`a = (b x some multiplier) + remainder`

where `some multiplier` is the largest number of multiples of `b` that will fit inside `a`.

Inserting `9` and `4` into this equation yields:

9 = (4 x 2) + 1

The same method is applied when calculating the remainder for a negative value of `a`:

```
-9 % 4   // equals -1
```

Inserting `-9` and `4` into the equation yields:

-9 = (4 x -2) + -1

giving a remainder value of `-1`.

The sign of `b` is ignored for negative values of `b`. This means that `a % b` and `a % -b` always give the same answer.

Unary Minus Operator

The sign of a numeric value can be toggled using a prefixed `-`, known as the *unary minus operator*:

```
let three = 3
let minusThree = -three       // minusThree equals -3
let plusThree = -minusThree   // plusThree equals 3, or "minus minus three"
```

The unary minus operator (`-`) is prepended directly before the value it operates on, without any white space.

Unary Plus Operator

The *unary plus operator* (`+`) simply returns the value it operates on, without any change:

```
let minusSix = -6
let alsoMinusSix = +minusSix  // alsoMinusSix equals -6
```

Although the unary plus operator doesn't actually do anything, you can use it to provide symmetry in your code for positive numbers when also using the unary minus operator for negative numbers.

Compound Assignment Operators

Like C, Swift provides *compound assignment operators* that combine assignment (`=`) with another operation. One example is the *addition assignment operator* (`+=`):

```
var a = 1
a += 2
// a is now equal to 3
```

The expression `a += 2` is shorthand for `a = a + 2`. Effectively, the addition and the assignment are combined into one operator that performs both tasks at the same time.

> NOTE
> The compound assignment operators don't return a value. For example, you can't write `let b = a += 2`.

For information about the operators provided by the Swift standard library, see Operator Declarations.

Comparison Operators

Swift supports all standard C *comparison operators*:

- Equal to (`a == b`)
- Not equal to (`a != b`)
- Greater than (`a > b`)
- Less than (`a < b`)
- Greater than or equal to (`a >= b`)
- Less than or equal to (`a <= b`)

> NOTE
>
> Swift also provides two *identity operators* (`===` and `!==`), which you use to test whether two object references both refer to the same object instance. For more information, see Classes and Structures.

Each of the comparison operators returns a `Bool` value to indicate whether or not the statement is true:

```
1 == 1   // true because 1 is equal to 1
2 != 1   // true because 2 is not equal to 1
2 > 1    // true because 2 is greater than 1
1 < 2    // true because 1 is less than 2
1 >= 1   // true because 1 is greater than or equal to 1
2 <= 1   // false because 2 is not less than or equal to 1
```

Comparison operators are often used in conditional statements, such as the `if` statement:

```
let name = "world"
if name == "world" {
    print("hello, world")
} else {
    print("I'm sorry \(name), but I don't recognize you")
}
// Prints "hello, world", because name is indeed equal to "world".
```

For more about the `if` statement, see Control Flow.

You can compare two tuples if they have the same type and the same number of values. Tuples are compared from left to right, one value at a time, until the comparison finds two values that aren't equal. Those two values are compared, and the result of that comparison determines the overall result of the tuple comparison. If all the elements are equal, then the tuples themselves are equal. For example:

```
(1, "zebra") < (2, "apple")   // true because 1 is less than 2; "zebra" and "apple" are not compared
(3, "apple") < (3, "bird")    // true because 3 is equal to 3, and "apple" is less than "bird"
(4, "dog") == (4, "dog")      // true because 4 is equal to 4, and "dog" is equal to "dog"
```

In the example above, you can see the left-to-right comparison behavior on the first line. Because `1` is less than `2`, `(1, "zebra")` is considered less than `(2, "apple")`, regardless of any other values in the tuples. It doesn't matter that `"zebra"` isn't less than `"apple"`, because the comparison is already determined by the tuples' first elements. However, when the tuples' first elements are the same, their second elements *are* compared—this is what happens on the second and third line.

Tuples can be compared with a given operator only if the operator can be applied to each value in the respective tuples. For example, as demonstrated in the code below, you can compare two tuples of type `(String, Int)` because both `String` and `Int` values can be compared using the < operator. In contrast, two tuples of type `(String, Bool)` can't be compared with the < operator because the < operator can't be applied to `Bool` values.

```
("blue", -1) < ("purple", 1)        // OK, evaluates to true
("blue", false) < ("purple", true)  // Error because < can't compare Boolean values
```

> NOTE
>
> The Swift standard library includes tuple comparison operators for tuples with fewer than seven elements. To compare tuples with seven or more elements, you must implement the comparison operators yourself.

