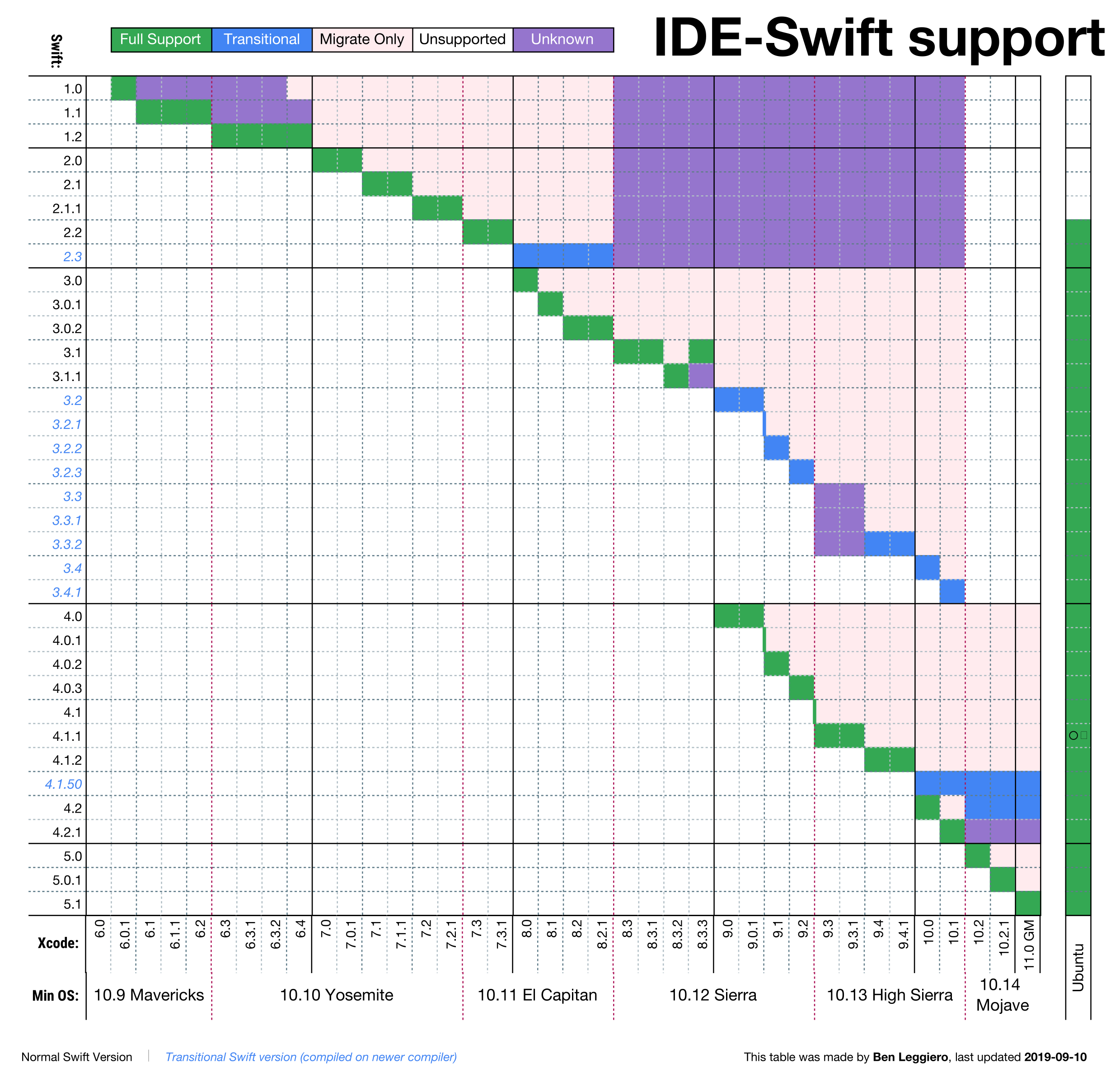
| **Version** | **Release Date** | [**macOS**](https://en.wikipedia.org/wiki/MacOS) | [**Linux**](https://en.wikipedia.org/wiki/Linux) |
| --- | --- | --- | --- |
| Swift 1.0 | September 9, 2014 | Yes | No |
| Swift 1.1 | October 22, 2014 | Yes | No |
| Swift 1.2 | April 8, 2015 | Yes | No |
| Swift 2.0 | September 21, 2015 | Yes | No |
| Swift 2.1 | October 20, 2015 | Yes | No |
| Swift 2.2 | March 21, 2016 | Yes | Yes |
| Swift 2.2.1 | May 3, 2016 | Yes | Yes |
| Swift 3.0 | September 13, 2016 | Yes | Yes |
| Swift 3.0.1 | October 28, 2016 | Yes | Yes |
| Swift 3.0.2 | December 13, 2016 | Yes | Yes |
| Swift 3.1 | March 27, 2017 | Yes | Yes |
| Swift 3.1.1 | April 21, 2017 | Yes | Yes |
| Swift 4.0 | September 19, 2017 | Yes | Yes |
| Swift 4.0.2 | November 1, 2017 | Yes | Yes |
| Swift 4.0.3 | December 5, 2017 | Yes | Yes |
| Swift 4.1 | March 29, 2018 | Yes | Yes |
| Swift 4.1.1 | May 4, 2018 | No | Yes |
| Swift 4.1.2 | May 31, 2018 | Yes | Yes |
| Swift 4.1.3 | July 27, 2018 | No | Yes |
| Swift 4.2 | September 17, 2018 | Yes | Yes |
| Swift 4.2.1 | October 30, 2018 | Yes | Yes |
| Swift 4.2.2 | February 4, 2019 | No | Yes |
| Swift 4.2.3 | February 28, 2019 | No | Yes |
| Swift 4.2.4 | March 29, 2019 | No | Yes |
| Swift 5.0[[43]](https://en.wikipedia.org/wiki/Swift_(programming_language)#cite_note-43) | March 25, 2019 | Yes | Yes |
| Swift 5.0.1 | April 18, 2019 | Yes | Yes |
| Swift 5.0.2 | July 15, 2019 | No | Yes |



**Homebrew** is a [free](https://en.wikipedia.org/wiki/Free_software) and [open-source](https://en.wikipedia.org/wiki/Open-source_software) software [package management system](https://en.wikipedia.org/wiki/Package_manager) that simplifies the installation of software on [Apple's](https://en.wikipedia.org/wiki/Apple_Inc.) [macOS](https://en.wikipedia.org/wiki/MacOS) operating system and [Linux](https://en.wikipedia.org/wiki/Linux). The name is intended to suggest the idea of building software on the Mac depending on the user's taste. Originally written by Max Howell, the package manager has gained popularity in the [Ruby on Rails](https://en.wikipedia.org/wiki/Ruby_on_Rails) community and earned praise for its [extensibility](https://en.wikipedia.org/wiki/Extensibility).[[3]](https://en.wikipedia.org/wiki/Homebrew_(package_management_software)#cite_note-3) Homebrew has been recommended for its ease of use[[4]](https://en.wikipedia.org/wiki/Homebrew_(package_management_software)#cite_note-4) as well as its integration into the [command line](https://en.wikipedia.org/wiki/Command_line_interface).[[5]](https://en.wikipedia.org/wiki/Homebrew_(package_management_software)#cite_note-5) Homebrew is a non-profit project member of the [Software Freedom Conservancy](https://en.wikipedia.org/wiki/Software_Freedom_Conservancy), and is run entirely by unpaid volunteers.[6]

Swift 5.1, the default version of Swift that’s included in Xcode 11. You can use Xcode 11 to build targets that are written in either Swift 5.1, Swift 4.2, or Swift 4.

That said, the following changes are available only to code that uses Swift 5.1 or later:

* Functions that return an opaque type require the Swift 5.1 runtime.
* The try? expression doesn’t introduce an extra level of optionality to expressions that already return optionals.
* Large integer literal initialization expressions are inferred to be of the correct integer type. For example, UInt64(0xffff\_ffff\_ffff\_ffff) evaluates to the correct value rather than overflowing.

1. var myVariable = 42
2. myVariable = 50
3. let myConstant = 42

A constant or variable must have the same type as the value you want to assign to it. However, you don’t always have to write the type explicitly. Providing a value when you create a constant or variable lets the compiler infer its type. In the example above, the compiler infers that myVariable is an integer because its initial value is an integer.

If the initial value doesn’t provide enough information (or if there is no initial value), specify the type by writing it after the variable, separated by a colon.

1. let implicitInteger = 70
2. let implicitDouble = 70.0
3. let explicitDouble: Double = 70

Values are never implicitly converted to another type. If you need to convert a value to a different type, explicitly make an instance of the desired type.

1. let label = "The width is "
2. let width = 94
3. let widthLabel = label + String(width

There’s an even simpler way to include values in strings: Write the value in parentheses, and write a backslash (\) before the parentheses. For example:

let apples = 3

let oranges = 5

let appleSummary = "I have \(apples) apples."

let fruitSummary = "I have \(apples + oranges) pieces of fruit."

Use three double quotation marks (""") for strings that take up multiple lines. Indentation at the start of each quoted line is removed, as long as it matches the indentation of the closing quotation marks. For example:

let quotation = """

I said "I have \(apples) apples."

And then I said "I have \(apples + oranges) pieces of fruit."

"""

let quotation = """

I said "I have \(apples) apples."

And then I said "I have \(apples + oranges) pieces of fruit."

"""

print(quotation)

let labeltest = UILabel(frame: CGRect(x: 0, y: 0, width: 200, height: 200))

labeltest.center = CGPoint(x: 160, y: 285)

labeltest.lineBreakMode = NSLineBreakMode.byWordWrapping

labeltest.numberOfLines = 3

labeltest.textAlignment = .center

labeltest.text = quotation

self.view.addSubview(labeltest)

Create arrays and dictionaries using brackets ([]), and access their elements by writing the index or key in brackets. A comma is allowed after the last element.

var shoppingList = ["catfish", "water", "tulips"]

shoppingList[1] = "bottle of water"

var occupations = [

"Malcolm": "Captain",

"Kaylee": "Mechanic",

]

occupations["Jayne"] = "Public Relations"

Arrays automatically grow as you add elements.

shoppingList.append("blue paint")

print(shoppingList)

To create an empty array or dictionary, use the initializer syntax.

let emptyArray = [String]()

let emptyDictionary = [String: Float]()

If type information can be inferred, you can write an empty array as [] and an empty dictionary as [:]—for example, when you set a new value for a variable or pass an argument to a function.

shoppingList = []

occupations = [:]

You can use if and let together to work with values that might be missing. These values are represented as optionals. An optional value either contains a value or contains nil to indicate that a value is missing. Write a question mark (?) after the type of a value to mark the value as optional.

var optionalString: String? = "Hello"

print(optionalString == nil)

// Prints "false"

var optionalName: String? = "John Appleseed"

var greeting = "Hello!"

if let name = optionalName {

greeting = "Hello, \(name)"

}

If the optional value is nil, the conditional is false and the code in braces is skipped.

let vegetable = "red pepper"

switch vegetable {

case "celery":

print("Add some raisins and make ants on a log.")

case "cucumber", "watercress":

print("That would make a good tea sandwich.")

case let x where x.hasSuffix("pepper"):

print("Is it a spicy \(x)?")

default:

print("Everything tastes good in soup.")

}

// Prints "Is it a spicy red pepper?"

let interestingNumbers = [

"Prime": [2, 3, 5, 7, 11, 13],

"Fibonacci": [1, 1, 2, 3, 5, 8],

"Square": [1, 4, 9, 16, 25],

]

var largest = 0

for (kind, numbers) in interestingNumbers {

for number in numbers {

if number > largest {

largest = number

}

}

}

print(largest)

// Prints "25"

Use while to repeat a block of code until a condition changes. The condition of a loop can be at the end instead, ensuring that the loop is run at least once.

var n = 2

while n < 100 {

n \*= 2

}

print(n)

// Prints "128"

var m = 2

repeat {

m \*= 2

} while m < 100

print(m)

// Prints "128"

You can keep an index in a loop by using ..< to make a range of indexes.

var total = 0

for i in 0..<4 {

total += i

}

print(total)

// Prints "6"

Use ..< to make a range that omits its upper value, and use ... to make a range that includes both values.

Use func to declare a function. Call a function by following its name with a list of arguments in parentheses. Use -> to separate the parameter names and types from the function’s return type.

func greet(person: String, day: String) -> String {

return "Hello \(person), today is \(day)."

}

greet(person: "Bob", day: "Tuesday")

By default, functions use their parameter names as labels for their arguments. Write a custom argument label before the parameter name, or write \_ to use no argument label.

func greet(\_ person: String, on day: String) -> String {

return "Hello \(person), today is \(day)."

}

greet("John", on: "Wednesday")

Use a tuple to make a compound value—for example, to return multiple values from a function. The elements of a tuple can be referred to either by name or by number.

func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {

var min = scores[0]

var max = scores[0]

var sum = 0

for score in scores {

if score > max {

max = score

} else if score < min {

min = score

}

sum += score

}

return (min, max, sum)

}

let statistics = calculateStatistics(scores: [5, 3, 100, 3, 9])

print(statistics.sum)

// Prints "120"

print(statistics.2)

// Prints "120"

Functions can be nested. Nested functions have access to variables that were declared in the outer function. You can use nested functions to organize the code in a function that is long or complex.

func returnFifteen() -> Int {

var y = 10

func add() {

y += 5

}

add()

return y

}

returnFifteen()

Functions are a first-class type. This means that a function can return another function as its value.

func makeIncrementer() -> ((Int) -> Int) {

func addOne(number: Int) -> Int {

return 1 + number

}

return addOne

}

var increment = makeIncrementer()

increment(7)

A function can take another function as one of its arguments.

func hasAnyMatches(list: [Int], condition: (Int) -> Bool) -> Bool {

for item in list {

if condition(item) {

return true

}

}

return false

}

func lessThanTen(number: Int) -> Bool {

return number < 10

}

var numbers = [20, 19, 7, 12]

hasAnyMatches(list: numbers, condition: lessThanTen)

Functions are actually a special case of closures: blocks of code that can be called later. The code in a closure has access to things like variables and functions that were available in the scope where the closure was created, even if the closure is in a different scope when it is executed—you saw an example of this already with nested functions. You can write a closure without a name by surrounding code with braces ({}). Use in to separate the arguments and return type from the body.

numbers.map({ (number: Int) -> Int in

let result = 3 \* number

return result

})

var numbers = [20, 19, 7, 12]

You have several options for writing closures more concisely. When a closure’s type is already known, such as the callback for a delegate, you can omit the type of its parameters, its return type, or both. Single statement closures implicitly return the value of their only statement.

let mappedNumbers = numbers.map({ number in 3 \* number })

print(mappedNumbers)

// Prints "[60, 57, 21, 36]"

You can refer to parameters by number instead of by name—this approach is especially useful in very short closures. A closure passed as the last argument to a function can appear immediately after the parentheses. When a closure is the only argument to a function, you can omit the parentheses entirely.

let sortedNumbers = numbers.sorted { $0 > $1 }

print(sortedNumbers)

// Prints "[20, 19, 12, 7]"

Objects and Classes

Use class followed by the class’s name to create a class. A property declaration in a class is written the same way as a constant or variable declaration, except that it is in the context of a class. Likewise, method and function declarations are written the same way.

class Shape {

var numberOfSides = 0

func simpleDescription() -> String {

return "A shape with \(numberOfSides) sides."

}

}

Create an instance of a class by putting parentheses after the class name. Use dot syntax to access the properties and methods of the instance.

var shape = Shape()

shape.numberOfSides = 7

var shapeDescription = shape.simpleDescription()

This version of the Shape class is missing something important: an initializer to set up the class when an instance is created. Use init to create one.

class NamedShape {

var numberOfSides: Int = 0

var name: String

init(name: String) {

self.name = name

}

func simpleDescription() -> String {

return "A shape with \(numberOfSides) sides."

}

}

Use deinit to create a deinitializer if you need to perform some cleanup before the object is deallocated.

Methods on a subclass that override the superclass’s implementation are marked with override—overriding a method by accident, without override, is detected by the compiler as an error. The compiler also detects methods with override that don’t actually override any method in the superclass.

class Square: NamedShape {

var sideLength: Double

init(sideLength: Double, name: String) {

self.sideLength = sideLength

super.init(name: name)

numberOfSides = 4

}

func area() -> Double {

return sideLength \* sideLength

}

override func simpleDescription() -> String {

return "A square with sides of length \(sideLength)."

}

}

let test = Square(sideLength: 5.2, name: "my test square")

test.area()

test.simpleDescription()

In addition to simple properties that are stored, properties can have a getter and a setter.

class EquilateralTriangle: NamedShape {

var sideLength: Double = 0.0

init(sideLength: Double, name: String) {

self.sideLength = sideLength

super.init(name: name)

numberOfSides = 3

}

var perimeter: Double {

get {

return 3.0 \* sideLength

}

set {

sideLength = newValue / 3.0

}

}

override func simpleDescription() -> String {

return "An equilateral triangle with sides of length \(sideLength)."

}

}

var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")

print(triangle.perimeter)

// Prints "9.3"

triangle.perimeter = 9.9

print(triangle.sideLength)

// Prints "3.3000000000000003"

If you don’t need to compute the property but still need to provide code that is run before and after setting a new value, use willSet and didSet. The code you provide is run any time the value changes outside of an initializer. For example, the class below ensures that the side length of its triangle is always the same as the side length of its square.

class TriangleAndSquare {

var triangle: EquilateralTriangle {

willSet {

square.sideLength = newValue.sideLength

}

}

var square: Square {

willSet {

triangle.sideLength = newValue.sideLength

}

}

init(size: Double, name: String) {

square = Square(sideLength: size, name: name)

triangle = EquilateralTriangle(sideLength: size, name: name)

}

}

var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shape")

print(triangleAndSquare.square.sideLength)

// Prints "10.0"

print(triangleAndSquare.triangle.sideLength)

// Prints "10.0"

triangleAndSquare.square = Square(sideLength: 50, name: "larger square")

print(triangleAndSquare.triangle.sideLength)

// Prints "50.0"

Use enum to create an enumeration. Like classes and all other named types, enumerations can have methods associated with them.

enum Rank: Int {

case ace = 1

case two, three, four, five, six, seven, eight, nine, ten

case jack, queen, king

func simpleDescription() -> String {

switch self {

case .ace:

return "ace"

case .jack:

return "jack"

case .queen:

return "queen"

case .king:

return "king"

default:

return String(self.rawValue)

}

}

}

let ace = Rank.ace

let aceRawValue = ace.rawValue

## Protocols and Extensions

Use protocol to declare a protocol.

protocol ExampleProtocol {

var simpleDescription: String { get }

mutating func adjust()

}

Classes, enumerations, and structs can all adopt protocols.

class SimpleClass: ExampleProtocol {

var simpleDescription: String = "A very simple class."

var anotherProperty: Int = 69105

func adjust() {

simpleDescription += " Now 100% adjusted."

