Advanced Operators

On This Page

In addition to the operators described in Basic Operators, Swift provides several advanced operators that perform more complex value manipulation. These include all of the bitwise and bit shifting operators you will be familiar with from C and Objective-C.

Unlike arithmetic operators in C, arithmetic operators in Swift do not overflow by default. Overflow behavior is trapped and reported as an error. To opt in to overflow behavior, use Swift's second set of arithmetic operators that overflow by default, such as the overflow addition operator (&+). All of these overflow operators begin with an ampersand (&).

When you define your own structures, classes, and enumerations, it can be useful to provide your own implementations of the standard Swift operators for these custom types. Swift makes it easy to provide tailored implementations of these operators and to determine exactly what their behavior should be for each type you create.

You're not limited to the predefined operators. Swift gives you the freedom to define your own custom infix, prefix, postfix, and assignment operators, with custom precedence and associativity values. These operators can be used and adopted in your code like any of the predefined operators, and you can even extend existing types to support the custom operators you define.

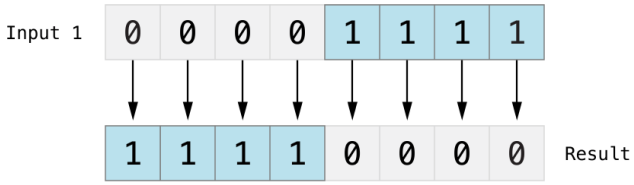
Bitwise Operators

Bitwise operators enable you to manipulate the individual raw data bits within a data structure. They are often used in low-level programming, such as graphics programming and device driver creation. Bitwise operators can also be useful when you work with raw data from external sources, such as encoding and decoding data for communication over a custom protocol.

Swift supports all of the bitwise operators found in C, as described below.

Bitwise NOT Operator

The *bitwise NOT operator* (~) inverts all bits in a number:



The bitwise NOT operator is a prefix operator, and appears immediately before the value it operates on, without any white space:

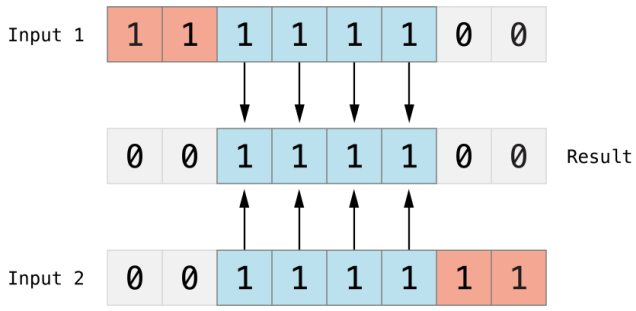
```
let initialBits: UInt8 = 0b00001111
let invertedBits = ~initialBits  // equals 11110000
```

`UInt8` integers have eight bits and can store any value between `0` and `255`. This example initializes a `UInt8` integer with the binary value `00001111`, which has its first four bits set to `0`, and its second four bits set to `1`. This is equivalent to a decimal value of `15`.

The bitwise NOT operator is then used to create a new constant called `invertedBits`, which is equal to `initialBits`, but with all of the bits inverted. Zeros become ones, and ones become zeros. The value of `invertedBits` is `11110000`, which is equal to an unsigned decimal value of `240`.

Bitwise AND Operator

The *bitwise AND operator* (`&`) combines the bits of two numbers. It returns a new number whose bits are set to `1` only if the bits were equal to `1` in *both* input numbers:

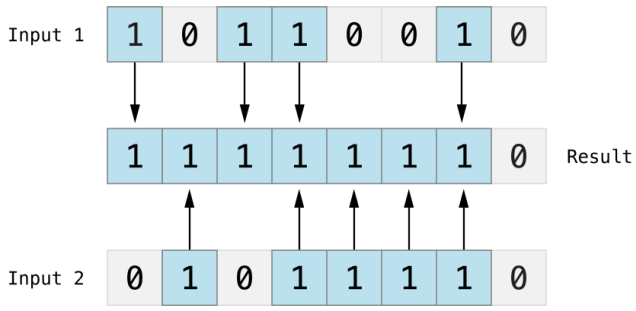


In the example below, the values of `firstSixBits` and `lastSixBits` both have four middle bits equal to `1`. The bitwise AND operator combines them to make the number `00111100`, which is equal to an unsigned decimal value of `60`:

```
let firstSixBits: UInt8 = 0b11111100
let lastSixBits: UInt8  = 0b00111111
let middleFourBits = firstSixBits & lastSixBits  // equals 00111100
```