Ternary Conditional Operator

The *ternary conditional operator* is a special operator with three parts, which takes the form `question ? answer1 : answer2`. It's a shortcut for evaluating one of two expressions based on whether `question` is true or false. If `question` is true, it evaluates `answer1` and returns its value; otherwise, it evaluates `answer2` and returns its value.

The ternary conditional operator is shorthand for the code below:

```
if question {
    answer1
} else {
    answer2
}
```

Here's an example, which calculates the height for a table row. The row height should be 50 points taller than the content height if the row has a header, and 20 points taller if the row doesn't have a header:

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight is equal to 90
```

The example above is shorthand for the code below:

```
let contentHeight = 40
let hasHeader = true
let rowHeight: Int
if hasHeader {
    rowHeight = contentHeight + 50
} else {
    rowHeight = contentHeight + 20
}
// rowHeight is equal to 90
```

The first example's use of the ternary conditional operator means that `rowHeight` can be set to the correct value on a single line of code, which is more concise than the code used in the second example.

The ternary conditional operator provides an efficient shorthand for deciding which of two expressions to consider. Use the ternary conditional operator with care, however. Its conciseness can lead to hard-to-read code if overused. Avoid combining multiple instances of the ternary conditional operator into one compound statement.

Nil-Coalescing Operator

The *nil-coalescing operator* (`a ?? b`) unwraps an optional `a` if it contains a value, or returns a default value `b` if `a` is `nil`. The expression `a` is always of an optional type. The expression `b` must match the type that is stored inside `a`.

The nil-coalescing operator is shorthand for the code below:

```
a != nil ? a! : b
```

The code above uses the ternary conditional operator and forced unwrapping (`a!`) to access the value wrapped inside `a` when `a` is not `nil`, and to return `b` otherwise. The nil-coalescing operator provides a more elegant way to encapsulate this conditional checking and unwrapping in a concise and readable form.

> NOTE
> If the value of `a` is non-`nil`, the value of `b` is not evaluated. This is known as *short-circuit evaluation*.

The example below uses the nil-coalescing operator to choose between a default color name and an optional user-defined color name:

```
let defaultColorName = "red"
var userDefinedColorName: String?   // defaults to nil

var colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName is nil, so colorNameToUse is set to the default of "red"
```

The `userDefinedColorName` variable is defined as an optional `String`, with a default value of `nil`. Because `userDefinedColorName` is of an optional type, you can use the nil-coalescing operator to consider its value. In the example above, the operator is used to determine an initial value for a `String` variable called `colorNameToUse`. Because `userDefinedColorName` is `nil`, the expression `userDefinedColorName ?? defaultColorName` returns the value of `defaultColorName`, or `"red"`.

If you assign a non-`nil` value to `userDefinedColorName` and perform the nil-coalescing operator check again, the value wrapped inside `userDefinedColorName` is used instead of the default:

```
userDefinedColorName = "green"
colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName is not nil, so colorNameToUse is set to "green"
```

Range Operators

Swift includes several *range operators*, which are shortcuts for expressing a range of values.

Closed Range Operator

The *closed range operator* (`a...b`) defines a range that runs from `a` to `b`, and includes the values `a` and `b`. The value of `a` must not be greater than `b`.

The closed range operator is useful when iterating over a range in which you want all of the values to be used, such as with a `for-in` loop:

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
```

```
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

For more about `for-in` loops, see Control Flow.

Half-Open Range Operator

The *half-open range operator* (`a..<b`) defines a range that runs from `a` to `b`, but doesn't include `b`. It's said to be *half-open* because it contains its first value, but not its final value. As with the closed range operator, the value of `a` must not be greater than `b`. If the value of `a` is equal to `b`, then the resulting range will be empty.

Half-open ranges are particularly useful when you work with zero-based lists such as arrays, where it's useful to count up to (but not including) the length of the list:

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..<count {
    print("Person \(i + 1) is called \(names[i])")
}
// Person 1 is called Anna
// Person 2 is called Alex
// Person 3 is called Brian
// Person 4 is called Jack
```

Note that the array contains four items, but `0..<count` only counts as far as `3` (the index of the last item in the array), because it's a half-open range. For more about arrays, see Arrays.