}

}

var a = SimpleClass()

a.adjust()

let aDescription = a.simpleDescription

struct SimpleStructure: ExampleProtocol {

var simpleDescription: String = "A simple structure"

mutating func adjust() {

simpleDescription += " (adjusted)"

}

}

var b = SimpleStructure()

b.adjust()

let bDescription = b.simpleDescription

Notice the use of the mutating keyword in the declaration of SimpleStructure to mark a method that modifies the structure. The declaration of SimpleClass doesn’t need any of its methods marked as mutating because methods on a class can always modify the class.

Use extension to add functionality to an existing type, such as new methods and computed properties. You can use an extension to add protocol conformance to a type that is declared elsewhere, or even to a type that you imported from a library or framework.

extension Int: ExampleProtocol {

var simpleDescription: String {

return "The number \(self)"

}

mutating func adjust() {

self += 42

}

}

print(7.simpleDescription)

// Prints "The number 7"

Another way to handle errors is to use try? to convert the result to an optional. If the function throws an error, the specific error is discarded and the result is nil. Otherwise, the result is an optional containing the value that the function returned.

let printerSuccess = try? send(job: 1884, toPrinter: "Mergenthaler")

let printerFailure = try? send(job: 1885, toPrinter: "Never Has Toner")

Swift is a *type-safe* language, which means the language helps you to be clear about the types of values your code can work with. If part of your code requires a String, type safety prevents you from passing it an Int by mistake. Likewise, type safety prevents you from accidentally passing an optional String to a piece of code that requires a non-optional String. Type safety helps you catch and fix errors as early as possible in the development process.

The value of a *constant* can’t be changed once it’s set, whereas a *variable* can be set to a different value in the future.

You can declare multiple constants or multiple variables on a single line, separated by commas:

var x = 0.0, y = 0.0, z = 0.0

### Type Annotations

You can provide a type annotation when you declare a constant or variable, to be clear about the kind of values the constant or variable can store. Write a type annotation by placing a colon after the constant or variable name, followed by a space, followed by the name of the type to use.

This example provides a type annotation for a variable called welcomeMessage, to indicate that the variable can store String values:

var welcomeMessage: String

The colon in the declaration means “…of type…,” so the code above can be read as:

You can define multiple related variables of the same type on a single line, separated by commas, with a single type annotation after the final variable name:

var red, green, blue: Double

### Naming Constants and Variables

Constant and variable names can contain almost any character, including Unicode characters:

let π = 3.14159

let 你好 = "你好世界"

let 🐶🐮 = "dogcow"

Constant and variable names can’t contain whitespace characters, mathematical symbols, arrows, private-use Unicode scalar values, or line- and box-drawing characters. Nor can they begin with a number, although numbers may be included elsewhere within the name.

The print(\_:separator:terminator:) function is a global function that prints one or more values to an appropriate output. In Xcode, for example, the print(\_:separator:terminator:) function prints its output in Xcode’s “console” pane. The separator and terminator parameter have default values, so you can omit them when you call this function. By default, the function terminates the line it prints by adding a line break. To print a value without a line break after it, pass an empty string as the terminator—for example, print(someValue, terminator: "").

Swift uses *string interpolation* to include the name of a constant or variable as a placeholder in a longer string, and to prompt Swift to replace it with the current value of that constant or variable. Wrap the name in parentheses and escape it with a backslash before the opening parenthesis:

var friendlyWelcome = "Hello!"

friendlyWelcome = "Bonjour!"

print("The current value of friendlyWelcome is \(friendlyWelcome)")

// Prints "The current value of friendlyWelcome is Bonjour!"

## Comments

Use comments to include nonexecutable text in your code, as a note or reminder to yourself. Comments are ignored by the Swift compiler when your code is compiled.

Comments in Swift are very similar to comments in C. Single-line comments begin with two forward-slashes (//):

// This is a comment.

Multiline comments start with a forward-slash followed by an asterisk (/\*) and end with an asterisk followed by a forward-slash (\*/):

/\* This is also a comment

but is written over multiple lines. \*/

Unlike multiline comments in C, multiline comments in Swift can be nested inside other multiline comments. You write nested comments by starting a multiline comment block and then starting a second multiline comment within the first block. The second block is then closed, followed by the first block:

/\* This is the start of the first multiline comment.

/\* This is the second, nested multiline comment. \*/

This is the end of the first multiline comment. \*/

## Semicolons

Unlike many other languages, Swift doesn’t require you to write a semicolon (;) after each statement in your code, although you can do so if you wish. However, semicolons are required if you want to write multiple separate statements on a single line:

let cat = "🐱"; print(cat)

// Prints "🐱"

### Integer Bounds

You can access the minimum and maximum values of each integer type with its min and max properties:

let minValue = UInt8.min // minValue is equal to 0, and is of type UInt8

let maxValue = UInt8.max // maxValue is equal to 255, and is of type UInt8

The values of these properties are of the appropriate-sized number type (such as UInt8 in the example above) and can therefore be used in expressions alongside other values of the same type.

### Int

In most cases, you don’t need to pick a specific size of integer to use in your code. Swift provides an additional integer type, Int, which has the same size as the current platform’s native word size:

* On a 32-bit platform, Int is the same size as Int32.
* On a 64-bit platform, Int is the same size as Int64.

### UInt

Swift also provides an unsigned integer type, UInt, which has the same size as the current platform’s native word size:

* On a 32-bit platform, UInt is the same size as UInt32.
* On a 64-bit platform, UInt is the same size as UInt64.

## Floating-Point Numbers

Floating-point numbers are numbers with a fractional component, such as 3.14159, 0.1, and -273.15.

Floating-point types can represent a much wider range of values than integer types, and can store numbers that are much larger or smaller than can be stored in an Int. Swift provides two signed floating-point number types:

* Double represents a 64-bit floating-point number.
* Float represents a 32-bit floating-point number.

Because Swift is type safe, it performs *type checks* when compiling your code and flags any mismatched types as errors. This enables you to catch and fix errors as early as possible in the development process.

Swift uses *type inference* to work out the appropriate type. Type inference enables a compiler to deduce the type of a particular expression automatically when it compiles your code, simply by examining the values you provide.

Likewise, if you don’t specify a type for a floating-point literal, Swift infers that you want to create a Double:

let pi = 3.14159

// pi is inferred to be of type Double

Swift always chooses Double (rather than Float) when inferring the type of floating-point numbers.

If you combine integer and floating-point literals in an expression, a type of Double will be inferred from the context:

let anotherPi = 3 + 0.14159

// anotherPi is also inferred to be of type Double

## Numeric Literals

Integer literals can be written as:

* A decimal number, with no prefix
* A binary number, with a 0b prefix
* An octal number, with a 0o prefix
* A hexadecimal number, with a 0x prefix

All of these integer literals have a decimal value of 17:

let decimalInteger = 17

let binaryInteger = 0b10001 // 17 in binary notation

let octalInteger = 0o21 // 17 in octal notation

let hexadecimalInteger = 0x11 // 17 in hexadecimal notation

Floating-point literals can be decimal (with no prefix), or hexadecimal (with a 0x prefix). They must always have a number (or hexadecimal number) on both sides of the decimal point. Decimal floats can also have an optional *exponent*, indicated by an uppercase or lowercase e; hexadecimal floats must have an exponent, indicated by an uppercase or lowercase p.

For decimal numbers with an exponent of exp, the base number is multiplied by 10exp:

* 1.25e2 means 1.25 x 102, or 125.0.
* 1.25e-2 means 1.25 x 10-2, or 0.0125.

For hexadecimal numbers with an exponent of exp, the base number is multiplied by 2exp:

* 0xFp2 means 15 x 22, or 60.0.
* 0xFp-2 means 15 x 2-2, or 3.75.

All of these floating-point literals have a decimal value of 12.1875:

let decimalDouble = 12.1875

let exponentDouble = 1.21875e1

let hexadecimalDouble = 0xC.3p0

 number that won’t fit into a constant or variable of a sized integer type is reported as an error when your code is compiled:

let cannotBeNegative: UInt8 = -1

// UInt8 cannot store negative numbers, and so this will report an error

let tooBig: Int8 = Int8.max + 1

// Int8 cannot store a number larger than its maximum value,

// and so this will also report an error

To convert one specific number type to another, you initialize a new number of the desired type with the existing value. In the example below, the constant twoThousand is of type UInt16, whereas the constant one is of type UInt8. They can’t be added together directly, because they’re not of the same type. Instead, this example calls UInt16(one) to create a new UInt16 initialized with the value of one, and uses this value in place of the original:

let twoThousand: UInt16 = 2\_000

let one: UInt8 = 1

let twoThousandAndOne = twoThousand + UInt16(one)

Because both sides of the addition are now of type UInt16, the addition is allowed. The output constant (twoThousandAndOne) is inferred to be of type UInt16, because it’s the sum of two UInt16 values.

### Integer and Floating-Point Conversion

Conversions between integer and floating-point numeric types must be made explicit:

let three = 3

let pointOneFourOneFiveNine = 0.14159

let pi = Double(three) + pointOneFourOneFiveNine

// pi equals 3.14159, and is inferred to be of type Double

Floating-point to integer conversion must also be made explicit. An integer type can be initialized with a Double or Float value:

1. let integerPi = Int(pi)
2. // integerPi equals 3, and is inferred to be of type Int

Floating-point values are always truncated when used to initialize a new integer value in this way. This means that 4.75 becomes 4, and -3.9 becomes -3.

## Type Aliases

Type aliases define an alternative name for an existing type. You define type aliases with the typealias keyword.

Type aliases are useful when you want to refer to an existing type by a name that is contextually more appropriate, such as when working with data of a specific size from an external source:

typealias AudioSample = UInt16

Once you define a type alias, you can use the alias anywhere you might use the original name:

var maxAmplitudeFound = AudioSample.min

// maxAmplitudeFound is now 0

As with Int and Double above, you don’t need to declare constants or variables as Bool if you set them to true or false as soon as you create them.

## Tuples

Tuples group multiple values into a single compound value. The values within a tuple can be of any type and don’t have to be of the same type as each other.

In this example, (404, "Not Found") is a tuple that describes an HTTP status code. An HTTP status code is a special value returned by a web server whenever you request a web page. A status code of 404 Not Found is returned if you request a webpage that doesn’t exist.

let http404Error = (404, "Not Found")

// http404Error is of type (Int, String), and equals (404, "Not Found")

The (404, "Not Found") tuple groups together an Int and a String to give the HTTP status code two separate values: a number and a human-readable description. It can be described as “a tuple of type (Int, String)”.

You can *decompose* a tuple’s contents into separate constants or variables, which you then access as usual:

let (statusCode, statusMessage) = http404Error

print("The status code is \(statusCode)")

// Prints "The status code is 404"

print("The status message is \(statusMessage)")

// Prints "The status message is Not Found"

If you only need some of the tuple’s values, ignore parts of the tuple with an underscore (\_) when you decompose the tuple:

let (justTheStatusCode, \_) = http404Error

print("The status code is \(justTheStatusCode)")

// Prints "The status code is 404"

Alternatively, access the individual element values in a tuple using index numbers starting at zero:

print("The status code is \(http404Error.0)")

// Prints "The status code is 404"

print("The status message is \(http404Error.1)")

// Prints "The status message is Not Found"

## Optionals

You use optionals in situations where a value may be absent. An optional represents two possibilities: Either there is a value, and you can unwrap the optional to access that value, or there isn’t a value at all.

NOTE

The concept of optionals doesn’t exist in C or Objective-C. The nearest thing in Objective-C is the ability to return nil from a method that would otherwise return an object, with nil meaning “the absence of a valid object.” However, this only works for objects—it doesn’t work for structures, basic C types, or enumeration values. For these types, Objective-C methods typically return a special value (such as NSNotFound) to indicate the absence of a value. This approach assumes that the method’s caller knows there’s a special value to test against and remembers to check for it. Swift’s optionals let you indicate the absence of a value for any type at all, without the need for special constants.

### nil

You set an optional variable to a valueless state by assigning it the special value nil:

var serverResponseCode: Int? = 404

// serverResponseCode contains an actual Int value of 404

serverResponseCode = nil

// serverResponseCode now contains no value

NOTE

You can’t use nil with non-optional constants and variables. If a constant or variable in your code needs to work with the absence of a value under certain conditions, always declare it as an optional value of the appropriate type.

If you define an optional variable without providing a default value, the variable is automatically set to nil for you:

var surveyAnswer: String?

// surveyAnswer is automatically set to nil

NOTE

Swift’s nil isn’t the same as nil in Objective-C. In Objective-C, nil is a pointer to a nonexistent object. In Swift, nil isn’t a pointer—it’s the absence of a value of a certain type. Optionals of *any* type can be set to nil, not just object types.

### If Statements and Forced Unwrapping

You can use an if statement to find out whether an optional contains a value by comparing the optional against nil. You perform this comparison with the “equal to” operator (==) or the “not equal to” operator (!=).

If an optional has a value, it’s considered to be “not equal to” nil:

if convertedNumber != nil {

print("convertedNumber contains some integer value.")

}

// Prints "convertedNumber contains some integer value."

Once you’re sure that the optional *does* contain a value, you can access its underlying value by adding an exclamation mark (!) to the end of the optional’s name. The exclamation mark effectively says, “I know that this optional definitely has a value; please use it.” This is known as *forced unwrapping* of the optional’s value:

if convertedNumber != nil {

print("convertedNumber has an integer value of \(convertedNumber!).")

}

// Prints "convertedNumber has an integer value of 123.

NOTE

Trying to use ! to access a nonexistent optional value triggers a runtime error. Always make sure that an optional contains a non-nil value before using ! to force-unwrap its value.

You use *optional binding* to find out whether an optional contains a value, and if so, to make that value available as a temporary constant or variable. Optional binding can be used with if and while statements to check for a value inside an optional, and to extract that value into a constant or variable, as part of a single action.

Write an optional binding for an if statement as follows:

if let constantName = someOptional {

statements

}

You can include as many optional bindings and Boolean conditions in a single if statement as you need to, separated by commas. If any of the values in the optional bindings are nil or any Boolean condition evaluates to false, the whole if statement’s condition is considered to be false. The following if statements are equivalent:

if let firstNumber = Int("4"), let secondNumber = Int("42"), firstNumber < secondNumber && secondNumber < 100 {

print("\(firstNumber) < \(secondNumber) < 100")

}

// Prints "4 < 42 < 100"

if let firstNumber = Int("4") {

if let secondNumber = Int("42") {

if firstNumber < secondNumber && secondNumber < 100 {

print("\(firstNumber) < \(secondNumber) < 100")

}

}

}

// Prints "4 < 42 < 100"

Constants and variables created with optional binding in an if statement are available only within the body of the if statement. In contrast, the constants and variables created with a guard statement are available in the lines of code that follow the guard statement, as described in [Early Exit](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID525).

Sometimes it’s clear from a program’s structure that an optional will *always* have a value, after that value is first set. In these cases, it’s useful to remove the need to check and unwrap the optional’s value every time it’s accessed, because it can be safely assumed to have a value all of the time.

These kinds of optionals are defined as *implicitly unwrapped optionals*. You write an implicitly unwrapped optional by placing an exclamation mark (String!) rather than a question mark (String?) after the type that you want to make optional.

Implicitly unwrapped optionals are useful when an optional’s value is confirmed to exist immediately after the optional is first defined and can definitely be assumed to exist at every point thereafter. The primary use of implicitly unwrapped optionals in Swift is during class initialization, as described in [Unowned References and Implicitly Unwrapped Optional Properties](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID55).

An implicitly unwrapped optional is a normal optional behind the scenes, but can also be used like a non-optional value, without the need to unwrap the optional value each time it’s accessed. The following example shows the difference in behavior between an optional string and an implicitly unwrapped optional string when accessing their wrapped value as an explicit String:

1. let possibleString: String? = "An optional string."
2. let forcedString: String = possibleString! // requires an exclamation mark
3. let assumedString: String! = "An implicitly unwrapped optional string."
4. let implicitString: String = assumedString // no need for an exclamation mark

You can think of an implicitly unwrapped optional as giving permission for the optional to be unwrapped automatically whenever it’s used. Rather than placing an exclamation mark after the optional’s name each time you use it, you place an exclamation mark after the optional’s type when you declare it.

NOTE

If an implicitly unwrapped optional is nil and you try to access its wrapped value, you’ll trigger a runtime error. The result is exactly the same as if you place an exclamation mark after a normal optional that doesn’t contain a value.

You can still treat an implicitly unwrapped optional like a normal optional, to check if it contains a value:

1. if assumedString != nil {
2. print(assumedString!)
3. }
4. // Prints "An implicitly unwrapped optional string."

You can also use an implicitly unwrapped optional with optional binding, to check and unwrap its value in a single statement:

1. if let definiteString = assumedString {
2. print(definiteString)
3. }
4. // Prints "An implicitly unwrapped optional string."

NOTE

Don’t use an implicitly unwrapped optional when there’s a possibility of a variable becoming nil at a later point. Always use a normal optional type if you need to check for a nil value during the lifetime of a variable.

Swift automatically propagates errors out of their current scope until they’re handled by a catch clause.

do {

try canThrowAnError()

// no error was thrown

} catch {

// an error was thrown

}

## Assertions and Preconditions

Assertions and preconditions are checks that happen at runtime. You use them to make sure an essential condition is satisfied before executing any further code. If the Boolean condition in the assertion or precondition evaluates to true, code execution continues as usual. If the condition evaluates to false, the current state of the program is invalid; code execution ends, and your app is terminated.

You use assertions and preconditions to express the assumptions you make and the expectations you have while coding, so you can include them as part of your code. Assertions help you find mistakes and incorrect assumptions during development, and preconditions help you detect issues in production.

In addition to verifying your expectations at runtime, assertions and preconditions also become a useful form of documentation within the code. Unlike the error conditions discussed in [Error Handling](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID515) above, assertions and preconditions aren’t used for recoverable or expected errors. Because a failed assertion or precondition indicates an invalid program state, there’s no way to catch a failed assertion.

Using assertions and preconditions isn’t a substitute for designing your code in such a way that invalid conditions are unlikely to arise. However, using them to enforce valid data and state causes your app to terminate more predictably if an invalid state occurs, and helps make the problem easier to debug. Stopping execution as soon as an invalid state is detected also helps limit the damage caused by that invalid state.

The difference between assertions and preconditions is in when they’re checked: Assertions are checked only in debug builds, but preconditions are checked in both debug and production builds. In production builds, the condition inside an assertion isn’t evaluated. This means you can use as many assertions as you want during your development process, without impacting performance in production.