Bitwise OR Operator

The *bitwise OR operator* (`|`) compares the bits of two numbers. The operator returns a new number whose bits are set to `1` if the bits are equal to `1` in *either* input number:

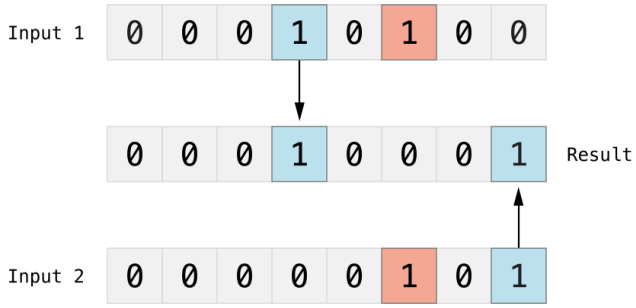


In the example below, the values of `someBits` and `moreBits` have different bits set to `1`. The bitwise OR operator combines them to make the number `11111110`, which equals an unsigned decimal of `254`:

```
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
let combinedbits = someBits | moreBits  // equals 11111110
```

Bitwise XOR Operator

The *bitwise XOR operator*, or "exclusive OR operator" (`^`), compares the bits of two numbers. The operator returns a new number whose bits are set to `1` where the input bits are different and are set to `0` where the input bits are the same:



In the example below, the values of `firstBits` and `otherBits` each have a bit set to `1` in a location that the other does not. The bitwise XOR operator sets both of these bits to `1` in its output value. All of the other bits in `firstBits` and `otherBits` match and are set to `0` in the output value:

```
let firstBits: UInt8 = 0b00010100
let otherBits: UInt8 = 0b00000101
let outputBits = firstBits ^ otherBits  // equals 00010001
```

Bitwise Left and Right Shift Operators

The *bitwise left shift operator* (<<) and *bitwise right shift operator* (>>) move all bits in a number to the left or the right by a certain number of places, according to the rules defined below.

Bitwise left and right shifts have the effect of multiplying or dividing an integer by a factor of two. Shifting an integer's bits to the left by one position doubles its value, whereas shifting it to the right by one position halves its value.

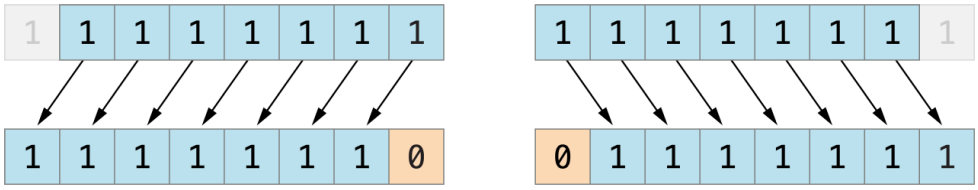
Shifting Behavior for Unsigned Integers

The bit-shifting behavior for unsigned integers is as follows:

1. Existing bits are moved to the left or right by the requested number of places.
2. Any bits that are moved beyond the bounds of the integer's storage are discarded.
3. Zeros are inserted in the spaces left behind after the original bits are moved to the left or right.

This approach is known as a *logical shift*.

The illustration below shows the results of `11111111 << 1` (which is 11111111 shifted to the left by 1 place), and `11111111 >> 1` (which is 11111111 shifted to the right by 1 place). Blue numbers are shifted, gray numbers are discarded, and orange zeros are inserted:



Here's how bit shifting looks in Swift code:

```
let shiftBits: UInt8 = 4   // 00000100 in binary
shiftBits << 1             // 00001000
shiftBits << 2             // 00010000
shiftBits << 5             // 10000000
shiftBits << 6             // 00000000
```

```
shiftBits >> 2             // 00000001
```

You can use bit shifting to encode and decode values within other data types:

```
let pink: UInt32 = 0xCC6699
let redComponent = (pink & 0xFF0000) >> 16    // redComponent is 0xCC, or 204
let greenComponent = (pink & 0x00FF00) >> 8   // greenComponent is 0x66, or 102
let blueComponent = pink & 0x0000FF           // blueComponent is 0x99, or 153
```

This example uses a `UInt32` constant called `pink` to store a Cascading Style Sheets color value for the color pink. The CSS color value `#CC6699` is written as `0xCC6699` in Swift's hexadecimal number representation. This color is then decomposed into its red (`CC`), green (`66`), and blue (`99`) components by the bitwise AND operator (`&`) and the bitwise right shift operator (`>>`).