One-Sided Ranges

The closed range operator has an alternative form for ranges that continue as far as possible in one direction—for example, a range that includes all the elements of an array from index 2 to the end of the array. In these cases, you can omit the value from one side of the range operator. This kind of range is called a *one-sided range* because the operator has a value on only one side. For example:

```
for name in names[2...] {
    print(name)
}
// Brian
// Jack

for name in names[...2] {
    print(name)
}
// Anna
// Alex
// Brian
```

The half-open range operator also has a one-sided form that's written with only its final value. Just like when you include a value on both sides, the final value isn't part of the range. For example:

```
for name in names[..<2] {
    print(name)
```

```
}
// Anna
// Alex
```

One-sided ranges can be used in other contexts, not just in subscripts. You can't iterate over a one-sided range that omits a first value, because it isn't clear where iteration should begin. You *can* iterate over a one-sided range that omits its final value; however, because the range continues indefinitely, make sure you add an explicit end condition for the loop. You can also check whether a one-sided range contains a particular value, as shown in the code below.

```
let range = ...5
range.contains(7)   // false
range.contains(4)   // true
range.contains(-1)  // true
```

Logical Operators

Logical operators modify or combine the Boolean logic values `true` and `false`. Swift supports the three standard logical operators found in C-based languages:

- Logical NOT (`!a`)
- Logical AND (`a && b`)
- Logical OR (`a || b`)

Logical NOT Operator

The *logical NOT operator* (`!a`) inverts a Boolean value so that `true` becomes `false`, and `false` becomes `true`.

The logical NOT operator is a prefix operator, and appears immediately before the value it operates on, without any white space. It can be read as "not `a`", as seen in the following example:

```
let allowedEntry = false
if !allowedEntry {
    print("ACCESS DENIED")
}
// Prints "ACCESS DENIED"
```

The phrase `if !allowedEntry` can be read as "if not allowed entry." The subsequent line is only executed if "not allowed entry" is true; that is, if `allowedEntry` is `false`.

As in this example, careful choice of Boolean constant and variable names can help to keep code readable and concise, while avoiding double negatives or confusing logic statements.

Logical AND Operator

The *logical AND operator* (`a && b`) creates logical expressions where both values must be `true` for the overall expression to also be `true`.

If either value is `false`, the overall expression will also be `false`. In fact, if the *first* value is `false`, the second value won't even be evaluated, because it can't possibly make the overall expression equate to `true`. This is known as *short-circuit evaluation*.

This example considers two `Bool` values and only allows access if both values are `true`:

```
let enteredDoorCode = true
let passedRetinaScan = false
```

```
if enteredDoorCode && passedRetinaScan {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "ACCESS DENIED"
```

Logical OR Operator

The *logical OR operator* (`a || b`) is an infix operator made from two adjacent pipe characters. You use it to create logical expressions in which only *one* of the two values has to be `true` for the overall expression to be `true`.

Like the Logical AND operator above, the Logical OR operator uses short-circuit evaluation to consider its expressions. If the left side of a Logical OR expression is `true`, the right side is not evaluated, because it can't change the outcome of the overall expression.

In the example below, the first `Bool` value (`hasDoorKey`) is `false`, but the second value (`knowsOverridePassword`) is `true`. Because one value is `true`, the overall expression also evaluates to `true`, and access is allowed:

```
let hasDoorKey = false
let knowsOverridePassword = true
if hasDoorKey || knowsOverridePassword {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "Welcome!"
```

Combining Logical Operators

You can combine multiple logical operators to create longer compound expressions:

```
if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "Welcome!"
```

This example uses multiple `&&` and `||` operators to create a longer compound expression. However, the `&&` and `||` operators still operate on only two values, so this is actually three smaller expressions chained together. The example can be read as:

If we've entered the correct door code and passed the retina scan, or if we have a valid door key, or if we know the emergency override password, then allow access.

Based on the values of `enteredDoorCode`, `passedRetinaScan`, and `hasDoorKey`, the first two subexpressions are `false`. However, the emergency override password is known, so the overall compound expression still evaluates to `true`.

> NOTE
>
> The Swift logical operators `&&` and `||` are left-associative, meaning that compound expressions with multiple logical operators evaluate the leftmost subexpression first.

Explicit Parentheses

It's sometimes useful to include parentheses when they're not strictly needed, to make the intention of a complex expression easier to read. In the door access example above, it's useful to add parentheses around the first part of the compound expression to make its intent explicit:

```
if (enteredDoorCode && passedRetinaScan) || hasDoorKey || knowsOverridePassword {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "Welcome!"
```

The parentheses make it clear that the first two values are considered as part of a separate possible state in the overall logic. The output of the compound expression doesn't change, but the overall intention is clearer to the reader. Readability is always preferred over brevity; use parentheses where they help to make your intentions clear.