### Debugging with Assertions

You write an assertion by calling the [assert(\_:\_:file:line:)](https://developer.apple.com/documentation/swift/1541112-assert) function from the Swift standard library. You pass this function an expression that evaluates to true or false and a message to display if the result of the condition is false. For example:

let age = -3

assert(age >= 0, "A person's age can't be less than zero.")

// This assertion fails because -3 is not >= 0.

In this example, code execution continues if age >= 0 evaluates to true, that is, if the value of age is nonnegative. If the value of age is negative, as in the code above, then age >= 0 evaluates to false, and the assertion fails, terminating the application.

You can omit the assertion message—for example, when it would just repeat the condition as prose.

assert(age >= 0)

If the code already checks the condition, you use the [assertionFailure(\_:file:line:)](https://developer.apple.com/documentation/swift/1539616-assertionfailure) function to indicate that an assertion has failed. For example:

if age > 10 {

print("You can ride the roller-coaster or the ferris wheel.")

} else if age >= 0 {

print("You can ride the ferris wheel.")

} else {

assertionFailure("A person's age can't be less than zero.")

}

### Enforcing Preconditions

Use a precondition whenever a condition has the potential to be false, but must definitely be true for your code to continue execution. For example, use a precondition to check that a subscript is not out of bounds, or to check that a function has been passed a valid value.

You write a precondition by calling the [precondition(\_:\_:file:line:)](https://developer.apple.com/documentation/swift/1540960-precondition) function. You pass this function an expression that evaluates to true or false and a message to display if the result of the condition is false. For example:

1. // In the implementation of a subscript...
2. precondition(index > 0, "Index must be greater than zero.")

You can also call the [preconditionFailure(\_:file:line:)](https://developer.apple.com/documentation/swift/1539374-preconditionfailure) function to indicate that a failure has occurred—for example, if the default case of a switch was taken, but all valid input data should have been handled by one of the switch’s other cases.

NOTE

If you compile in unchecked mode (-Ounchecked), preconditions aren’t checked. The compiler assumes that preconditions are always true, and it optimizes your code accordingly. However, the fatalError(\_:file:line:) function always halts execution, regardless of optimization settings.

You can use the fatalError(\_:file:line:) function during prototyping and early development to create stubs for functionality that hasn’t been implemented yet, by writing fatalError("Unimplemented") as the stub implementation. Because fatal errors are never optimized out, unlike assertions or preconditions, you can be sure that execution always halts if it encounters a stub implementation.

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:

## Arithmetic Operators

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

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

### 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).

-9 % 4 // equals -1

### 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"

### 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

## 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)

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

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"

if ("aplle" < "zebra") {

print("aplle is somparable")

}

if ("aplle" < "aplle") {

print("aplle is somparable")

}

if((1, "zebra") < (2, "zebra")){

print(" true because 1 is less than 2; \"zebra\" and \"apple\" are not compared")

}else {

print(" true because 1 is less than 2; \"zebra\" and \"apple\" are not compared")

}

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.

let contentHeight = 40

let hasHeader = true

let rowHeight = contentHeight + (hasHeader ? 50 : 20)

// rowHeight is equal to 90

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

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

### 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

### 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

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)

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.

NOTE

Swift’s String type is bridged with Foundation’s NSString class. Foundation also extends String to expose methods defined by NSString. This means, if you import Foundation, you can access those NSString methods on String without casting.

### Multiline String Literals

If you need a string that spans several lines, use a multiline string literal—a sequence of characters surrounded by three double quotation marks:

let quotation = """

The White Rabbit put on his spectacles. "Where shall I begin,

please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on

till you come to the end; then stop."

"""

When your source code includes a line break inside of a multiline string literal, that line break also appears in the string’s value. If you want to use line breaks to make your source code easier to read, but you don’t want the line breaks to be part of the string’s value, write a backslash (\) at the end of those lines:

let softWrappedQuotation = """

The White Rabbit put on his spectacles. "Where shall I begin, \

please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on \

till you come to the end; then stop."

"""

To make a multiline string literal that begins or ends with a line feed, write a blank line as the first or last line. For example:

let lineBreaks = """

This string starts with a line break.

It also ends with a line break.

"""

A multiline string can be indented to match the surrounding code. The whitespace before the closing quotation marks (""") tells Swift what whitespace to ignore before all of the other lines. However, if you write whitespace at the beginning of a line in addition to what’s before the closing quotation marks, that whitespace *is* included.

let linesWithIndentation = """

This line doesn't begin with whitespace.

This line begins with four spaces.

This line doesn't begin with whitespace.

"""

In the example above, even though the entire multiline string literal is indented, the first and last lines in the string don’t begin with any whitespace. The middle line has more indentation than the closing quotation marks, so it starts with that extra four-space indentation.

### Special Characters in String Literals

String literals can include the following special characters:

* The escaped special characters \0 (null character), \\ (backslash), \t (horizontal tab), \n (line feed), \r (carriage return), \" (double quotation mark) and \' (single quotation mark)
* An arbitrary Unicode scalar value, written as \u{n}, where n is a 1–8 digit hexadecimal number (Unicode is discussed in [Unicode](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID293) below)

The code below shows four examples of these special characters. The wiseWords constant contains two escaped double quotation marks. The dollarSign, blackHeart, and sparklingHeart constants demonstrate the Unicode scalar format:

let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"

// "Imagination is more important than knowledge" - Einstein

let dollarSign = "\u{24}" // $, Unicode scalar U+0024

let blackHeart = "\u{2665}" // ♥, Unicode scalar U+2665

let sparklingHeart = "\u{1F496}" // 💖, Unicode scalar U+1F496

Because multiline string literals use three double quotation marks instead of just one, you can include a double quotation mark (") inside of a multiline string literal without escaping it. To include the text """ in a multiline string, escape at least one of the quotation marks. For example:

let threeDoubleQuotationMarks = """

Escaping the first quotation mark \"""

Escaping all three quotation marks \"\"\"

"""

### Extended String Delimiters

You can place a string literal within extended delimiters to include special characters in a string without invoking their effect. You place your string within quotation marks (") and surround that with number signs (#). For example, printing the string literal #"Line 1\nLine 2"# prints the line feed escape sequence (\n) rather than printing the string across two lines.

If you need the special effects of a character in a string literal, match the number of number signs within the string following the escape character (\). For example, if your string is #"Line 1\nLine 2"# and you want to break the line, you can use #"Line 1\#nLine 2"# instead. Similarly, ###"Line1\###nLine2"### also breaks the line.

String literals created using extended delimiters can also be multiline string literals. You can use extended delimiters to include the text """ in a multiline string, overriding the default behavior that ends the literal. For example:

let threeMoreDoubleQuotationMarks = #"""

Here are three more double quotes: """

"""#

Initializing an Empty String

To create an empty String value as the starting point for building a longer string, either assign an empty string literal to a variable, or initialize a new String instance with initializer syntax:

var emptyString = "" // empty string literal

var anotherEmptyString = String() // initializer syntax

// these two strings are both empty, and are equivalent to each other

Find out whether a String value is empty by checking its Boolean isEmpty property:

if emptyString.isEmpty {

print("Nothing to see here")

}

// Prints "Nothing to see here"

var variableString = "Horse"

variableString += " and carriage"

// variableString is now "Horse and carriage"

let constantString = "Highlander"

constantString += " and another Highlander"

// this reports a compile-time error - a constant string cannot be modified

Swift’s String type is a *value type*. If you create a new String value, that String value is *copied* when it’s passed to a function or method, or when it’s assigned to a constant or variable. In each case, a new copy of the existing String value is created, and the new copy is passed or assigned, not the original version.

Swift’s copy-by-default String behavior ensures that when a function or method passes you a String value, it’s clear that you own that exact String value, regardless of where it came from. You can be confident that the string you are passed won’t be modified unless you modify it yourself.

Working with Characters

You can access the individual Character values for a String by iterating over the string with a for-in loop:

for character in "Dog!🐶" {

print(character)

}

// D

// o

// g

// !

// 🐶

Alternatively, you can create a stand-alone Character constant or variable from a single-character string literal by providing a Character type annotation:

let exclamationMark: Character = "!"

String values can be constructed by passing an array of Character values as an argument to its initializer:

let catCharacters: [Character] = ["C", "a", "t", "!", "🐱"]

let catString = String(catCharacters)

print(catString)

// Prints "Cat!🐱"

String values can be added together (or *concatenated*) with the addition operator (+) to create a new String value:

let string1 = "hello"

let string2 = " there"

var welcome = string1 + string2

// welcome now equals "hello there"

You can also append a String value to an existing String variable with the addition assignment operator (+=):

var instruction = "look over"

instruction += string2

// instruction now equals "look over there"

You can append a Character value to a String variable with the String type’s append() method:

let exclamationMark: Character = "!"

welcome.append(exclamationMark)

// welcome now equals "hello there!"

If you’re using multiline string literals to build up the lines of a longer string, you want every line in the string to end with a line break, including the last line. For example:

let badStart = """

one

two

"""

let end = """

three

"""

print(badStart + end)

// Prints two lines:

// one

// twothree

let goodStart = """

one

two

"""

print(goodStart + end)

// Prints three lines:

// one

// two

// three

In the code above, concatenating badStart with end produces a two-line string, which isn’t the desired result. Because the last line of badStart doesn’t end with a line break, that line gets combined with the first line of end. In contrast, both lines of goodStart end with a line break, so when it’s combined with end the result has three lines, as expected.

String Interpolation

*String interpolation* is a way to construct a new String value from a mix of constants, variables, literals, and expressions by including their values inside a string literal. You can use string interpolation in both single-line and multiline string literals. Each item that you insert into the string literal is wrapped in a pair of parentheses, prefixed by a backslash (\):

let multiplier = 3

let message = "\(multiplier) times 2.5 is \(Double(multiplier) \* 2.5)"

// message is "3 times 2.5 is 7.5"

You can use extended string delimiters to create strings containing characters that would otherwise be treated as a string interpolation. For example:

print(#"Write an interpolated string in Swift using \(multiplier)."#)

// Prints "Write an interpolated string in Swift using \(multiplier)."

To use string interpolation inside a string that uses extended delimiters, match the number of number signs before the backslash to the number of number signs at the beginning and end of the string. For example:

print(#"6 times 7 is \#(6 \* 7)."#)

// Prints "6 times 7 is 42."

NOTE

The expressions you write inside parentheses within an interpolated string can’t contain an unescaped backslash (\), a carriage return, or a line feed. However, they can contain other string literals.

## Unicode

Unicode is an international standard for encoding, representing, and processing text in different writing systems. It enables you to represent almost any character from any language in a standardized form, and to read and write those characters to and from an external source such as a text file or web page. Swift’s String and Character types are fully Unicode-compliant, as described in this section.

### Unicode Scalar Values

Behind the scenes, Swift’s native String type is built from Unicode scalar values. A Unicode scalar value is a unique 21-bit number for a character or modifier, such as U+0061 for LATIN SMALL LETTER A ("a"), or U+1F425 for FRONT-FACING BABY CHICK ("🐥").

Note that not all 21-bit Unicode scalar values are assigned to a character—some scalars are reserved for future assignment or for use in UTF-16 encoding. Scalar values that have been assigned to a character typically also have a name, such as LATIN SMALL LETTER A and FRONT-FACING BABY CHICK in the examples above.

### Extended Grapheme Clusters

Every instance of Swift’s Character type represents a single extended grapheme cluster. An extended grapheme cluster is a sequence of one or more Unicode scalars that (when combined) produce a single human-readable character.

Here’s an example. The letter é can be represented as the single Unicode scalar é (LATIN SMALL LETTER E WITH ACUTE, or U+00E9). However, the same letter can also be represented as a pair of scalars—a standard letter e (LATIN SMALL LETTER E, or U+0065), followed by the COMBINING ACUTE ACCENT scalar (U+0301). The COMBINING ACUTE ACCENT scalar is graphically applied to the scalar that precedes it, turning an e into an é when it’s rendered by a Unicode-aware text-rendering system.

In both cases, the letter é is represented as a single Swift Character value that represents an extended grapheme cluster. In the first case, the cluster contains a single scalar; in the second case, it’s a cluster of two scalars:

1. let eAcute: Character = "\u{E9}" // é
2. let combinedEAcute: Character = "\u{65}\u{301}" // e followed by ́
3. // eAcute is é, combinedEAcute is é

Extended grapheme clusters are a flexible way to represent many complex script characters as a single Character value. For example, Hangul syllables from the Korean alphabet can be represented as either a precomposed or decomposed sequence. Both of these representations qualify as a single Character value in Swift:

let precomposed: Character = "\u{D55C}" // 한

let decomposed: Character = "\u{1112}\u{1161}\u{11AB}" // ᄒ, ᅡ, ᆫ

// precomposed is 한, decomposed is 한

Extended grapheme clusters enable scalars for enclosing marks (such as COMBINING ENCLOSING CIRCLE, or U+20DD) to enclose other Unicode scalars as part of a single Character value:

let enclosedEAcute: Character = "\u{E9}\u{20DD}"

// enclosedEAcute is é⃝

Unicode scalars for regional indicator symbols can be combined in pairs to make a single Character value, such as this combination of REGIONAL INDICATOR SYMBOL LETTER U (U+1F1FA) and REGIONAL INDICATOR SYMBOL LETTER S (U+1F1F8):

let regionalIndicatorForUS: Character = "\u{1F1FA}\u{1F1F8}"

// regionalIndicatorForUS is 🇺🇸

To retrieve a count of the Character values in a string, use the count property of the string:

1. let unusualMenagerie = "Koala 🐨, Snail 🐌, Penguin 🐧, Dromedary 🐪"
2. print("unusualMenagerie has \(unusualMenagerie.count) characters")
3. // Prints "unusualMenagerie has 40 characters"

Note that Swift’s use of extended grapheme clusters for Character values means that string concatenation and modification may not always affect a string’s character count.

For example, if you initialize a new string with the four-character word cafe, and then append a COMBINING ACUTE ACCENT (U+0301) to the end of the string, the resulting string will still have a character count of 4, with a fourth character of é, not e:

var word = "cafe"

print("the number of characters in \(word) is \(word.count)")

// Prints "the number of characters in cafe is 4"

word += "\u{301}" // COMBINING ACUTE ACCENT, U+0301

print("the number of characters in \(word) is \(word.count)")

// Prints "the number of characters in café is 4"

NOTE

Extended grapheme clusters can be composed of multiple Unicode scalars. This means that different characters—and different representations of the same character—can require different amounts of memory to store. Because of this, characters in Swift don’t each take up the same amount of memory within a string’s representation. As a result, the number of characters in a string can’t be calculated without iterating through the string to determine its extended grapheme cluster boundaries. If you are working with particularly long string values, be aware that the count property must iterate over the Unicode scalars in the entire string in order to determine the characters for that string.

The count of the characters returned by the count property isn’t always the same as the length property of an NSString that contains the same characters. The length of an NSString is based on the number of 16-bit code units within the string’s UTF-16 representation and not the number of Unicode extended grapheme clusters within the string.

### String Indices

Each String value has an associated index type, String.Index, which corresponds to the position of each Character in the string.

As mentioned above, different characters can require different amounts of memory to store, so in order to determine which Character is at a particular position, you must iterate over each Unicode scalar from the start or end of that String. For this reason, Swift strings can’t be indexed by integer values.

Use the startIndex property to access the position of the first Character of a String. The endIndex property is the position after the last character in a String. As a result, the endIndex property isn’t a valid argument to a string’s subscript. If a String is empty, startIndex and endIndex are equal.

You access the indices before and after a given index using the index(before:) and index(after:) methods of String. To access an index farther away from the given index, you can use the index(\_:offsetBy:) method instead of calling one of these methods multiple times.

You can use subscript syntax to access the Character at a particular String index.

let greeting = "Guten Tag!"

greeting[greeting.startIndex]

// G

greeting[greeting.index(before: greeting.endIndex)]

// !

greeting[greeting.index(after: greeting.startIndex)]

// u

let index = greeting.index(greeting.startIndex, offsetBy: 7)

greeting[index]

// a

Attempting to access an index outside of a string’s range or a Character at an index outside of a string’s range will trigger a runtime error.

greeting[greeting.endIndex] // Error

greeting.index(after: greeting.endIndex) // Error

Use the indices property to access all of the indices of individual characters in a string.

for index in greeting.indices {

print("\(greeting[index]) ", terminator: "")

}

// Prints "G u t e n T a g ! "

NOTE

You can use the startIndex and endIndex properties and the index(before:), index(after:), and index(\_:offsetBy:) methods on any type that conforms to the Collection protocol. This includes String, as shown here, as well as collection types such as Array, Dictionary, and Set.

### Inserting and Removing

To insert a single character into a string at a specified index, use the insert(\_:at:) method, and to insert the contents of another string at a specified index, use the insert(contentsOf:at:) method.

1. var welcome = "hello"
2. welcome.insert("!", at: welcome.endIndex)
3. // welcome now equals "hello!"
4. welcome.insert(contentsOf: " there", at: welcome.index(before: welcome.endIndex))
5. // welcome now equals "hello there!"

To remove a single character from a string at a specified index, use the remove(at:) method, and to remove a substring at a specified range, use the removeSubrange(\_:) method:

1. welcome.remove(at: welcome.index(before: welcome.endIndex))
2. // welcome now equals "hello there"
3. let range = welcome.index(welcome.endIndex, offsetBy: -6)..<welcome.endIndex
4. welcome.removeSubrange(range)
5. // welcome now equals "hello"

NOTE

You can use the insert(\_:at:), insert(contentsOf:at:), remove(at:), and removeSubrange(\_:) methods on any type that conforms to the RangeReplaceableCollection protocol. This includes String, as shown here, as well as collection types such as Array, Dictionary, and Set.

 you convert the substring to an instance of String. For example:

1. let greeting = "Hello, world!"
2. let index = greeting.firstIndex(of: ",") ?? greeting.endIndex
3. let beginning = greeting[..<index]
4. // beginning is "Hello"
5. // Convert the result to a String for long-term storage.
6. let newString = String(beginning)

Like strings, each substring has a region of memory where the characters that make up the substring are stored. The difference between strings and substrings is that, as a performance optimization, a substring can reuse part of the memory that’s used to store the original string, or part of the memory that’s used to store another substring. (Strings have a similar optimization, but if two strings share memory, they are equal.) This performance optimization means you don’t have to pay the performance cost of copying memory until you modify either the string or substring. As mentioned above, substrings aren’t suitable for long-term storage—because they reuse the storage of the original string, the entire original string must be kept in memory as long as any of its substrings are being used.

In the example above, greeting is a string, which means it has a region of memory where the characters that make up the string are stored. Because beginning is a substring of greeting, it reuses the memory that greeting uses. In contrast, newString is a string—when it’s created from the substring, it has its own storage.

Both String and Substring conform to the [StringProtocol](https://developer.apple.com/documentation/swift/stringprotocol) protocol, which means it’s often convenient for string-manipulation functions to accept a StringProtocol value. You can call such functions with either a String or Substring value.