The red component is obtained by performing a bitwise AND between the numbers `0xCC6699` and `0xFF0000`. The zeros in `0xFF0000` effectively "mask" the second and third bytes of `0xCC6699`, causing the `6699` to be ignored and leaving `0xCC0000` as the result.

This number is then shifted 16 places to the right (`>> 16`). Each pair of characters in a hexadecimal number uses 8 bits, so a move 16 places to the right will convert `0xCC0000` into `0x0000CC`. This is the same as `0xCC`, which has a decimal value of `204`.

Similarly, the green component is obtained by performing a bitwise AND between the numbers `0xCC6699` and `0x00FF00`, which gives an output value of `0x006600`. This output value is then shifted eight places to the right, giving a value of `0x66`, which has a decimal value of `102`.

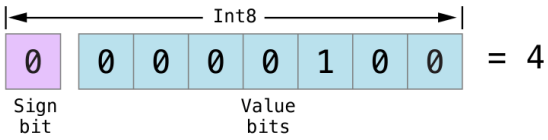
Finally, the blue component is obtained by performing a bitwise AND between the numbers `0xCC6699` and `0x0000FF`, which gives an output value of `0x000099`. There's no need to shift this to the right, as `0x000099` already equals `0x99`, which has a decimal value of `153`.

Shifting Behavior for Signed Integers

The shifting behavior is more complex for signed integers than for unsigned integers, because of the way signed integers are represented in binary. (The examples below are based on 8-bit signed integers for simplicity, but the same principles apply for signed integers of any size.)

Signed integers use their first bit (known as the *sign bit*) to indicate whether the integer is positive or negative. A sign bit of `0` means positive, and a sign bit of `1` means negative.

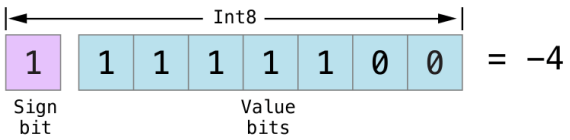
The remaining bits (known as the *value bits*) store the actual value. Positive numbers are stored in exactly the same way as for unsigned integers, counting upwards from `0`. Here's how the bits inside an `Int8` look for the number `4`:



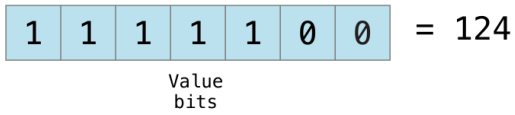
The sign bit is `0` (meaning "positive"), and the seven value bits are just the number `4`, written in binary notation.

Negative numbers, however, are stored differently. They are stored by subtracting their absolute value from `2` to the power of `n`, where `n` is the number of value bits. An eight-bit number has seven value bits, so this means `2` to the power of `7`, or `128`.

Here's how the bits inside an `Int8` look for the number `-4`:

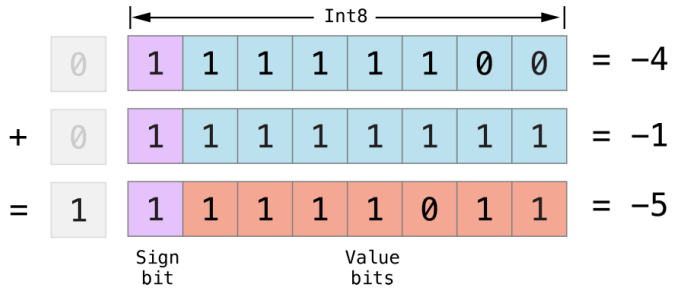


This time, the sign bit is `1` (meaning "negative"), and the seven value bits have a binary value of `124` (which is `128 - 4`):

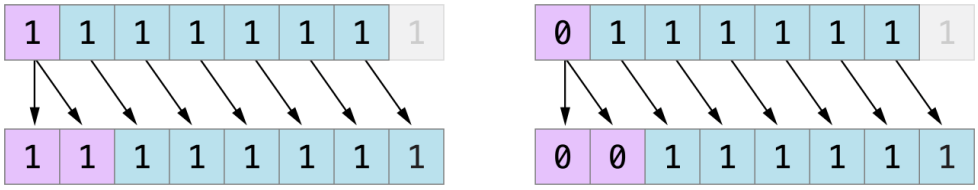


This encoding for negative numbers is known as a *two's complement* representation. It may seem an unusual way to represent negative numbers, but it has several advantages.