## Comparing Strings

Swift provides three ways to compare textual values: string and character equality, prefix equality, and suffix equality.

String and character equality is checked with the “equal to” operator (==) and the “not equal to” operator (!=), as described in [Comparison Operators](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID70):

let quotation = "We're a lot alike, you and I."

let sameQuotation = "We're a lot alike, you and I."

if quotation == sameQuotation {

print("These two strings are considered equal")

}

// Prints "These two strings are considered equal"

Two String values (or two Character values) are considered equal if their extended grapheme clusters are *canonically equivalent*. Extended grapheme clusters are canonically equivalent if they have the same linguistic meaning and appearance, even if they’re composed from different Unicode scalars behind the scenes.

For example, LATIN SMALL LETTER E WITH ACUTE (U+00E9) is canonically equivalent to LATIN SMALL LETTER E (U+0065) followed by COMBINING ACUTE ACCENT (U+0301). Both of these extended grapheme clusters are valid ways to represent the character é, and so they’re considered to be canonically equivalent:

1. // "Voulez-vous un café?" using LATIN SMALL LETTER E WITH ACUTE
2. let eAcuteQuestion = "Voulez-vous un caf\u{E9}?"
3. // "Voulez-vous un café?" using LATIN SMALL LETTER E and COMBINING ACUTE ACCENT
4. let combinedEAcuteQuestion = "Voulez-vous un caf\u{65}\u{301}?"
5. if eAcuteQuestion == combinedEAcuteQuestion {
6. print("These two strings are considered equal")
7. }
8. // Prints "These two strings are considered equal"

Conversely, LATIN CAPITAL LETTER A (U+0041, or "A"), as used in English, is *not* equivalent to CYRILLIC CAPITAL LETTER A (U+0410, or "А"), as used in Russian. The characters are visually similar, but don’t have the same linguistic meaning:

1. let latinCapitalLetterA: Character = "\u{41}"
2. let cyrillicCapitalLetterA: Character = "\u{0410}"
3. if latinCapitalLetterA != cyrillicCapitalLetterA {
4. print("These two characters are not equivalent.")
5. }
6. // Prints "These two characters are not equivalent."

String and character comparisons in Swift are not locale-sensitive.

To check whether a string has a particular string prefix or suffix, call the string’s hasPrefix(\_:) and hasSuffix(\_:) methods, both of which take a single argument of type String and return a Boolean value.

Swift provides several different ways to access Unicode representations of strings. You can iterate over the string with a for-in statement, to access its individual Character values as Unicode extended grapheme clusters. This process is described in [Working with Characters](https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID290).

Alternatively, access a String value in one of three other Unicode-compliant representations:

* A collection of UTF-8 code units (accessed with the string’s utf8 property)
* A collection of UTF-16 code units (accessed with the string’s utf16 property)
* A collection of 21-bit Unicode scalar values, equivalent to the string’s UTF-32 encoding form (accessed with the string’s unicodeScalars property)

let dogString = "Dog‼🐶"

for codeUnit in dogString.utf8 {

print("\(codeUnit) ", terminator: "")

}

print("")

// Prints "68 111 103 226 128 188 240 159 144 182 "

for codeUnit in dogString.utf16 {

print("\(codeUnit) ", terminator: "")

}

print("")

// Prints "68 111 103 8252 55357 56374 "

for scalar in dogString.unicodeScalars {

print("\(scalar.value) ", terminator: "")

}

print("")

// Prints "68 111 103 8252 128054 "

As an alternative to querying their value properties, each UnicodeScalar value can also be used to construct a new String value, such as with string interpolation:

for scalar in dogString.unicodeScalars {

print("\(scalar) ")

}

// D

// o

// g

// ‼

// 🐶

Swift provides three primary *collection types*, known as arrays, sets, and dictionaries, for storing collections of values. Arrays are ordered collections of values. Sets are unordered collections of unique values. Dictionaries are unordered collections of key-value associations.

Swift’s array, set, and dictionary types are implemented as *generic collections*. For more about generic types and collections

## Mutability of Collections

If you create an array, a set, or a dictionary, and assign it to a variable, the collection that is created will be mutable. This means that you can change (or mutate) the collection after it’s created by adding, removing, or changing items in the collection. If you assign an array, a set, or a dictionary to a constant, that collection is immutable, and its size and contents cannot be changed.

NOTE

It is good practice to create immutable collections in all cases where the collection does not need to change. Doing so makes it easier for you to reason about your code and enables the Swift compiler to optimize the performance of the collections you create.

## Arrays

An array stores values of the same type in an ordered list. The same value can appear in an array multiple times at different positions.

NOTE

Swift’s Array type is bridged to Foundation’s NSArray class.

### Creating an Empty Array

You can create an empty array of a certain type using initializer syntax:

var someInts = [Int]()

print("someInts is of type [Int] with \(someInts.count) items.")

// Prints "someInts is of type [Int] with 0 items."

Note that the type of the someInts variable is inferred to be [Int] from the type of the initializer.

Alternatively, if the context already provides type information, such as a function argument or an already typed variable or constant, you can create an empty array with an empty array literal, which is written as [] (an empty pair of square brackets):

someInts.append(3)

// someInts now contains 1 value of type Int

someInts = []

// someInts is now an empty array, but is still of type [Int]

### Creating an Array with a Default Value

Swift’s Array type also provides an initializer for creating an array of a certain size with all of its values set to the same default value. You pass this initializer a default value of the appropriate type (called repeating): and the number of times that value is repeated in the new array (called count):

var threeDoubles = Array(repeating: 0.0, count: 3)

// threeDoubles is of type [Double], and equals [0.0, 0.0, 0.0]

### Creating an Array by Adding Two Arrays Together

You can create a new array by adding together two existing arrays with compatible types with the addition operator (+). The new array’s type is inferred from the type of the two arrays you add together:

var anotherThreeDoubles = Array(repeating: 2.5, count: 3)

// anotherThreeDoubles is of type [Double], and equals [2.5, 2.5, 2.5]

var sixDoubles = threeDoubles + anotherThreeDoubles

// sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]

The example below creates an array called shoppingList to store String values:

var shoppingList: [String] = ["Eggs", "Milk"]

// shoppingList has been initialized with two initial items

Use the Boolean isEmpty property as a shortcut for checking whether the count property is equal to 0:

if shoppingList.isEmpty {

print("The shopping list is empty.")

} else {

print("The shopping list is not empty.")

}

// Prints "The shopping list is not empty."

You can add a new item to the end of an array by calling the array’s append(\_:) method:

shoppingList.append("Flour")

// shoppingList now contains 3 items, and someone is making pancakes

Alternatively, append an array of one or more compatible items with the addition assignment operator (+=):

shoppingList += ["Baking Powder"]

// shoppingList now contains 4 items

shoppingList += ["Chocolate Spread", "Cheese", "Butter"]

// shoppingList now contains 7 items

Retrieve a value from the array by using *subscript syntax*, passing the index of the value you want to retrieve within square brackets immediately after the name of the array:

var firstItem = shoppingList[0]

// firstItem is equal to "Eggs"

NOTE

The first item in the array has an index of 0, not 1. Arrays in Swift are always zero-indexed.

You can also use subscript syntax to change a range of values at once, even if the replacement set of values has a different length than the range you are replacing. The following example replaces "Chocolate Spread", "Cheese", and "Butter" with "Bananas" and "Apples":

shoppingList[4...6] = ["Bananas", "Apples"]

// shoppingList now contains 6 items

To insert an item into the array at a specified index, call the array’s insert(\_:at:) method:

1. shoppingList.insert("Maple Syrup", at: 0)
2. // shoppingList now contains 7 items
3. // "Maple Syrup" is now the first item in the list

This call to the insert(\_:at:) method inserts a new item with a value of "Maple Syrup" at the very beginning of the shopping list, indicated by an index of 0.

Similarly, you remove an item from the array with the remove(at:) method. This method removes the item at the specified index and returns the removed item (although you can ignore the returned value if you do not need it):

let mapleSyrup = shoppingList.remove(at: 0)

// the item that was at index 0 has just been removed

// shoppingList now contains 6 items, and no Maple Syrup

// the mapleSyrup constant is now equal to the removed "Maple Syrup" string

Any gaps in an array are closed when an item is removed, and so the value at index 0 is once again equal to "Six eggs":

If you want to remove the final item from an array, use the removeLast() method rather than the remove(at:) method to avoid the need to query the array’s count property. Like the remove(at:) method, removeLast() returns the removed item:

let apples = shoppingList.removeLast()

// the last item in the array has just been removed

// shoppingList now contains 5 items, and no apples

// the apples constant is now equal to the removed "Apples" string

If you need the integer index of each item as well as its value, use the enumerated() method to iterate over the array instead. For each item in the array, the enumerated() method returns a tuple composed of an integer and the item. The integers start at zero and count up by one for each item; if you enumerate over a whole array, these integers match the items’ indices. You can decompose the tuple into temporary constants or variables as part of the iteration:

for (index, value) in shoppingList.enumerated() {

print("Item \(index + 1): \(value)")

}

// Item 1: Six eggs

// Item 2: Milk

// Item 3: Flour

// Item 4: Baking Powder

// Item 5: Bananas

## Sets

A set stores distinct values of the same type in a collection with no defined ordering. You can use a set instead of an array when the order of items is not important, or when you need to ensure that an item only appears once.

NOTE

Swift’s Set type is bridged to Foundation’s NSSet class.

### Hash Values for Set Types

A type must be hashable in order to be stored in a set—that is, the type must provide a way to compute a hash value for itself. A hash value is an Int value that is the same for all objects that compare equally, such that if a == b, it follows that a.hashValue == b.hashValue.

All of Swift’s basic types (such as String, Int, Double, and Bool) are hashable by default, and can be used as set value types or dictionary key types. Enumeration case values without associated values (as described in [Enumerations](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html)) are also hashable by default.

You can use your own custom types as set value types or dictionary key types by making them conform to the Hashable protocol from Swift’s standard library. Types that conform to the Hashable protocol must provide a gettable Int property called hashValue. The value returned by a type’s hashValue property is not required to be the same across different executions of the same program, or in different programs.

Because the Hashable protocol conforms to Equatable, conforming types must also provide an implementation of the equals operator (==). The Equatable protocol requires any conforming implementation of == to be an equivalence relation. That is, an implementation of == must satisfy the following three conditions, for all values a, b, and c:

* a == a (Reflexivity)
* a == b implies b == a (Symmetry)
* a == b && b == c implies a == c (Transitivity)

### Set Type Syntax

The type of a Swift set is written as Set<Element>, where Element is the type that the set is allowed to store. Unlike arrays, sets do not have an equivalent shorthand form.

### Creating and Initializing an Empty Set

You can create an empty set of a certain type using initializer syntax:

var letters = Set<Character>()

print("letters is of type Set<Character> with \(letters.count) items.")

// Prints "letters is of type Set<Character> with 0 items."

letters.insert("a")

// letters now contains 1 value of type Character

letters = []

// letters is now an empty set, but is still of type Set<Character>

### Creating a Set with an Array Literal

You can also initialize a set with an array literal, as a shorthand way to write one or more values as a set collection.

The example below creates a set called favoriteGenres to store String values:

var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]

// favoriteGenres has been initialized with three initial items

The favoriteGenres variable is declared as “a set of String values”, written as Set<String>. Because this particular set has specified a value type of String, it is *only* allowed to store String values. Here, the favoriteGenres set is initialized with three String values ("Rock", "Classical", and "Hip hop"), written within an array literal.

NOTE

The favoriteGenres set is declared as a variable (with the var introducer) and not a constant (with the let introducer) because items are added and removed in the examples below.

### Accessing and Modifying a Set

You access and modify a set through its methods and properties.

To find out the number of items in a set, check its read-only count property:

print("I have \(favoriteGenres.count) favorite music genres.")

// Prints "I have 3 favorite music genres."

Use the Boolean isEmpty property as a shortcut for checking whether the count property is equal to 0:

if favoriteGenres.isEmpty {

print("As far as music goes, I'm not picky.")

} else {

print("I have particular music preferences.")

}

// Prints "I have particular music preferences."

You can add a new item into a set by calling the set’s insert(\_:) method:

favoriteGenres.insert("Jazz")

// favoriteGenres now contains 4 items

You can remove an item from a set by calling the set’s remove(\_:) method, which removes the item if it’s a member of the set, and returns the removed value, or returns nil if the set did not contain it. Alternatively, all items in a set can be removed with its removeAll() method.

if let removedGenre = favoriteGenres.remove("Rock") {

print("\(removedGenre)? I'm over it.")

} else {

print("I never much cared for that.")

}

// Prints "Rock? I'm over it."

To check whether a set contains a particular item, use the contains(\_:) method.

if favoriteGenres.contains("Funk") {

print("I get up on the good foot.")

} else {

print("It's too funky in here.")

}

// Prints "It's too funky in here."

### Iterating Over a Set

You can iterate over the values in a set with a for-in loop.

for genre in favoriteGenres {

print("\(genre)")

}

// Classical

// Jazz

// Hip hop

Swift’s Set type does not have a defined ordering. To iterate over the values of a set in a specific order, use the sorted() method, which returns the set’s elements as an array sorted using the < operator.

for genre in favoriteGenres.sorted() {

print("\(genre)")

}

// Classical

// Hip hop

// Jazz

## Performing Set Operations

You can efficiently perform fundamental set operations, such as combining two sets together, determining which values two sets have in common, or determining whether two sets contain all, some, or none of the same values.

* Use the intersection(\_:) method to create a new set with only the values common to both sets.
* Use the symmetricDifference(\_:) method to create a new set with values in either set, but not both.
* Use the union(\_:) method to create a new set with all of the values in both sets.
* Use the subtracting(\_:) method to create a new set with values not in the specified set.

let oddDigits: Set = [1, 3, 5, 7, 9]

let evenDigits: Set = [0, 2, 4, 6, 8]

let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.union(evenDigits).sorted()

// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

oddDigits.intersection(evenDigits).sorted()

// []

oddDigits.subtracting(singleDigitPrimeNumbers).sorted()

// [1, 9]

oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()

// [1, 2, 9]

* Use the “is equal” operator (==) to determine whether two sets contain all of the same values.
* Use the isSubset(of:) method to determine whether all of the values of a set are contained in the specified set.
* Use the isSuperset(of:) method to determine whether a set contains all of the values in a specified set.
* Use the isStrictSubset(of:) or isStrictSuperset(of:) methods to determine whether a set is a subset or superset, but not equal to, a specified set.
* Use the isDisjoint(with:) method to determine whether two sets have no values in common.

let houseAnimals: Set = ["🐶", "🐱"]

let farmAnimals: Set = ["🐮", "🐔", "🐑", "🐶", "🐱"]

let cityAnimals: Set = ["🐦", "🐭"]

houseAnimals.isSubset(of: farmAnimals)

// true

farmAnimals.isSuperset(of: houseAnimals)

// true

farmAnimals.isDisjoint(with: cityAnimals)

// true

Unlike items in an array, items in a dictionary do not have a specified order.

A dictionary Key type must conform to the Hashable protocol, like a set’s value type.

You can also write the type of a dictionary in shorthand form as [Key: Value]. Although the two forms are functionally identical, the shorthand form is preferred and is used throughout this guide when referring to the type of a dictionary.

### Creating an Empty Dictionary

As with arrays, you can create an empty Dictionary of a certain type by using initializer syntax:

var namesOfIntegers = [Int: String]()

// namesOfIntegers is an empty [Int: String] dictionary

This example creates an empty dictionary of type [Int: String] to store human-readable names of integer values. Its keys are of type Int, and its values are of type String.

If the context already provides type information, you can create an empty dictionary with an empty dictionary literal, which is written as [:] (a colon inside a pair of square brackets):

namesOfIntegers[16] = "sixteen"

// namesOfIntegers now contains 1 key-value pair

namesOfIntegers = [:]

// namesOfIntegers is once again an empty dictionary of type [Int: String]

[key 1: value 1, key 2: value 2, key 3: value 3]

var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]

print("The airports dictionary contains \(airports.count) items.")

// Prints "The airports dictionary contains 2 items."

Use the Boolean isEmpty property as a shortcut for checking whether the count property is equal to 0:

if airports.isEmpty {

print("The airports dictionary is empty.")

} else {

print("The airports dictionary is not empty.")

}

// Prints "The airports dictionary is not empty."

As an alternative to subscripting, use a dictionary’s updateValue(\_:forKey:) method to set or update the value for a particular key. Like the subscript examples above, the updateValue(\_:forKey:) method sets a value for a key if none exists, or updates the value if that key already exists. Unlike a subscript, however, the updateValue(\_:forKey:) method returns the *old* value after performing an update. This enables you to check whether or not an update took place.

The updateValue(\_:forKey:) method returns an optional value of the dictionary’s value type. For a dictionary that stores String values, for example, the method returns a value of type String?, or “optional String”. This optional value contains the old value for that key if one existed before the update, or nil if no value existed:

if let oldValue = airports.updateValue("Dublin Airport", forKey: "DUB") {

print("The old value for DUB was \(oldValue).")

}

// Prints "The old value for DUB was Dublin."

You can also use subscript syntax to retrieve a value from the dictionary for a particular key. Because it is possible to request a key for which no value exists, a dictionary’s subscript returns an optional value of the dictionary’s value type. If the dictionary contains a value for the requested key, the subscript returns an optional value containing the existing value for that key. Otherwise, the subscript returns nil:

if let airportName = airports["DUB"] {

print("The name of the airport is \(airportName).")

} else {

print("That airport is not in the airports dictionary.")

}

// Prints "The name of the airport is Dublin Airport."

You can use subscript syntax to remove a key-value pair from a dictionary by assigning a value of nil for that key:

airports["APL"] = "Apple International"

// "Apple International" is not the real airport for APL, so delete it

airports["APL"] = nil

// APL has now been removed from the dictionary

Alternatively, remove a key-value pair from a dictionary with the removeValue(forKey:) method. This method removes the key-value pair if it exists and returns the removed value, or returns nil if no value existed:

if let removedValue = airports.removeValue(forKey: "DUB") {

print("The removed airport's name is \(removedValue).")

} else {

print("The airports dictionary does not contain a value for DUB.")

}

// Prints "The removed airport's name is Dublin Airport."

### Iterating Over a Dictionary

You can iterate over the key-value pairs in a dictionary with a for-in loop. Each item in the dictionary is returned as a (key, value) tuple, and you can decompose the tuple’s members into temporary constants or variables as part of the iteration:

for (airportCode, airportName) in airports {

print("\(airportCode): \(airportName)")

}

// LHR: London Heathrow

// YYZ: Toronto Pearson

You can also retrieve an iterable collection of a dictionary’s keys or values by accessing its keys and values properties:

for airportCode in airports.keys {

print("Airport code: \(airportCode)")

}

// Airport code: LHR

// Airport code: YYZ

for airportName in airports.values {

print("Airport name: \(airportName)")

}

// Airport name: London Heathrow

// Airport name: Toronto Pearson

If you need to use a dictionary’s keys or values with an API that takes an Array instance, initialize a new array with the keys or values property:

let airportCodes = [String](airports.keys)

// airportCodes is ["LHR", "YYZ"]

let airportNames = [String](airports.values)

// airportNames is ["London Heathrow", "Toronto Pearson"]

Swift’s Dictionary type does not have a defined ordering. To iterate over the keys or values of a dictionary in a specific order, use the sorted() method on its keys or values property.

Swift provides a variety of control flow statements. These include while loops to perform a task multiple times; if, guard, and switch statements to execute different branches of code based on certain conditions; and statements such as break and continue to transfer the flow of execution to another point in your code.

You can also iterate over a dictionary to access its key-value pairs. Each item in the dictionary is returned as a (key, value) tuple when the dictionary is iterated, and you can decompose the (key, value) tuple’s members as explicitly named constants for use within the body of the for-in loop. In the code example below, the dictionary’s keys are decomposed into a constant called animalName, and the dictionary’s values are decomposed into a constant called legCount.

let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]

for (animalName, legCount) in numberOfLegs {

print("\(animalName)s have \(legCount) legs")

}

// cats have 4 legs

// ants have 6 legs

// spiders have 8 legs

The contents of a Dictionary are inherently unordered, and iterating over them does not guarantee the order in which they will be retrieved. In particular, the order you insert items into a Dictionary doesn’t define the order they are iterated.

You can also use for-in loops with numeric ranges. This example prints the first few entries in a five-times table:

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

If you don’t need each value from a sequence, you can ignore the values by using an underscore in place of a variable name.

let base = 3

let power = 10

var answer = 1

for \_ in 1...power {

answer \*= base

}

print("\(base) to the power of \(power) is \(answer)")

// Prints "3 to the power of 10 is 59049"

The underscore character (\_) used in place of a loop variable causes the individual values to be ignored and does not provide access to the current value during each iteration of the loop.

Use the half-open range operator (..<) to include the lower bound but not the upper bound. For more about ranges, see [Range Operators](https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html#ID73).

let minutes = 60

for tickMark in 0..<minutes {

// render the tick mark each minute (60 times)

}

Some users might want fewer tick marks in their UI. They could prefer one mark every 5 minutes instead. Use the stride(from:to:by:) function to skip the unwanted marks.

let minuteInterval = 5

for tickMark in stride(from: 0, to: minutes, by: minuteInterval) {

// render the tick mark every 5 minutes (0, 5, 10, 15 ... 45, 50, 55)

}

Closed ranges are also available, by using stride(from:through:by:) instead:

let hours = 12

let hourInterval = 3

for tickMark in stride(from: 3, through: hours, by: hourInterval) {

// render the tick mark every 3 hours (3, 6, 9, 12)

}

Here’s the general form of a repeat-while loop:

repeat {

statements

} while condition

A switch statement provides an alternative to the if statement for responding to multiple potential states.

In its simplest form, a switch statement compares a value against one or more values of the same type.

switch some value to consider {

case value 1:

respond to value 1

case value 2,

value 3:

respond to value 2 or 3

default:

otherwise, do something else

}

Instead, the entire switch statement finishes its execution as soon as the first matching switch case is completed, without requiring an explicit break statement. This makes the switch statement safer and easier to use than the one in C and avoids executing more than one switch case by mistake.

NOTE

Although break is not required in Swift, you can use a break statement to match and ignore a particular case or to break out of a matched case before that case has completed its execution.

The body of each case *must* contain at least one executable statement. It is not valid to write the following code, because the first case is empty:

let anotherCharacter: Character = "a"

switch anotherCharacter {

case "a": // Invalid, the case has an empty body

case "A":

print("The letter A")

default:

print("Not the letter A")

}

// This will report a compile-time error.

Unlike a switch statement in C, this switch statement does not match both "a" and "A". Rather, it reports a compile-time error that case "a": does not contain any executable statements. This approach avoids accidental fallthrough from one case to another and makes for safer code that is clearer in its intent.

To make a switch with a single case that matches both "a" and "A", combine the two values into a compound case, separating the values with commas.

1. let anotherCharacter: Character = "a"
2. switch anotherCharacter {
3. case "a", "A":
4. print("The letter A")
5. default:
6. print("Not the letter A")
7. }
8. // Prints "The letter A"

For readability, a compound case can also be written over multiple lines.

#### Interval Matching

Values in switch cases can be checked for their inclusion in an interval. This example uses number intervals to provide a natural-language count for numbers of any size:

let approximateCount = 62

let countedThings = "moons orbiting Saturn"

let naturalCount: String

switch approximateCount {

case 0:

naturalCount = "no"

case 1..<5:

naturalCount = "a few"

case 5..<12:

naturalCount = "several"

case 12..<100:

naturalCount = "dozens of"

case 100..<1000:

naturalCount = "hundreds of"

default:

naturalCount = "many"

}

print("There are \(naturalCount) \(countedThings).")

// Prints "There are dozens of moons orbiting Saturn."

In the above example, approximateCount is evaluated in a switch statement. Each case compares that value to a number or interval. Because the value of approximateCount falls between 12 and 100, naturalCount is assigned the value "dozens of", and execution is transferred out of the switch statement

#### Tuples

You can use tuples to test multiple values in the same switch statement. Each element of the tuple can be tested against a different value or interval of values. Alternatively, use the underscore character (\_), also known as the wildcard pattern, to match any possible value.

The example below takes an (x, y) point, expressed as a simple tuple of type (Int, Int), and categorizes it on the graph that follows the example.

let somePoint = (1, 1)

switch somePoint {

case (0, 0):

print("\(somePoint) is at the origin")

case (\_, 0):

print("\(somePoint) is on the x-axis")

case (0, \_):

print("\(somePoint) is on the y-axis")

case (-2...2, -2...2):

print("\(somePoint) is inside the box")

default:

print("\(somePoint) is outside of the box")

}

// Prints "(1, 1) is inside the box"

#### Value Bindings

A switch case can name the value or values it matches to temporary constants or variables, for use in the body of the case. This behavior is known as value binding, because the values are bound to temporary constants or variables within the case’s body.

The example below takes an (x, y) point, expressed as a tuple of type (Int, Int), and categorizes it on the graph that follows:

let anotherPoint = (2, 0)

switch anotherPoint {

case (let x, 0):

print("on the x-axis with an x value of \(x)")

case (0, let y):

print("on the y-axis with a y value of \(y)")

case let (x, y):

print("somewhere else at (\(x), \(y))")

}

// Prints "on the x-axis with an x value of 2"

The three switch cases declare placeholder constants x and y, which temporarily take on one or both tuple values from anotherPoint. The first case, case (let x, 0), matches any point with a y value of 0 and assigns the point’s x value to the temporary constant x. Similarly, the second case, case (0, let y), matches any point with an x value of 0 and assigns the point’s y value to the temporary constant y.

After the temporary constants are declared, they can be used within the case’s code block. Here, they are used to print the categorization of the point.

This switch statement does not have a default case. The final case, case let (x, y), declares a tuple of two placeholder constants that can match any value. Because anotherPoint is always a tuple of two values, this case matches all possible remaining values, and a default case is not needed to make the switch statement exhaustive.

#### Where

A switch case can use a where clause to check for additional conditions.

The example below categorizes an (x, y) point on the following graph:

let yetAnotherPoint = (1, -1)

switch yetAnotherPoint {

case let (x, y) where x == y:

print("(\(x), \(y)) is on the line x == y")

case let (x, y) where x == -y:

print("(\(x), \(y)) is on the line x == -y")

case let (x, y):

print("(\(x), \(y)) is just some arbitrary point")

}

// Prints "(1, -1) is on the line x == -y"

The switch statement determines whether the point is on the green diagonal line where x == y, on the purple diagonal line where x == -y, or neither.

The three switch cases declare placeholder constants x and y, which temporarily take on the two tuple values from yetAnotherPoint. These constants are used as part of a where clause, to create a dynamic filter. The switch case matches the current value of point only if the where clause’s condition evaluates to true for that value.

As in the previous example, the final case matches all possible remaining values, and so a default case is not needed to make the switch statement exhaustive.

#### Compound Cases

Multiple switch cases that share the same body can be combined by writing several patterns after case, with a comma between each of the patterns. If any of the patterns match, then the case is considered to match. The patterns can be written over multiple lines if the list is long. For example:

let someCharacter: Character = "e"

switch someCharacter {

case "a", "e", "i", "o", "u":

print("\(someCharacter) is a vowel")

case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",

"n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":

print("\(someCharacter) is a consonant")

default:

print("\(someCharacter) is not a vowel or a consonant")

}

// Prints "e is a vowel"

The switch statement’s first case matches all five lowercase vowels in the English language. Similarly, its second case matches all lowercase English consonants. Finally, the default case matches any other character.

Compound cases can also include value bindings. All of the patterns of a compound case have to include the same set of value bindings, and each binding has to get a value of the same type from all of the patterns in the compound case. This ensures that, no matter which part of the compound case matched, the code in the body of the case can always access a value for the bindings and that the value always has the same type.

let stillAnotherPoint = (9, 0)

switch stillAnotherPoint {

case (let distance, 0), (0, let distance):

print("On an axis, \(distance) from the origin")

default:

print("Not on an axis")

}

// Prints "On an axis, 9 from the origin"

The case above has two patterns: (let distance, 0) matches points on the x-axis and (0, let distance) matches points on the y-axis. Both patterns include a binding for distance and distance is an integer in both patterns—which means that the code in the body of the case can always access a value for distance.

Control Transfer Statements

*Control transfer statements* change the order in which your code is executed, by transferring control from one piece of code to another. Swift has five control transfer statements:

* continue
* break
* fallthrough
* return
* throw

### Continue

The continue statement tells a loop to stop what it is doing and start again at the beginning of the next iteration through the loop. It says “I am done with the current loop iteration” without leaving the loop altogether.

The following example removes all vowels and spaces from a lowercase string to create a cryptic puzzle phrase:

let puzzleInput = "great minds think alike"

var puzzleOutput = ""

let charactersToRemove: [Character] = ["a", "e", "i", "o", "u", " "]

for character in puzzleInput {

if charactersToRemove.contains(character) {

continue

}

puzzleOutput.append(character)

}

print(puzzleOutput)

// Prints "grtmndsthnklk"

### Break

The break statement ends execution of an entire control flow statement immediately. The break statement can be used inside a switch or loop statement when you want to terminate the execution of the switch or loop statement earlier than would otherwise be the case.

#### Break in a Loop Statement

When used inside a loop statement, break ends the loop’s execution immediately and transfers control to the code after the loop’s closing brace (}). No further code from the current iteration of the loop is executed, and no further iterations of the loop are started.

#### Break in a Switch Statement

When used inside a switch statement, break causes the switch statement to end its execution immediately and to transfer control to the code after the switch statement’s closing brace (}).

This behavior can be used to match and ignore one or more cases in a switch statement. Because Swift’s switch statement is exhaustive and does not allow empty cases, it is sometimes necessary to deliberately match and ignore a case in order to make your intentions explicit. You do this by writing the break statement as the entire body of the case you want to ignore. When that case is matched by the switch statement, the break statement inside the case ends the switch statement’s execution immediately.

NOTE

A switch case that contains only a comment is reported as a compile-time error. Comments are not statements and do not cause a switch case to be ignored. Always use a break statement to ignore a switch case.

The following example switches on a Character value and determines whether it represents a number symbol in one of four languages. For brevity, multiple values are covered in a single switch case.

1. let numberSymbol: Character = "三" // Chinese symbol for the number 3
2. var possibleIntegerValue: Int?
3. switch numberSymbol {
4. case "1", "١", "一", "๑":
5. possibleIntegerValue = 1
6. case "2", "٢", "二", "๒":
7. possibleIntegerValue = 2
8. case "3", "٣", "三", "๓":
9. possibleIntegerValue = 3
10. case "4", "٤", "四", "๔":
11. possibleIntegerValue = 4
12. default:
13. break
14. }
15. if let integerValue = possibleIntegerValue {
16. print("The integer value of \(numberSymbol) is \(integerValue).")
17. } else {
18. print("An integer value could not be found for \(numberSymbol).")
19. }
20. // Prints "The integer value of 三 is 3."

This example checks numberSymbol to determine whether it is a Latin, Arabic, Chinese, or Thai symbol for the numbers 1 to 4. If a match is found, one of the switch statement’s cases sets an optional Int? variable called possibleIntegerValue to an appropriate integer value.

After the switch statement completes its execution, the example uses optional binding to determine whether a value was found. The possibleIntegerValue variable has an implicit initial value of nil by virtue of being an optional type, and so the optional binding will succeed only if possibleIntegerValue was set to an actual value by one of the switch statement’s first four cases.

Because it’s not practical to list every possible Character value in the example above, a default case handles any characters that are not matched. This default case does not need to perform any action, and so it is written with a single break statement as its body. As soon as the default case is matched, the break statement ends the switch statement’s execution, and code execution continues from the if let statement.

### Fallthrough

In Swift, switch statements don’t fall through the bottom of each case and into the next one. That is, the entire switch statement completes its execution as soon as the first matching case is completed. By contrast, C requires you to insert an explicit break statement at the end of every switch case to prevent fallthrough. Avoiding default fallthrough means that Swift switch statements are much more concise and predictable than their counterparts in C, and thus they avoid executing multiple switch cases by mistake.

If you need C-style fallthrough behavior, you can opt in to this behavior on a case-by-case basis with the fallthrough keyword. The example below uses fallthrough to create a textual description of a number.

let integerToDescribe = 5

var description = "The number \(integerToDescribe) is"

switch integerToDescribe {

case 2, 3, 5, 7, 11, 13, 17, 19:

description += " a prime number, and also"

fallthrough

default:

description += " an integer."

}

print(description)

// Prints "The number 5 is a prime number, and also an integer."

This example declares a new String variable called description and assigns it an initial value. The function then considers the value of integerToDescribe using a switch statement. If the value of integerToDescribe is one of the prime numbers in the list, the function appends text to the end of description, to note that the number is prime. It then uses the fallthrough keyword to “fall into” the default case as well. The default case adds some extra text to the end of the description, and the switch statement is complete.

NOTE

The fallthrough keyword does not check the case conditions for the switch case that it causes execution to fall into. The fallthrough keyword simply causes code execution to move directly to the statements inside the next case (or default case) block, as in C’s standard switch statement behavior.

### Labeled Statements

In Swift, you can nest loops and conditional statements inside other loops and conditional statements to create complex control flow structures. However, loops and conditional statements can both use the break statement to end their execution prematurely. Therefore, it is sometimes useful to be explicit about which loop or conditional statement you want a break statement to terminate. Similarly, if you have multiple nested loops, it can be useful to be explicit about which loop the continue statement should affect.

To achieve these aims, you can mark a loop statement or conditional statement with a statement label. With a conditional statement, you can use a statement label with the break statement to end the execution of the labeled statement. With a loop statement, you can use a statement label with the break or continue statement to end or continue the execution of the labeled statement.

A labeled statement is indicated by placing a label on the same line as the statement’s introducer keyword, followed by a colon. Here’s an example of this syntax for a while loop, although the principle is the same for all loops and switch statements:

1. label name: while condition {
2. statements
3. }

If the break statement above did not use the gameLoop label, it would break out of the switch statement, not the while statement. Using the gameLoop label makes it clear which control statement should be terminated.

It is not strictly necessary to use the gameLoop label when calling continue gameLoop to jump to the next iteration of the loop. There is only one loop in the game, and therefore no ambiguity as to which loop the continue statement will affect. However, there is no harm in using the gameLoop label with the continue statement. Doing so is consistent with the label’s use alongside the break statement and helps make the game’s logic clearer to read and understand.

The while loop’s condition is while square != finalSquare, to reflect that you must land exactly on square 25.

1. gameLoop: while square != finalSquare {
2. diceRoll += 1
3. if diceRoll == 7 { diceRoll = 1 }
4. switch square + diceRoll {
5. case finalSquare:
6. // diceRoll will move us to the final square, so the game is over
7. break gameLoop
8. case let newSquare where newSquare > finalSquare:
9. // diceRoll will move us beyond the final square, so roll again
10. continue gameLoop
11. default:
12. // this is a valid move, so find out its effect
13. square += diceRoll
14. square += board[square]
15. }
16. }
17. print("Game over!")

The dice is rolled at the start of each loop. Rather than moving the player immediately, the loop uses a switch statement to consider the result of the move and to determine whether the move is allowed:

* If the dice roll will move the player onto the final square, the game is over. The break gameLoop statement transfers control to the first line of code outside of the while loop, which ends the game.
* If the dice roll will move the player *beyond* the final square, the move is invalid and the player needs to roll again. The continue gameLoop statement ends the current while loop iteration and begins the next iteration of the loop.
* In all other cases, the dice roll is a valid move. The player moves forward by diceRoll squares, and the game logic checks for any snakes and ladders. The loop then ends, and control returns to the while condition to decide whether another turn is required.

func greet(person: [String: String]) {

guard let name = person["name"] else {

return

}

print("Hello \(name)!")

guard let location = person["location"] else {

print("I hope the weather is nice near you.")

return

}

print("I hope the weather is nice in \(location).")

}

greet(person: ["name": "John"])

// Prints "Hello John!"

// Prints "I hope the weather is nice near you."

greet(person: ["name": "Jane", "location": "Cupertino"])

// Prints "Hello Jane!"

// Prints "I hope the weather is nice in Cupertino."

## Checking API Availability

Swift has built-in support for checking API availability, which ensures that you don’t accidentally use APIs that are unavailable on a given deployment target.

The compiler uses availability information in the SDK to verify that all of the APIs used in your code are available on the deployment target specified by your project. Swift reports an error at compile time if you try to use an API that isn’t available.

You use an *availability condition* in an if or guard statement to conditionally execute a block of code, depending on whether the APIs you want to use are available at runtime. The compiler uses the information from the availability condition when it verifies that the APIs in that block of code are available.

if #available(iOS 10, macOS 10.12, \*) {

// Use iOS 10 APIs on iOS, and use macOS 10.12 APIs on macOS

} else {

// Fall back to earlier iOS and macOS APIs

}

The availability condition above specifies that in iOS, the body of the if statement executes only in iOS 10 and later; in macOS, only in macOS 10.12 and later. The last argument, \*, is required and specifies that on any other platform, the body of the if executes on the minimum deployment target specified by your target.

In its general form, the availability condition takes a list of platform names and versions. You use platform names such as iOS, macOS, watchOS, and tvOS—for the full list, see [Declaration Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html#ID348). In addition to specifying major version numbers like iOS 8 or macOS 10.10, you can specify minor versions numbers like iOS 11.2.6 and macOS 10.13.3.