First, you can add –1 to –4, simply by performing a standard binary addition of all eight bits (including the sign bit), and discarding anything that doesn't fit in the eight bits once you're done:



Second, the two's complement representation also lets you shift the bits of negative numbers to the left and right like positive numbers, and still end up doubling them for every shift you make to the left, or halving them for every shift you make to the right. To achieve this, an extra rule is used when signed integers are shifted to the right: When you shift signed integers to the right, apply the same rules as for unsigned integers, but fill any empty bits on the left with the *sign bit*, rather than with a zero.



This action ensures that signed integers have the same sign after they are shifted to the right, and is known as an *arithmetic shift*.

Because of the special way that positive and negative numbers are stored, shifting either of them to the right moves them closer to zero. Keeping the sign bit the same during this shift means that negative integers remain negative as their value moves closer to zero.

Overflow Operators

If you try to insert a number into an integer constant or variable that cannot hold that value, by default Swift

reports an error rather than allowing an invalid value to be created. This behavior gives extra safety when you work with numbers that are too large or too small.

For example, the `Int16` integer type can hold any signed integer between `-32768` and `32767`. Trying to set an `Int16` constant or variable to a number outside of this range causes an error:

```
var potentialOverflow = Int16.max
// potentialOverflow equals 32767, which is the maximum value an Int16 can hold
potentialOverflow += 1
// this causes an error
```

Providing error handling when values get too large or too small gives you much more flexibility when coding for boundary value conditions.

However, when you specifically want an overflow condition to truncate the number of available bits, you can opt in to this behavior rather than triggering an error. Swift provides three arithmetic *overflow operators* that opt in to the overflow behavior for integer calculations. These operators all begin with an ampersand (`&`):

- Overflow addition (`&+`)
- Overflow subtraction (`&-`)
- Overflow multiplication (`&*`)

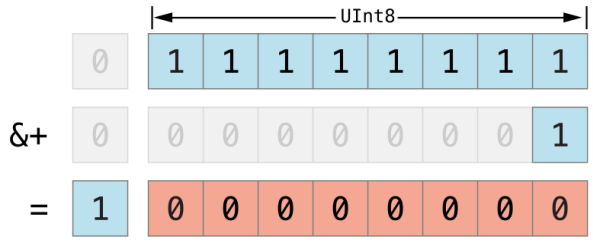
Value Overflow

Numbers can overflow in both the positive and negative direction.

Here's an example of what happens when an unsigned integer is allowed to overflow in the positive direction, using the overflow addition operator (`&+`):

```
var unsignedOverflow = UInt8.max
// unsignedOverflow equals 255, which is the maximum value a UInt8 can hold
unsignedOverflow = unsignedOverflow &+ 1
// unsignedOverflow is now equal to 0
```

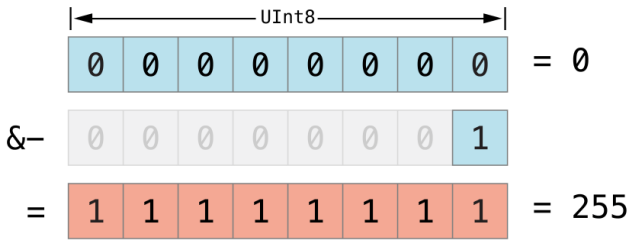
The variable `unsignedOverflow` is initialized with the maximum value a `UInt8` can hold (`255`, or `11111111` in binary). It is then incremented by `1` using the overflow addition operator (`&+`). This pushes its binary representation just over the size that a `UInt8` can hold, causing it to overflow beyond its bounds, as shown in the diagram below. The value that remains within the bounds of the `UInt8` after the overflow addition is `00000000`, or zero.



Something similar happens when an unsigned integer is allowed to overflow in the negative direction. Here's an example using the overflow subtraction operator (`&-`):

```
var unsignedOverflow = UInt8.min
// unsignedOverflow equals 0, which is the minimum value a UInt8 can hold
unsignedOverflow = unsignedOverflow &- 1
// unsignedOverflow is now equal to 255
```

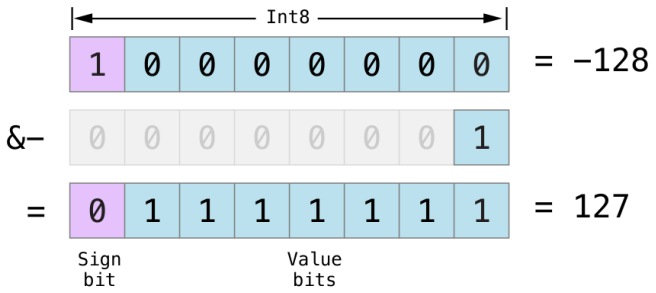
The minimum value that a `UInt8` can hold is zero, or `00000000` in binary. If you subtract `1` from `00000000` using the overflow subtraction operator (`&-`), the number will overflow and wrap around to `11111111`, or `255` in decimal.



Overflow also occurs for signed integers. All addition and subtraction for signed integers is performed in bitwise fashion, with the sign bit included as part of the numbers being added or subtracted, as described in Bitwise Left and Right Shift Operators.

```
var signedOverflow = Int8.min
// signedOverflow equals -128, which is the minimum value an Int8 can hold
signedOverflow = signedOverflow &- 1
// signedOverflow is now equal to 127
```

The minimum value that an `Int8` can hold is `-128`, or `10000000` in binary. Subtracting `1` from this binary number with the overflow operator gives a binary value of `01111111`, which toggles the sign bit and gives positive `127`, the maximum positive value that an `Int8` can hold.



For both signed and unsigned integers, overflow in the positive direction wraps around from the maximum valid integer value back to the minimum, and overflow in the negative direction wraps around from the minimum value to the maximum.

Precedence and Associativity

Operator *precedence* gives some operators higher priority than others; these operators are applied first.

Operator *associativity* defines how operators of the same precedence are grouped together—either grouped from the left, or grouped from the right. Think of it as meaning "they associate with the expression to their left," or "they associate with the expression to their right."

It is important to consider each operator's precedence and associativity when working out the order in which a compound expression will be calculated. For example, operator precedence explains why the following expression equals `17`.

```
2 + 3 % 4 * 5
// this equals 17
```

If you read strictly from left to right, you might expect the expression to be calculated as follows:

- 2 plus 3 equals 5
- 5 remainder 4 equals 1
- 1 times 5 equals 5

However, the actual answer is 17, not 5. Higher-precedence operators are evaluated before lower-precedence ones. In Swift, as in C, the remainder operator (%) and the multiplication operator (*) have a higher precedence than the addition operator (+). As a result, they are both evaluated before the addition is considered.

However, remainder and multiplication have the *same* precedence as each other. To work out the exact evaluation order to use, you also need to consider their associativity. Remainder and multiplication both associate with the expression to their left. Think of this as adding implicit parentheses around these parts of the expression, starting from their left:

```
2 + ((3 % 4) * 5)
```

`(3 % 4)` is 3, so this is equivalent to:

```
2 + (3 * 5)
```

`(3 * 5)` is 15, so this is equivalent to:

```
2 + 15
```

This calculation yields the final answer of 17.

For information about the operators provided by the Swift standard library, including a complete list of the operator precedence groups and associativity settings, see Operator Declarations.

> NOTE
> Swift's operator precedences and associativity rules are simpler and more predictable than those found in C and Objective-C. However, this means that they are not exactly the same as in C-based languages. Be careful to ensure that operator interactions still behave in the way you intend when porting existing code to Swift.

Operator Methods

Classes and structures can provide their own implementations of existing operators. This is known as *overloading* the existing operators.

The example below shows how to implement the arithmetic addition operator (+) for a custom structure. The arithmetic addition operator is a *binary operator* because it operates on two targets and is said to be *infix* because it appears in between those two targets.