1. if #available(platform name version, ..., \*) {
2. statements to execute if the APIs are available
3. } else {
4. fallback statements to execute if the APIs are unavailable
5. }

The return value of a function can be ignored when it is called:

func printAndCount(string: String) -> Int {

print(string)

return string.count

}

func printWithoutCounting(string: String) {

let \_ = printAndCount(string: string)

}

printAndCount(string: "hello, world")

// prints "hello, world" and returns a value of 12

printWithoutCounting(string: "hello, world")

// prints "hello, world" but does not return a value

Return values can be ignored, but a function that says it will return a value must always do so. A function with a defined return type cannot allow control to fall out of the bottom of the function without returning a value, and attempting to do so will result in a compile-time error.

### Functions with Multiple Return Values

You can use a tuple type as the return type for a function to return multiple values as part of one compound return value.

The example below defines a function called minMax(array:), which finds the smallest and largest numbers in an array of Int values:

func minMax(array: [Int]) -> (min: Int, max: Int) {

var currentMin = array[0]

var currentMax = array[0]

for value in array[1..<array.count] {

if value < currentMin {

currentMin = value

} else if value > currentMax {

currentMax = value

}

}

return (currentMin, currentMax)

}

let bounds = minMax(array: [8, -6, 2, 109, 3, 71])

print("min is \(bounds.min) and max is \(bounds.max)")

// Prints "min is -6 and max is 109"

#### Optional Tuple Return Types

If the tuple type to be returned from a function has the potential to have “no value” for the entire tuple, you can use an optional tuple return type to reflect the fact that the entire tuple can be nil. You write an optional tuple return type by placing a question mark after the tuple type’s closing parenthesis, such as (Int, Int)? or (String, Int, Bool)?.

NOTE

An optional tuple type such as (Int, Int)? is different from a tuple that contains optional types such as (Int?, Int?). With an optional tuple type, the entire tuple is optional, not just each individual value within the tuple.

To handle an empty array safely, write the minMax(array:) function with an optional tuple return type and return a value of nil when the array is empty:

func minMax(array: [Int]) -> (min: Int, max: Int)? {

if array.isEmpty { return nil }

var currentMin = array[0]

var currentMax = array[0]

for value in array[1..<array.count] {

if value < currentMin {

currentMin = value

} else if value > currentMax {

currentMax = value

}

}

return (currentMin, currentMax)

}

### Functions With an Implicit Return

If the entire body of the function is a single expression, the function implicitly returns that expression. For example, both functions below have the same behavior:

func greeting(for person: String) -> String {

"Hello, " + person + "!"

}

print(greeting(for: "Dave"))

// Prints "Hello, Dave!"

func anotherGreeting(for person: String) -> String {

return "Hello, " + person + "!"

}

print(anotherGreeting(for: "Dave"))

// Prints "Hello, Dave!"

Function Argument Labels and Parameter Names

Each function parameter has both an *argument label* and a *parameter name*. The argument label is used when calling the function; each argument is written in the function call with its argument label before it. The parameter name is used in the implementation of the function. By default, parameters use their parameter name as their argument label.

func someFunction(firstParameterName: Int, secondParameterName: Int) {

// In the function body, firstParameterName and secondParameterName

// refer to the argument values for the first and second parameters.

}

someFunction(firstParameterName: 1, secondParameterName: 2)

All parameters must have unique names. Although it’s possible for multiple parameters to have the same argument label, unique argument labels help make your code more readable.

### Specifying Argument Labels

You write an argument label before the parameter name, separated by a space:

func someFunction(argumentLabel parameterName: Int) {

// In the function body, parameterName refers to the argument value

// for that parameter.

}

Here’s a variation of the greet(person:) function that takes a person’s name and hometown and returns a greeting:

func greet(person: String, from hometown: String) -> String {

return "Hello \(person)! Glad you could visit from \(hometown)."

}

print(greet(person: "Bill", from: "Cupertino"))

// Prints "Hello Bill! Glad you could visit from Cupertino."

The use of argument labels can allow a function to be called in an expressive, sentence-like manner, while still providing a function body that is readable and clear in intent.

### Omitting Argument Labels

If you don’t want an argument label for a parameter, write an underscore (\_) instead of an explicit argument label for that parameter.

func someFunction(\_ firstParameterName: Int, secondParameterName: Int) {

// In the function body, firstParameterName and secondParameterName

// refer to the argument values for the first and second parameters.

}

someFunction(1, secondParameterName: 2)

If a parameter has an argument label, the argument must be labeled when you call the function.

### Default Parameter Values

You can define a default value for any parameter in a function by assigning a value to the parameter after that parameter’s type. If a default value is defined, you can omit that parameter when calling the function.

func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) {

// If you omit the second argument when calling this function, then

// the value of parameterWithDefault is 12 inside the function body.

}

someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6) // parameterWithDefault is 6

someFunction(parameterWithoutDefault: 4) // parameterWithDefault is 12

Place parameters that don’t have default values at the beginning of a function’s parameter list, before the parameters that have default values. Parameters that don’t have default values are usually more important to the function’s meaning—writing them first makes it easier to recognize that the same function is being called, regardless of whether any default parameters are omitted.

### Variadic Parameters

A variadic parameter accepts zero or more values of a specified type. You use a variadic parameter to specify that the parameter can be passed a varying number of input values when the function is called. Write variadic parameters by inserting three period characters (...) after the parameter’s type name.

The values passed to a variadic parameter are made available within the function’s body as an array of the appropriate type. For example, a variadic parameter with a name of numbers and a type of Double... is made available within the function’s body as a constant array called numbers of type [Double].

The example below calculates the arithmetic mean (also known as the average) for a list of numbers of any length:

func arithmeticMean(\_ numbers: Double...) -> Double {

var total: Double = 0

for number in numbers {

total += number

}

return total / Double(numbers.count)

}

arithmeticMean(1, 2, 3, 4, 5)

// returns 3.0, which is the arithmetic mean of these five numbers

arithmeticMean(3, 8.25, 18.75)

// returns 10.0, which is the arithmetic mean of these three numbers

NOTE

A function may have at most one variadic parameter.

### In-Out Parameters

Function parameters are constants by default. Trying to change the value of a function parameter from within the body of that function results in a compile-time error. This means that you can’t change the value of a parameter by mistake. If you want a function to modify a parameter’s value, and you want those changes to persist after the function call has ended, define that parameter as an in-out parameter instead.

You write an in-out parameter by placing the inout keyword right before a parameter’s type. An in-out parameter has a value that is passed in to the function, is modified by the function, and is passed back out of the function to replace the original value.

You can only pass a variable as the argument for an in-out parameter. You cannot pass a constant or a literal value as the argument, because constants and literals cannot be modified. You place an ampersand (&) directly before a variable’s name when you pass it as an argument to an in-out parameter, to indicate that it can be modified by the function.

NOTE

In-out parameters cannot have default values, and variadic parameters cannot be marked as inout.

Here’s an example of a function called swapTwoInts(\_:\_:), which has two in-out integer parameters called a and b:

func swapTwoInts(\_ a: inout Int, \_ b: inout Int) {

let temporaryA = a

a = b

b = temporaryA

}

var someInt = 3

var anotherInt = 107

swapTwoInts(&someInt, &anotherInt)

print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")

// Prints "someInt is now 107, and anotherInt is now 3"

### Function Types as Return Types

You can use a function type as the return type of another function. You do this by writing a complete function type immediately after the return arrow (->) of the returning function.

The next example defines two simple functions called stepForward(\_:) and stepBackward(\_:). The stepForward(\_:) function returns a value one more than its input value, and the stepBackward(\_:) function returns a value one less than its input value. Both functions have a type of (Int) -> Int:

1. func stepForward(\_ input: Int) -> Int {
2. return input + 1
3. }
4. func stepBackward(\_ input: Int) -> Int {
5. return input - 1
6. }

Here’s a function called chooseStepFunction(backward:), whose return type is (Int) -> Int. The chooseStepFunction(backward:) function returns the stepForward(\_:) function or the stepBackward(\_:) function based on a Boolean parameter called backward:

1. func chooseStepFunction(backward: Bool) -> (Int) -> Int {
2. return backward ? stepBackward : stepForward
3. }

You can now use chooseStepFunction(backward:) to obtain a function that will step in one direction or the other:

1. var currentValue = 3
2. let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
3. // moveNearerToZero now refers to the stepBackward() function

The example above determines whether a positive or negative step is needed to move a variable called currentValue progressively closer to zero. currentValue has an initial value of 3, which means that currentValue > 0 returns true, causing chooseStepFunction(backward:) to return the stepBackward(\_:) function. A reference to the returned function is stored in a constant called moveNearerToZero.

Now that moveNearerToZero refers to the correct function, it can be used to count to zero:

print("Counting to zero:")

// Counting to zero:

while currentValue != 0 {

print("\(currentValue)... ")

currentValue = moveNearerToZero(currentValue)

}

print("zero!")

// 3...

// 2...

// 1...

// zero!

## Nested Functions

All of the functions you have encountered so far in this chapter have been examples of global functions, which are defined at a global scope. You can also define functions inside the bodies of other functions, known as nested functions.

Nested functions are hidden from the outside world by default, but can still be called and used by their enclosing function. An enclosing function can also return one of its nested functions to allow the nested function to be used in another scope.

You can rewrite the chooseStepFunction(backward:) example above to use and return nested functions:

func chooseStepFunction(backward: Bool) -> (Int) -> Int {

func stepForward(input: Int) -> Int { return input + 1 }

func stepBackward(input: Int) -> Int { return input - 1 }

return backward ? stepBackward : stepForward

}

var currentValue = -4

let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)

// moveNearerToZero now refers to the nested stepForward() function

while currentValue != 0 {

print("\(currentValue)... ")

currentValue = moveNearerToZero(currentValue)

}

print("zero!")

// -4...

// -3...

// -2...

// -1...

// zero!

# Closures

Closures are self-contained blocks of functionality that can be passed around and used in your code. Closures in Swift are similar to blocks in C and Objective-C and to lambdas in other programming languages.

Global and nested functions, as introduced in [Functions](https://docs.swift.org/swift-book/LanguageGuide/Functions.html), are actually special cases of closures. Closures take one of three forms:

* Global functions are closures that have a name and do not capture any values.
* Nested functions are closures that have a name and can capture values from their enclosing function.
* Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context.

### Closure Expression Syntax

Closure expression syntax has the following general form:

{ (parameters) -> return type in

statements

}

The *parameters* in closure expression syntax can be in-out parameters, but they can’t have a default value. Variadic parameters can be used if you name the variadic parameter. Tuples can also be used as parameter types and return types.

The example below shows a closure expression version of the backward(\_:\_:) function from above:

reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in

return s1 > s2

})

The start of the closure’s body is introduced by the in keyword. This keyword indicates that the definition of the closure’s parameters and return type has finished, and the body of the closure is about to begin.

### Inferring Type From Context

Because the sorting closure is passed as an argument to a method, Swift can infer the types of its parameters and the type of the value it returns. The sorted(by:) method is being called on an array of strings, so its argument must be a function of type (String, String) -> Bool. This means that the (String, String) and Bool types do not need to be written as part of the closure expression’s definition. Because all of the types can be inferred, the return arrow (->) and the parentheses around the names of the parameters can also be omitted:

1. reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )

It is always possible to infer the parameter types and return type when passing a closure to a function or method as an inline closure expression. As a result, you never need to write an inline closure in its fullest form when the closure is used as a function or method argument.

Nonetheless, you can still make the types explicit if you wish, and doing so is encouraged if it avoids ambiguity for readers of your code. In the case of the sorted(by:) method, the purpose of the closure is clear from the fact that sorting is taking place, and it is safe for a reader to assume that the closure is likely to be working with String values, because it is assisting with the sorting of an array of strings.

### Implicit Returns from Single-Expression Closures

Single-expression closures can implicitly return the result of their single expression by omitting the return keyword from their declaration, as in this version of the previous example:

1. reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )

Here, the function type of the sorted(by:) method’s argument makes it clear that a Bool value must be returned by the closure. Because the closure’s body contains a single expression (s1 > s2) that returns a Bool value, there is no ambiguity, and the return keyword can be omitted.

### Shorthand Argument Names

Swift automatically provides shorthand argument names to inline closures, which can be used to refer to the values of the closure’s arguments by the names $0, $1, $2, and so on.

If you use these shorthand argument names within your closure expression, you can omit the closure’s argument list from its definition, and the number and type of the shorthand argument names will be inferred from the expected function type. The in keyword can also be omitted, because the closure expression is made up entirely of its body:

1. reversedNames = names.sorted(by: { $0 > $1 } )

Here, $0 and $1 refer to the closure’s first and second String arguments.

### Operator Methods

There’s actually an even shorter way to write the closure expression above. Swift’s String type defines its string-specific implementation of the greater-than operator (>) as a method that has two parameters of type String, and returns a value of type Bool. This exactly matches the method type needed by the sorted(by:) method. Therefore, you can simply pass in the greater-than operator, and Swift will infer that you want to use its string-specific implementation:

reversedNames = names.sorted(by: >)

## Trailing Closures

If you need to pass a closure expression to a function as the function’s final argument and the closure expression is long, it can be useful to write it as a trailing closure instead. A trailing closure is written after the function call’s parentheses, even though it is still an argument to the function. When you use the trailing closure syntax, you don’t write the argument label for the closure as part of the function call.

func someFunctionThatTakesAClosure(closure: () -> Void) {

// function body goes here

}

// Here's how you call this function without using a trailing closure:

someFunctionThatTakesAClosure(closure: {

// closure's body goes here

})

// Here's how you call this function with a trailing closure instead:

someFunctionThatTakesAClosure() {

// trailing closure's body goes here

}

Trailing closures are most useful when the closure is sufficiently long that it is not possible to write it inline on a single line. As an example, Swift’s Array type has a map(\_:) method which takes a closure expression as its single argument. The closure is called once for each item in the array, and returns an alternative mapped value (possibly of some other type) for that item. The nature of the mapping and the type of the returned value is left up to the closure to specify.

After applying the provided closure to each array element, the map(\_:) method returns a new array containing all of the new mapped values, in the same order as their corresponding values in the original array.

Here’s how you can use the map(\_:) method with a trailing closure to convert an array of Int values into an array of String values. The array [16, 58, 510] is used to create the new array ["OneSix", "FiveEight", "FiveOneZero"]:

1. let digitNames = [
2. 0: "Zero", 1: "One", 2: "Two", 3: "Three", 4: "Four",
3. 5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
4. ]
5. let numbers = [16, 58, 510]

The code above creates a dictionary of mappings between the integer digits and English-language versions of their names. It also defines an array of integers, ready to be converted into strings.

You can now use the numbers array to create an array of String values, by passing a closure expression to the array’s map(\_:) method as a trailing closure:

let strings = numbers.map { (number) -> String in

var number = number

var output = ""

repeat {

output = digitNames[number % 10]! + output

number /= 10

} while number > 0

return output

}

// strings is inferred to be of type [String]

// its value is ["OneSix", "FiveEight", "FiveOneZero"]

Geofencing is a powerful technology with many practical and far-reaching applications in such realms as marketing, resource management, security, parental control and even gaming — what you can achieve is really up to your imagination.

# Monitoring the User's Proximity to Geographic Regions

Use region monitoring to determine when the user enters or leaves a geographic region.

Region monitoring (also known as geofencing) is a way for your app to be alerted when the user enters or exits a geographical region. You might use region monitoring to perform location-related tasks. For example, the Reminders app uses them to trigger reminders when the user arrives at or leaves a specified location,

In iOS, regions are monitored by the system, which wakes up your app as needed when the user crosses a defined region boundary. In macOS, region monitoring works only while the app is running (either in the foreground or background) and the user’s system is awake. The system does not launch Mac apps to deliver region-related notifications.

### Define and Monitor a Geographic Region

A region is a circular area centered on a geographic coordinate, and you define one using a [CLCircularRegion](https://developer.apple.com/documentation/corelocation/clcircularregion?language=objc) object. The radius of the region object defines its boundary. You define the regions you want to monitor and register them with the system by calling the [startMonitoringForRegion:](https://developer.apple.com/documentation/corelocation/cllocationmanager/1423656-startmonitoringforregion?language=objc) method of your [CLLocationManager](https://developer.apple.com/documentation/corelocation/cllocationmanager?language=objc) object. The system monitors your regions until you explicitly ask it to stop or until the device reboots.

### Define and Monitor a Geographic Region

A region is a circular area centered on a geographic coordinate, and you define one using a [CLCircularRegion](https://developer.apple.com/documentation/corelocation/clcircularregion?language=objc) object. The radius of the region object defines its boundary. You define the regions you want to monitor and register them with the system by calling the [startMonitoringForRegion:](https://developer.apple.com/documentation/corelocation/cllocationmanager/1423656-startmonitoringforregion?language=objc) method of your [CLLocationManager](https://developer.apple.com/documentation/corelocation/cllocationmanager?language=objc) object. The system monitors your regions until you explicitly ask it to stop or until the device reboots.

[Listing 1](https://developer.apple.com/documentation/corelocation/monitoring_the_user_s_proximity_to_geographic_regions?language=objc#2854905) shows how to configure and register a region centered around a point provided by the caller of the method. The method uses the largest allowed radius to define the boundaries of the region and asks that the system deliver notifications only when the user enters the region.

**Listing 1**

Monitoring a region around the specified coordinate

func monitorRegionAtLocation(center: CLLocationCoordinate2D, identifier: String ) { // Make sure the devices supports region monitoring. if CLLocationManager.isMonitoringAvailable(for: CLCircularRegion.self) { // Register the region. let maxDistance = locationManager.maximumRegionMonitoringDistance let region = CLCircularRegion(center: center, radius: maxDistance, identifier: identifier) region.notifyOnEntry = true region.notifyOnExit = false locationManager.startMonitoring(for: region) } }

**Tip**

Regions are shared resources that rely on specific hardware capabilities. To ensure that all apps can participate in region monitoring, Core Location prevents any single app from monitoring more than 20 regions simultaneously. To work around this limitation, monitor only regions that are close to the user’s current location. As the user moves, update the list based on the user’s new location.

### Handle a Region-Related Notification

Whenever the user crosses the boundary of one of your app's registered regions, the system notifies your app. If an iOS app is not running when the boundary crossing occurs, the system tries to launch it. An iOS app that supports region monitoring must enable the Location updates background mode so that it can be launched in the background.

Boundary crossing notifications are delivered to your location manager's delegate object. Specifically, the location manager calls the [locationManager:didEnterRegion:](https://developer.apple.com/documentation/corelocation/cllocationmanagerdelegate/1423560-locationmanager?language=objc) or [locationManager:didExitRegion:](https://developer.apple.com/documentation/corelocation/cllocationmanagerdelegate/1423630-locationmanager?language=objc) methods of its delegate. If your app was launched, you must configure a [CLLocationManager](https://developer.apple.com/documentation/corelocation/cllocationmanager?language=objc) object and delegate object right away so that you can receive these notifications. To determine whether your app was launched for a location event, look for the [UIApplicationLaunchOptionsKey](https://developer.apple.com/documentation/uikit/uiapplicationlaunchoptionskey?language=objc) in the launch options dictionary.

When determining whether a boundary crossing happened, the system waits to be sure before sending the notification. Specifically, the user must travel a minimum distance over the boundary and remain on the same side of the boundary for at least 20 seconds. These conditions help eliminate spurious calls to your delegate object’s methods.

[Listing 2](https://developer.apple.com/documentation/corelocation/monitoring_the_user_s_proximity_to_geographic_regions?language=objc#2854906) shows a delegate method that is called when the user enters a registered region. Regions have an associated identifier, which this method uses to look up information related to the region and perform the associated action.

**Listing 2**

Handling a region-entered notification

func locationManager(\_ manager: CLLocationManager, didEnterRegion region: CLRegion) { if let region = region as? CLCircularRegion { let identifier = region.identifier triggerTaskAssociatedWithRegionIdentifier(regionID: identifier) } }

Termination of your app (by the user or the system) does not prevent it from being relaunched to handle region boundary crossings. However, when Background App Refresh is disabled, either for your app or for all apps, the user must explicitly launch your app to resume the delivery of all location-related events.

### Region Monitoring

[CLCircularRegion](https://developer.apple.com/documentation/corelocation/clcircularregion?language=objc)

A circular geographic region, specified as a center point and radius.

[CLRegion](https://developer.apple.com/documentation/corelocation/clregion?language=objc)

A base class representing an area that can be monitored.

# What is the best iOS geofencing, region monitoring, and Core Location alternative?

Radar is more powerful than native iOS geofencing. Radar builds on top of Core Location to support unlimited geofences, polygon geofences, and stop detection. Radar also supports home, work, traveling, place, chain, and category detection, even if you haven't set up geofences for those places. Radar is a full-stack solution with iOS and Android SDKs, an API, a dashboard, webhooks, and integrations.

## What Is Geofencing?

A geofence is an invisible physical region on the surface of the Earth. Notifications are sent when a location-capable device physically enters or exits that region.

Geofences in iOS are circular regions with a latitude and longitude at the center and a radius. To register our intent to be notified when an iOS device enters or exits one or more geofences, we use the [CoreLocation framework](https://developer.apple.com/library/ios/documentation/CoreLocation/Reference/CoreLocation_Framework/index.html), asking it to start monitoring for the geofence regions we’ve provided.

We use geofences (instead of simply asking the device to deliver constant location updates) for two reasons:

* Efficiency – It doesn’t take nearly as much power to let iOS do its own geofence calculations.
* Privacy – We don’t need to know everywhere the user goes just so that we can figure out that they crossed over into our interesting area.

# Enumerations

enum SomeEnumeration {

// enumeration definition goes here

}

Here’s an example for the four main points of a compass:

enum CompassPoint {

case north

case south

case east

case west

}

The values defined in an enumeration (such as north, south, east, and west) are its *enumeration cases*. You use the case keyword to introduce new enumeration cases.

NOTE

Swift enumeration cases don’t have an integer value set by default, unlike languages like C and Objective-C. In the CompassPoint example above, north, south, east and west don’t implicitly equal 0, 1, 2 and 3. Instead, the different enumeration cases are values in their own right, with an explicitly defined type of CompassPoint.

Multiple cases can appear on a single line, separated by commas:

enum Planet {

case mercury, venus, earth, mars, jupiter, saturn, uranus, neptune

}

Each enumeration definition defines a new type. Like other types in Swift, their names (such as CompassPoint and Planet) start with a capital letter. Give enumeration types singular rather than plural names, so that they read as self-evident:

var directionToHead = CompassPoint.west

## terating over Enumeration Cases

For some enumerations, it’s useful to have a collection of all of that enumeration’s cases. You enable this by writing : CaseIterable after the enumeration’s name. Swift exposes a collection of all the cases as an allCases property of the enumeration type. Here’s an example:

enum Beverage: CaseIterable {

case coffee, tea, juice

}

let numberOfChoices = Beverage.allCases.count

print("\(numberOfChoices) beverages available")

// Prints "3 beverages available"

In the example above, you write Beverage.allCases to access a collection that contains all of the cases of the Beverage enumeration. You can use allCases like any other collection—the collection’s elements are instances of the enumeration type, so in this case they’re Beverage values. The example above counts how many cases there are, and the example below uses a for loop to iterate over all the cases.

for beverage in Beverage.allCases {

print(beverage)

}

// coffee

// tea

// juice

You can define Swift enumerations to store associated values of any given type, and the value types can be different for each case of the enumeration if needed. Enumerations similar to these are known as *discriminated unions*, *tagged unions*, or *variants* in other programming languages.

For example, suppose an inventory tracking system needs to track products by two different types of barcode. Some products are labeled with 1D barcodes in UPC format, which uses the numbers 0 to 9. Each barcode has a number system digit, followed by five manufacturer code digits and five product code digits. These are followed by a check digit to verify that the code has been scanned correctly:

Other products are labeled with 2D barcodes in QR code format, which can use any ISO 8859-1 character and can encode a string up to 2,953 characters long:

In Swift, an enumeration to define product barcodes of either type might look like this:

enum Barcode {

case upc(Int, Int, Int, Int)

case qrCode(String)

}

This can be read as:

“Define an enumeration type called Barcode, which can take either a value of upc with an associated value of type (Int, Int, Int, Int), or a value of qrCode with an associated value of type String.”

You can then create new barcodes using either type:

var productBarcode = Barcode.upc(8, 85909, 51226, 3)

## Raw Values

The barcode example in [Associated Values](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html#ID148) shows how cases of an enumeration can declare that they store associated values of different types. As an alternative to associated values, enumeration cases can come prepopulated with default values (called raw values), which are all of the same type.

Here’s an example that stores raw ASCII values alongside named enumeration cases:

enum ASCIIControlCharacter: Character {

case tab = "\t"

case lineFeed = "\n"

case carriageReturn = "\r"

}

Raw values can be strings, characters, or any of the integer or floating-point number types. Each raw value must be unique within its enumeration declaration.

NOTE

Raw values are *not* the same as associated values. Raw values are set to prepopulated values when you first define the enumeration in your code, like the three ASCII codes above. The raw value for a particular enumeration case is always the same. Associated values are set when you create a new constant or variable based on one of the enumeration’s cases, and can be different each time you do so.

### Implicitly Assigned Raw Values

When you’re working with enumerations that store integer or string raw values, you don’t have to explicitly assign a raw value for each case. When you don’t, Swift automatically assigns the values for you.

For example, when integers are used for raw values, the implicit value for each case is one more than the previous case. If the first case doesn’t have a value set, its value is 0.

The enumeration below is a refinement of the earlier Planet enumeration, with integer raw values to represent each planet’s order from the sun:

1. enum Planet: Int {
2. case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune
3. }

In the example above, Planet.mercury has an explicit raw value of 1, Planet.venus has an implicit raw value of 2, and so on.

When strings are used for raw values, the implicit value for each case is the text of that case’s name.

The enumeration below is a refinement of the earlier CompassPoint enumeration, with string raw values to represent each direction’s name:

1. enum CompassPoint: String {
2. case north, south, east, west
3. }

In the example above, CompassPoint.south has an implicit raw value of "south", and so on.

You access the raw value of an enumeration case with its rawValue property:

1. let earthsOrder = Planet.earth.rawValue
2. // earthsOrder is 3
3. let sunsetDirection = CompassPoint.west.rawValue
4. // sunsetDirection is "west"

### Initializing from a Raw Value

If you define an enumeration with a raw-value type, the enumeration automatically receives an initializer that takes a value of the raw value’s type (as a parameter called rawValue) and returns either an enumeration case or nil. You can use this initializer to try to create a new instance of the enumeration.

This example identifies Uranus from its raw value of 7:

1. let possiblePlanet = Planet(rawValue: 7)
2. // possiblePlanet is of type Planet? and equals Planet.uranus

Not all possible Int values will find a matching planet, however. Because of this, the raw value initializer always returns an optional enumeration case. In the example above, possiblePlanet is of type Planet?, or “optional Planet.”

NOTE

The raw value initializer is a failable initializer, because not every raw value will return an enumeration case. For more information, see [Failable Initializers](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID376).

If you try to find a planet with a position of 11, the optional Planet value returned by the raw value initializer will be nil:

1. let positionToFind = 11
2. if let somePlanet = Planet(rawValue: positionToFind) {
3. switch somePlanet {
4. case .earth:
5. print("Mostly harmless")
6. default:
7. print("Not a safe place for humans")
8. }
9. } else {
10. print("There isn't a planet at position \(positionToFind)")
11. }
12. // Prints "There isn't a planet at position 11"

This example uses optional binding to try to access a planet with a raw value of 11. The statement if let somePlanet = Planet(rawValue: 11) creates an optional Planet, and sets somePlanet to the value of that optional Planet if it can be retrieved. In this case, it isn’t possible to retrieve a planet with a position of 11, and so the else branch is executed instead.

## Recursive Enumerations

A recursive enumeration is an enumeration that has another instance of the enumeration as the associated value for one or more of the enumeration cases. You indicate that an enumeration case is recursive by writing indirect before it, which tells the compiler to insert the necessary layer of indirection.

For example, here is an enumeration that stores simple arithmetic expressions:

1. enum ArithmeticExpression {
2. case number(Int)
3. indirect case addition(ArithmeticExpression, ArithmeticExpression)
4. indirect case multiplication(ArithmeticExpression, ArithmeticExpression)
5. }

You can also write indirect before the beginning of the enumeration to enable indirection for all of the enumeration’s cases that have an associated value:

1. indirect enum ArithmeticExpression {
2. case number(Int)
3. case addition(ArithmeticExpression, ArithmeticExpression)
4. case multiplication(ArithmeticExpression, ArithmeticExpression)
5. }

This enumeration can store three kinds of arithmetic expressions: a plain number, the addition of two expressions, and the multiplication of two expressions. The addition and multiplication cases have associated values that are also arithmetic expressions—these associated values make it possible to nest expressions. For example, the expression (5 + 4) \* 2 has a number on the right-hand side of the multiplication and another expression on the left-hand side of the multiplication. Because the data is nested, the enumeration used to store the data also needs to support nesting—this means the enumeration needs to be recursive. The code below shows the ArithmeticExpression recursive enumeration being created for (5 + 4) \* 2:

1. let five = ArithmeticExpression.number(5)
2. let four = ArithmeticExpression.number(4)
3. let sum = ArithmeticExpression.addition(five, four)
4. let product = ArithmeticExpression.multiplication(sum, ArithmeticExpression.number(2))

A recursive function is a straightforward way to work with data that has a recursive structure. For example, here’s a function that evaluates an arithmetic expression:

func evaluate(\_ expression: ArithmeticExpression) -> Int {

switch expression {

case let .number(value):

return value

case let .addition(left, right):

return evaluate(left) + evaluate(right)

case let .multiplication(left, right):

return evaluate(left) \* evaluate(right)

}

}

print(evaluate(product))

// Prints "18"

This function evaluates a plain number by simply returning the associated value. It evaluates an addition or multiplication by evaluating the expression on the left-hand side, evaluating the expression on the right-hand side, and then adding them or multiplying them.

Comparing Structures and Classes

Structures and classes in Swift have many things in common. Both can:

* Define properties to store values
* Define methods to provide functionality
* Define subscripts to provide access to their values using subscript syntax
* Define initializers to set up their initial state
* Be extended to expand their functionality beyond a default implementation
* Conform to protocols to provide standard functionality of a certain kind

For more information, see [Properties](https://docs.swift.org/swift-book/LanguageGuide/Properties.html), [Methods](https://docs.swift.org/swift-book/LanguageGuide/Methods.html), [Subscripts](https://docs.swift.org/swift-book/LanguageGuide/Subscripts.html), [Initialization](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html), [Extensions](https://docs.swift.org/swift-book/LanguageGuide/Extensions.html), and [Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html).

Classes have additional capabilities that structures don’t have:

* Inheritance enables one class to inherit the characteristics of another.
* Type casting enables you to check and interpret the type of a class instance at runtime.
* Deinitializers enable an instance of a class to free up any resources it has assigned.
* Reference counting allows more than one reference to a class instance.

### Definition Syntax

Structures and classes have a similar definition syntax. You introduce structures with the struct keyword and classes with the class keyword. Both place their entire definition within a pair of braces:

struct SomeStructure {

// structure definition goes here

}

class SomeClass {

// class definition goes here

}

Here’s an example of a structure definition and a class definition:

struct Resolution {

var width = 0

var height = 0

}

class VideoMode {

var resolution = Resolution()

var interlaced = false

var frameRate = 0.0

var name: String?

}

## Structures and Enumerations Are Value Types

All structures and enumerations are value types in Swift. This means that any structure and enumeration instances you create—and any value types they have as properties—are always copied when they are passed around in your code.

NOTE

Collections defined by the standard library like arrays, dictionaries, and strings use an optimization to reduce the performance cost of copying. Instead of making a copy immediately, these collections share the memory where the elements are stored between the original instance and any copies. If one of the copies of the collection is modified, the elements are copied just before the modification. The behavior you see in your code is always as if a copy took place immediately.

Consider this example, which uses the Resolution structure from the previous example:

1. let hd = Resolution(width: 1920, height: 1080)
2. var cinema = hd

This example declares a constant called hd and sets it to a Resolution instance initialized with the width and height of full HD video (1920 pixels wide by 1080 pixels high).

It then declares a variable called cinema and sets it to the current value of hd. Because Resolution is a structure, a *copy* of the existing instance is made, and this new copy is assigned to cinema. Even though hd and cinema now have the same width and height, they are two completely different instances behind the scenes.

Next, the width property of cinema is amended to be the width of the slightly wider 2K standard used for digital cinema projection (2048 pixels wide and 1080 pixels high):

1. cinema.width = 2048

Checking the width property of cinema shows that it has indeed changed to be 2048:

1. print("cinema is now \(cinema.width) pixels wide")
2. // Prints "cinema is now 2048 pixels wide"

However, the width property of the original hd instance still has the old value of 1920:

1. print("hd is still \(hd.width) pixels wide")
2. // Prints "hd is still 1920 pixels wide"

When cinema was given the current value of hd, the *values* stored in hd were copied into the new cinema instance. The end result was two completely separate instances that contained the same numeric values. However, because they are separate instances, setting the width

The same behavior applies to enumerations:

1. enum CompassPoint {
2. case north, south, east, west
3. mutating func turnNorth() {
4. self = .north
5. }
6. }
7. var currentDirection = CompassPoint.west
8. let rememberedDirection = currentDirection
9. currentDirection.turnNorth()
10. print("The current direction is \(currentDirection)")
11. print("The remembered direction is \(rememberedDirection)")
12. // Prints "The current direction is north"
13. // Prints "The remembered direction is west"

When rememberedDirection is assigned the value of currentDirection, it’s actually set to a copy of that value. Changing the value of currentDirection thereafter doesn’t affect the copy of the original value that was stored in rememberedDirection.

Classes Are Reference Types

Unlike value types, *reference types* are *not* copied when they are assigned to a variable or constant, or when they are passed to a function. Rather than a copy, a reference to the same existing instance is used.

Here’s an example, using the VideoMode class defined above:

1. let tenEighty = VideoMode()
2. tenEighty.resolution = hd
3. tenEighty.interlaced = true
4. tenEighty.name = "1080i"
5. tenEighty.frameRate = 25.0

This example declares a new constant called tenEighty and sets it to refer to a new instance of the VideoMode class. The video mode is assigned a copy of the HD resolution of 1920 by 1080 from before. It’s set to be interlaced, its name is set to "1080i", and its frame rate is set to 25.0 frames per second.

Next, tenEighty is assigned to a new constant, called alsoTenEighty, and the frame rate of alsoTenEighty is modified:

let alsoTenEighty = tenEighty

alsoTenEighty.frameRate = 30.0

print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")

// Prints "The frameRate property of tenEighty is now 30.0"

### Identity Operators

Because classes are reference types, it’s possible for multiple constants and variables to refer to the same single instance of a class behind the scenes. (The same isn’t true for structures and enumerations, because they are always copied when they are assigned to a constant or variable, or passed to a function.)

It can sometimes be useful to find out whether two constants or variables refer to exactly the same instance of a class. To enable this, Swift provides two identity operators:

* Identical to (===)
* Not identical to (!==)

Use these operators to check whether two constants or variables refer to the same single instance:

1. if tenEighty === alsoTenEighty {
2. print("tenEighty and alsoTenEighty refer to the same VideoMode instance.")
3. }
4. // Prints "tenEighty and alsoTenEighty refer to the same VideoMode instance."

Note that identical to (represented by three equals signs, or ===) doesn’t mean the same thing as equal to (represented by two equals signs, or ==). Identical to means that two constants or variables of class type refer to exactly the same class instance. Equal to means that two instances are considered equal or equivalent in value, for some appropriate meaning of equal, as defined by the type’s designer.

When you define your own custom structures and classes, it’s your responsibility to decide what qualifies as two instances being equal. The process of defining your own implementations of the == and != operators is described in [Equivalence Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID45).

# Properties

Properties associate values with a particular class, structure, or enumeration. Stored properties store constant and variable values as part of an instance, whereas computed properties calculate (rather than store) a value. Computed properties are provided by classes, structures, and enumerations. Stored properties are provided only by classes and structures.

Stored and computed properties are usually associated with instances of a particular type. However, properties can also be associated with the type itself. Such properties are known as type properties.

In addition, you can define property observers to monitor changes in a property’s value, which you can respond to with custom actions. Property observers can be added to stored properties you define yourself, and also to properties that a subclass inherits from its superclass.

You can also use a property wrapper to reuse code in the getter and setter of multiple properties.

## Stored Properties

In its simplest form, a stored property is a constant or variable that is stored as part of an instance of a particular class or structure. Stored properties can be either variable stored properties (introduced by the var keyword) or constant stored properties (introduced by the let keyword).

You can provide a default value for a stored property as part of its definition, as described in [Default Property Values](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID206). You can also set and modify the initial value for a stored property during initialization. This is true even for constant stored properties, as described in [Assigning Constant Properties During Initialization](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID212).

The example below defines a structure called FixedLengthRange, which describes a range of integers whose range length cannot be changed after it is created:

1. struct FixedLengthRange {
2. var firstValue: Int
3. let length: Int
4. }
5. var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
6. // the range represents integer values 0, 1, and 2
7. rangeOfThreeItems.firstValue = 6
8. // the range now represents integer values 6, 7, and 8

Instances of FixedLengthRange have a variable stored property called firstValue and a constant stored property called length. In the example above, length is initialized when the new range is created and cannot be changed thereafter, because it is a constant property.