The example defines a `Vector2D` structure for a two-dimensional position vector `(x, y)`, followed by a definition of an *operator method* to add together instances of the `Vector2D` structure:

```
struct Vector2D {
    var x = 0.0, y = 0.0
}

extension Vector2D {
    static func + (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x, y: left.y + right.y)
    }
}
```

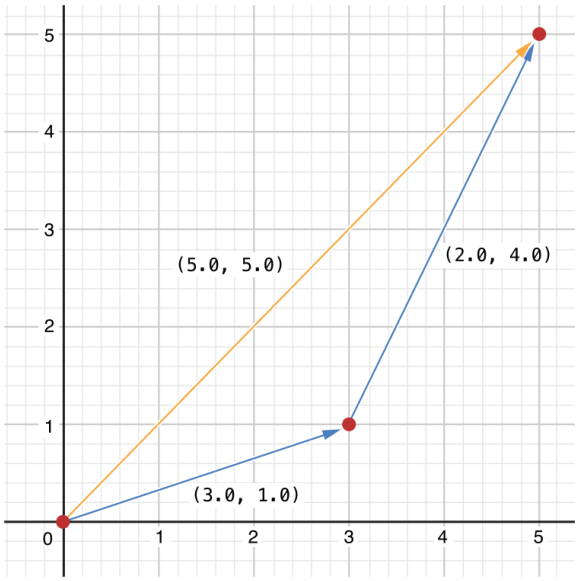
The operator method is defined as a type method on `Vector2D`, with a method name that matches the operator to be overloaded (`+`). Because addition isn't part of the essential behavior for a vector, the type method is defined in an extension of `Vector2D` rather than in the main structure declaration of `Vector2D`. Because the arithmetic addition operator is a binary operator, this operator method takes two input parameters of type `Vector2D` and returns a single output value, also of type `Vector2D`.

In this implementation, the input parameters are named `left` and `right` to represent the `Vector2D` instances that will be on the left side and right side of the `+` operator. The method returns a new `Vector2D` instance, whose `x` and `y` properties are initialized with the sum of the `x` and `y` properties from the two `Vector2D` instances that are added together.

The type method can be used as an infix operator between existing `Vector2D` instances:

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// combinedVector is a Vector2D instance with values of (5.0, 5.0)
```

This example adds together the vectors `(3.0, 1.0)` and `(2.0, 4.0)` to make the vector `(5.0, 5.0)`, as illustrated below.



Prefix and Postfix Operators

The example shown above demonstrates a custom implementation of a binary infix operator. Classes and structures can also provide implementations of the standard *unary operators*. Unary operators operate on a single target. They are *prefix* if they precede their target (such as `-a`) and *postfix* operators if they follow their target (such as `b!`).

You implement a prefix or postfix unary operator by writing the `prefix` or `postfix` modifier before the `func` keyword when declaring the operator method:

```
extension Vector2D {
    static prefix func - (vector: Vector2D) -> Vector2D {
        return Vector2D(x: -vector.x, y: -vector.y)
    }
}
```

The example above implements the unary minus operator (`-a`) for `Vector2D` instances. The unary minus operator is a prefix operator, and so this method has to be qualified with the `prefix` modifier.

For simple numeric values, the unary minus operator converts positive numbers into their negative equivalent and vice versa. The corresponding implementation for `Vector2D` instances performs this operation on both the `x` and `y` properties:

```
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// negative is a Vector2D instance with values of (-3.0, -4.0)
let alsoPositive = -negative
// alsoPositive is a Vector2D instance with values of (3.0, 4.0)
```

Compound Assignment Operators

Compound assignment operators combine assignment (=) with another operation. For example, the addition assignment operator (+=) combines addition and assignment into a single operation. You mark a compound assignment operator's left input parameter type as `inout`, because the parameter's value will be modified directly from within the operator method.

The example below implements an addition assignment operator method for `Vector2D` instances:

```
extension Vector2D {
    static func += (left: inout Vector2D, right: Vector2D) {
        left = left + right
    }
}
```

Because an addition operator was defined earlier, you don't need to reimplement the addition process here. Instead, the addition assignment operator method takes advantage of the existing addition operator method, and uses it to set the left value to be the left value plus the right value:

```
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original now has values of (4.0, 6.0)
```

> NOTE
>
> It is not possible to overload the default assignment operator (=). Only the compound assignment operators can be overloaded. Similarly, the ternary conditional operator (`a ? b : c`) cannot be overloaded.

Equivalence Operators

By default, custom classes and structures do not receive a default implementation of the *equivalence operators*, known as the "equal to" operator (==) and "not equal to" operator (`!=`).

To use the equivalence operators to check for equivalence of your own custom type, provide an implementation of the "equal to" operator in the same way as for other infix operators, and add conformance to the standard library's `Equatable` protocol:

```
extension Vector2D: Equatable {
    static func == (left: Vector2D, right: Vector2D) -> Bool {
        return (left.x == right.x) && (left.y == right.y)
    }
}
```

The above example implements an "equal to" operator (==) to check if two `Vector2D` instances have equivalent values. In the context of `Vector2D`, it makes sense to consider "equal" as meaning "both instances have the

same `x` values and `y` values", and so this is the logic used by the operator implementation. The standard library provides a default implementation of the "not equal to" operator (`!=`), which simply returns the inverse of the result of the "equal to" operator.

You can now use these operators to check whether two `Vector2D` instances are equivalent:

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    print("These two vectors are equivalent.")
}
// Prints "These two vectors are equivalent."
```

Swift provides synthesized implementations of the equivalence operators for the following kinds of custom types:

- Structures that have only stored properties that conform to the `Equatable` protocol
- Enumerations that have only associated types that conform to the `Equatable` protocol
- Enumerations that have no associated types

Declare `Equatable` conformance as part of the type's original declaration to receive these default implementations.

The example below defines a `Vector3D` structure for a three-dimensional position vector `(x, y, z)`, similar to the `Vector2D` structure. Because the `x`, `y`, and `z` properties are all of an `Equatable` type, `Vector3D` receives default implementations of the equivalence operators.

```
struct Vector3D: Equatable {
    var x = 0.0, y = 0.0, z = 0.0
}

let twoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
let anotherTwoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
if twoThreeFour == anotherTwoThreeFour {
    print("These two vectors are also equivalent.")
}
// Prints "These two vectors are also equivalent."
```

Custom Operators

You can declare and implement your own *custom operators* in addition to the standard operators provided by Swift. For a list of characters that can be used to define custom operators, see Operators.

New operators are declared at a global level using the `operator` keyword, and are marked with the `prefix`, `infix` or `postfix` modifiers:

```
prefix operator +++
```

The example above defines a new prefix operator called `+++`. This operator does not have an existing meaning in Swift, and so it is given its own custom meaning below in the specific context of working with `Vector2D` instances. For the purposes of this example, `+++` is treated as a new "prefix doubling" operator. It doubles the `x` and `y` values of a `Vector2D` instance, by adding the vector to itself with the addition assignment operator defined earlier. To implement the `+++` operator, you add a type method called `+++` to `Vector2D` as follows:

```
extension Vector2D {
    static prefix func +++ (vector: inout Vector2D) -> Vector2D {
        vector += vector
```

```
        return vector
    }
}

var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled now has values of (2.0, 8.0)
// afterDoubling also has values of (2.0, 8.0)
```

Precedence for Custom Infix Operators

Custom infix operators each belong to a precedence group. A precedence group specifies an operator's precedence relative to other infix operators, as well as the operator's associativity. See Precedence and Associativity for an explanation of how these characteristics affect an infix operator's interaction with other infix operators.

A custom infix operator that is not explicitly placed into a precedence group is given a default precedence group with a precedence immediately higher than the precedence of the ternary conditional operator.

The following example defines a new custom infix operator called `+-`, which belongs to the precedence group `AdditionPrecedence`:

```
infix operator +-: AdditionPrecedence
extension Vector2D {
    static func +- (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x, y: left.y - right.y)
    }
}
let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector +- secondVector
// plusMinusVector is a Vector2D instance with values of (4.0, -2.0)
```

This operator adds together the `x` values of two vectors, and subtracts the `y` value of the second vector from the first. Because it is in essence an "additive" operator, it has been given the same precedence group as additive infix operators such as `+` and `-`. For information about the operators provided by the Swift standard library, including a complete list of the operator precedence groups and associativity settings, see Operator Declarations. For more information about precedence groups and to see the syntax for defining your own operators and precedence groups, see Operator Declaration.

> NOTE
>
> You do not specify a precedence when defining a prefix or postfix operator. However, if you apply both a prefix and a postfix operator to the same operand, the postfix operator is applied first.