### Stored Properties of Constant Structure Instances

If you create an instance of a structure and assign that instance to a constant, you cannot modify the instance’s properties, even if they were declared as variable properties:

1. let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
2. // this range represents integer values 0, 1, 2, and 3
3. rangeOfFourItems.firstValue = 6
4. // this will report an error, even though firstValue is a variable property

Because rangeOfFourItems is declared as a constant (with the let keyword), it is not possible to change its firstValue property, even though firstValue is a variable property.

This behavior is due to structures being value types. When an instance of a value type is marked as a constant, so are all of its properties.

The same is not true for classes, which are reference types. If you assign an instance of a reference type to a constant, you can still change that instance’s variable properties.

### Lazy Stored Properties

A lazy stored property is a property whose initial value is not calculated until the first time it is used. You indicate a lazy stored property by writing the lazy modifier before its declaration.

NOTE

You must always declare a lazy property as a variable (with the var keyword), because its initial value might not be retrieved until after instance initialization completes. Constant properties must always have a value before initialization completes, and therefore cannot be declared as lazy.

Lazy properties are useful when the initial value for a property is dependent on outside factors whose values are not known until after an instance’s initialization is complete. Lazy properties are also useful when the initial value for a property requires complex or computationally expensive setup that should not be performed unless or until it is needed.

The example below uses a lazy stored property to avoid unnecessary initialization of a complex class. This example defines two classes called DataImporter and DataManager, neither of which is shown in full:

1. class DataImporter {

/\*

DataImporter is a class to import data from an external file.

The class is assumed to take a nontrivial amount of time to initialize.

\*/

var filename = "data.txt"

// the DataImporter class would provide data importing functionality here

}

class DataManager {

lazy var importer = DataImporter()

var data = [String]()

// the DataManager class would provide data management functionality here

}

let manager = DataManager()

manager.data.append("Some data")

manager.data.append("Some more data")

// the DataImporter instance for the importer property has not yet been created

The DataManager class has a stored property called data, which is initialized with a new, empty array of String values. Although the rest of its functionality is not shown, the purpose of this DataManager class is to manage and provide access to this array of String data.

Part of the functionality of the DataManager class is the ability to import data from a file. This functionality is provided by the DataImporter class, which is assumed to take a nontrivial amount of time to initialize. This might be because a DataImporter instance needs to open a file and read its contents into memory when the DataImporter instance is initialized.

It is possible for a DataManager instance to manage its data without ever importing data from a file, so there is no need to create a new DataImporter instance when the DataManager itself is created. Instead, it makes more sense to create the DataImporter instance if and when it is first used.

Because it is marked with the lazy modifier, the DataImporter instance for the importer property is only created when the importer property is first accessed, such as when its filename property is queried:

print(manager.importer.filename)

// the DataImporter instance for the importer property has now been created

// Prints "data.txt"

NOTE

If a property marked with the lazy modifier is accessed by multiple threads simultaneously and the property has not yet been initialized, there is no guarantee that the property will be initialized only once.

### Stored Properties and Instance Variables

If you have experience with Objective-C, you may know that it provides two ways to store values and references as part of a class instance. In addition to properties, you can use instance variables as a backing store for the values stored in a property.

Swift unifies these concepts into a single property declaration. A Swift property does not have a corresponding instance variable, and the backing store for a property is not accessed directly. This approach avoids confusion about how the value is accessed in different contexts and simplifies the property’s declaration into a single, definitive statement. All information about the property—including its name, type, and memory management characteristics—is defined in a single location as part of the type’s definition.

## Computed Properties

In addition to stored properties, classes, structures, and enumerations can define computed properties, which do not actually store a value. Instead, they provide a getter and an optional setter to retrieve and set other properties and values indirectly.

struct Point {

var x = 0.0, y = 0.0

}

struct Size {

var width = 0.0, height = 0.0

}

struct Rect {

var origin = Point()

var size = Size()

var center: Point {

get {

let centerX = origin.x + (size.width / 2)

let centerY = origin.y + (size.height / 2)

return Point(x: centerX, y: centerY)

}

set(newCenter) {

origin.x = newCenter.x - (size.width / 2)

origin.y = newCenter.y - (size.height / 2)

}

}

}

var square = Rect(origin: Point(x: 0.0, y: 0.0),

size: Size(width: 10.0, height: 10.0))

let initialSquareCenter = square.center

square.center = Point(x: 15.0, y: 15.0)

print("square.origin is now at (\(square.origin.x), \(square.origin.y))")

// Prints "square.origin is now at (10.0, 10.0)"

### Shorthand Setter Declaration

If a computed property’s setter doesn’t define a name for the new value to be set, a default name of newValue is used. Here’s an alternative version of the Rect structure that takes advantage of this shorthand notation:

1. struct AlternativeRect {
2. var origin = Point()
3. var size = Size()
4. var center: Point {
5. get {
6. let centerX = origin.x + (size.width / 2)
7. let centerY = origin.y + (size.height / 2)
8. return Point(x: centerX, y: centerY)
9. }
10. set {
11. origin.x = newValue.x - (size.width / 2)
12. origin.y = newValue.y - (size.height / 2)
13. }
14. }
15. }

### Shorthand Getter Declaration

If the entire body of a getter is a single expression, the getter implicitly returns that expression. Here’s an another version of the Rect structure that takes advantage of this shorthand notation and the shorthand notation for setters:

struct CompactRect {

var origin = Point()

var size = Size()

var center: Point {

get {

Point(x: origin.x + (size.width / 2),

y: origin.y + (size.height / 2))

}

set {

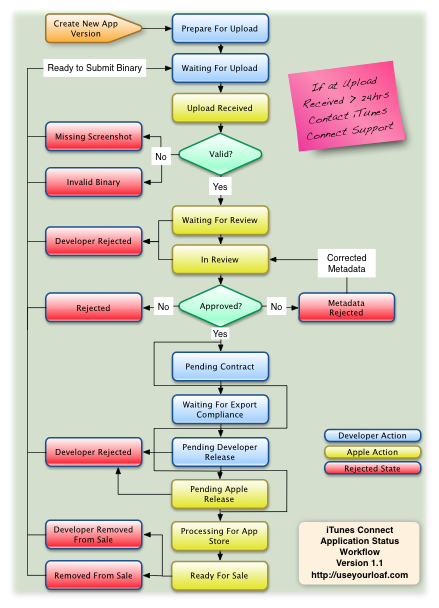
origin.x = newValue.x - (size.width / 2)

origin.y = newValue.y - (size.height / 2)

}

}

}



### Uploading

The process starts with the creation of a new app or a new version of an existing app. At this point whilst you are entering the app metadata, uploading screenshots and setting pricing the app status is Prepare For Upload. Once you hit the ready to submit binary button the app status will change to Waiting For Upload. Shortly after uploading the binary the status should change to Upload Received.

At this point it is possible your app can be rejected for a Missing Screenshot or an Invalid Binary. Common mistakes include forgetting to include an iPad screenshot for a universal app or submitting a binary without incrementing the bundle version number. If you are validating the app within Xcode you should be able to avoid these mistakes. Once you have resolved the issue (check your email for details) by editing the metadata to add the missing screenshots or uploaded a valid binary you should reach the Waiting For Review status.

If you are stuck in the Upload Received state for more than 24 hours you should contact iTunes Connect Support for help.

### Reviewing

The App Store submission process seems to have improved greatly over the last year with 98% of new apps and 99% of app updates now reviewed within 7 days. You can generally expect to spend a few days in the Waiting For Review state and then a few more days actually In Review. If your app is Rejected you should get an email notification indicating the reasons or you can check the Resolution Center. Once you have fixed the problems you submit the revised binary and start the process again.

Apple has recently introduced a new rejection status for when you have non-compliant metadata. For example a number of developers are currently reporting rejections because the app description mentions iOS 5 which at this time is still pre-release software and hence covered by the Apple NDA. If your app ends up in the Metadata Rejected state you need to use iTunes Connect Resolution Center to review the reason and edit the offending metadata item. Since you are only changing metadata you do not need to submit a new binary which should save some time.

Whilst you are waiting for Apple (Waiting For Review or In Review) you can abort the process by rejecting the binary using iTunes Connect. Your app status will change to Developer Rejected and you will need to restart from the beginning with a new binary.

### Waiting to Go Live

Once you get past the review process there are sill a number of prerequisites that can delay the app from reaching the App Store. The first three states all require developer action to resolve:

* Pending Contract: You have not yet completed the contracts in the Contracts, Tax & Banking section of iTunes Connect.
* Waiting For Export Compliance: If you have an app with encryption subject to export compliance you need to wait until your CCATS has been approved.
* Pending Developer Release: The version release control in iTunes Connect allows you to set whether an app update is automatically released once it is approved or if the developer will manually release it. You will get a reminder email from Apple if you have an App in this state for more than 30 days. You can also reject the app at this point if you do not want it to ever be released.

There is a final pending status that depends on Apple:

* Pending Apple Release: If you have submitted an app for an upcoming iOS release your app will stay in this state until the corresponding iOS release is made public. For example we can expect that Apple will shortly invite developers to submit apps ahead of the public release of iOS 5. You can also reject the app whilst waiting in this state.

### On Sale

The final stage happens quickly once you have the approvals and all prerequisites in place. The Processing For App Store status should quickly become Ready For Sale and then within 24 hours the app should appear in the country specific app stores that you specified when submitting the app. Once the app is on sale the developer can decide to remove it from sale (Developer Removed From Sale) or Apple can identify a problem and decide to remove it from the store (Removed From Sale).

[**Pending Developer Release**](https://developer.apple.com/library/content/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/ChangingAppStatus.html)  
Your app version has been approved by Apple and is waiting on you to release it. Release the version to the store when you’re ready.

# Delete Your App From The Apple App Store

**Note:** Once you delete your app from the Apple App Store, you will not be able to reuse the app name or SKU, and you will not be able to restore your app.

An alternative to deleting your app is simply removing it from sale on the Apple App Store. Once it is removed from sale, your app can no longer be downloaded from the Apple App Store. However, your app will remain in your iTunes connect account and you have the option in the future to update your app or restore it to the app store for sale. To remove your app from the Apple App Store, without deleting it, follow **only** steps 1-6 below.

**No**, once and App has been deleted, either by you or because of the expiration process, you cannot reuse your SKU or app name in the same account again. It can be used, though, by another developer.

Under the [Deleting an App](http://developer.apple.com/library/ios/#documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/10_ManagingYourApplications/ManagingYourApplications.html#//apple_ref/doc/uid/TP40011225-CH3-SW37) section of the iTunes Connect Developer Guide you'll learn more about the ramifications of an App Deletion.

Please see **Removing an App From Sale** (page 92) section on [iTunes Connect Developer Guide 6.3](https://itunesconnect.apple.com/docs/iTunesConnect_DeveloperGuide.pdf)

1. Click "Rights and Pricing"
2. Go to the sale territories section. There is a link near the end
3. Click on the "Deselect all" button to uncheck all App Store territories.
4. Click on the "Save Changes" button.

After removing all assigned territory checkboxes from your app in the Rights and Pricing section,, the status changes to Developer Removed from Sale and your app will not be seen on the App Store within 24 hours.

What you need to do is this.

1. Go to “Manage Your Applications” and select the app.
2. Click “Rights and Pricing” (blue button at the top right.
3. Below the availability date and price tier section, you should see a grid of checkboxes for the various countries your app is available in. Click the blue “Deselect All” button.
4. Click “Save Changes” at the bottom.

Your app's state will then be “Developer Removed From Sale”, and it will no longer be available on the App Store in any country.

**MANAGE APPS AND VERSIONS**

# Remove an app

To remove an app from your main view in My Apps, it must be first [removed from the App Store](https://help.apple.com/app-store-connect/#/dev7013b314c) and any associated [in-app purchases must be removed from sale](https://help.apple.com/app-store-connect/#/dev360aba524).

Additionally, you can only remove apps if all versions of that app are in one of the following states:

Prepare for Submission

Invalid Binary

Developer Rejected

Rejected

Metadata Rejected

Developer Removed from Sale

Removed from Sale

Apps can’t be removed if they are part of a [Game Center](https://help.apple.com/app-store-connect/#/dev8df418347) group, in an [app bundle](https://help.apple.com/app-store-connect/#/dev1c521d494), currently available on the App Store, associated with in-app purchases available for sale, or if they're being actively transferred to another account.

If the app is part of a Game Center group or app bundle and hasn't been approved yet, it must be removed from the group or bundle before it can be removed from your main view in My Apps. App bundles and approved apps that are part of a Game Center group cannot be removed.

**WARNING:** If you remove an app, you will lose ownership of the app name. Removed apps can only be restored if the name is not currently in use by another developer account. In addition, the [SKU](https://help.apple.com/app-store-connect/#/devb2675b955) can’t be reused in the same organization and if you’ve uploaded a build, your [bundle ID](https://help.apple.com/app-store-connect/#/deve8607d6eb) can’t be reused.

**Required role:** [Account Holder](https://help.apple.com/app-store-connect/#/dev1d7b9bccf), [Admin](https://help.apple.com/app-store-connect/#/devf3ef59c83). See [Role permissions](https://help.apple.com/app-store-connect/#/deve5f9a89d7).

## Remove an app

1. From the App Store Connect [homepage](https://itunesconnect.apple.com/), click My Apps, then choose the app you want to remove.
2. Scroll to the Additional Information section, then click Remove App. If you don’t see Remove App, make sure you are an Account Holder or Admin and that your app is in a state which allows removal.
3. In the dialog that appears, click Remove.

## Restore an app

1. From the [homepage](https://itunesconnect.apple.com/), click My Apps.
2. In the upper right-hand corner, click the arrow next to All Statuses.
3. From the drop-down menu, choose Removed Apps.
4. Choose the app you want to restore.
5. Scroll to the Additional Information section, then click Restore App.

If the app name is no longer available, the app cannot be restored. If you believe an app is preventing you from using your tradema

<https://help.apple.com/app-store-connect/#/dev28d17ed35>

SAP stands for **S**ystems **A**pplications and **P**roducts in Data Processing.

SAP system consists of a number of fully integrated modules, which covers virtually every aspect of business management.

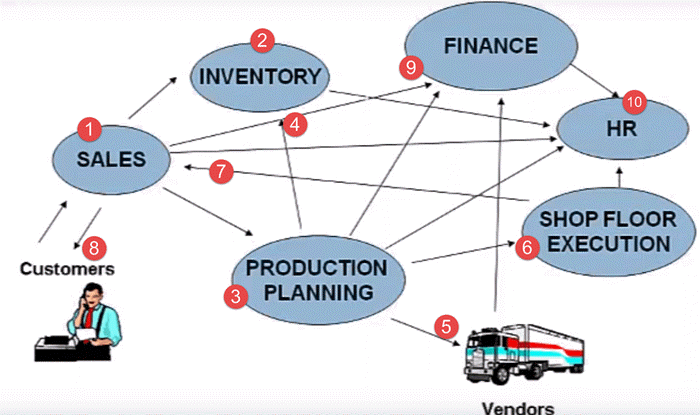
SAP is #1 in the ERP market. As of 2010, SAP has more  than 140,000 installations worldwide, over 25 industry-specific business solutions and more than 75,000 customers in 120 countries

Other Competitive products of SAP Software in the market are  Oracle, Microsoft Dynamics, etc.

why Enterprise Resource Planning also called ERP, is required?

Suppose a client approaches a sales team asking for a particular product. The sales team contacts the inventory department to check the availability of the product. To their surprise, the sales team found out that the product is out of stock. So next time this doesn’t happen, they have to introduce an SAP ERP tool.

Before we actually see in detail, what ERP is and how ERP can help in your business process, we will understand how different departments are involved in the whole business process, right from the ordering of the raw material – to manufacturing goods – to delivering final products to the customer.



**Here is the whole process that is followed by any business unit.**

1. Client contacts the sales team to check the availability of the product
2. Sales team approaches the Inventory department to check for the availability of the product
3. In case the product is out of stock, the sales team approaches the Production Planning Department to manufacture the product
4. The production planning team checks with inventory department for availability of raw material
5. If the raw material is not available with inventory, the Production Planning team buys the raw material from the Vendors
6. Then Production Planning forwards the raw materials to the Shop Floor Execution for actual production
7. Once ready, the Shop Floor Team sends the goods to the Sales Team
8. Sales Team who in turn deliver it to the client
9. The sales team updates the finance with revenue generated by the sale of the product. Production planning team update the finance with payments to be made to different vendors for raw materials.
10. All departments approach the HR for any Human Resource related issue.

That is a typical business process for any manufacturing company. Some key inferences one could derive from the scenario would be.

* It has many departments or business units
* These departments or business units continuously communicate and exchange data with each other
* The success of any organization lies in effective communication, and data exchange, within these departments, as well as associated third party such as vendors, outsourcers, and customers.

Based on the manner in which communication and data exchanged is managed. Enterprise systems can be broadly classified as

**1)**[**Decentralized System**](https://www.guru99.com/what-is-sap-definition-of-sap-erp-software.html#1)

**2)**[**Centralized System which is also called as ERP.**](https://www.guru99.com/what-is-sap-definition-of-sap-erp-software.html#2)

* **Data is maintained locally at the individual departments**
* **Departments do not have access to information or data from other departments**

To identify issues arising due to decentralized Enterprise management system lets look at the same business process again. The customer approaches the sales team for a product, but this time around, he needs the product, on an urgent basis.

Since it is a decentralized process, the Sales Team do not have any real-time information access to the productavailability. So they approach the Inventory department to check the availability of the product. This process takes time, and the customer chooses another vendor leading to loss of revenue and customer dissatisfaction.

Now, suppose the product is out of stock, and the Sales Team approaches the Production Planning team to manufacture the product for future use. Production Planning Team checks the availability of the raw materials required.

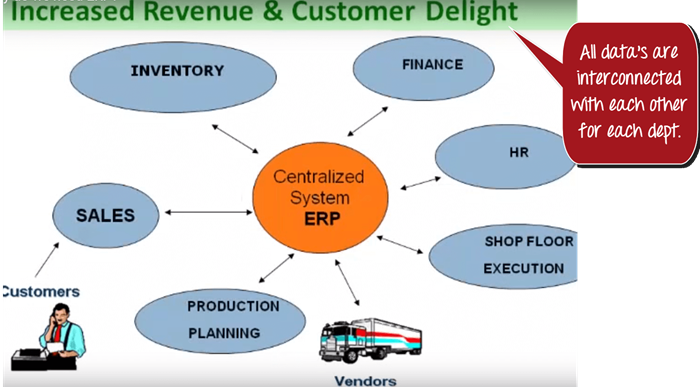
In a decentralized system, raw material information is stored separately by Production Planning as well as the Inventory Department. Thus, data maintenance cost (in this case, Raw Material) goes up.

## Centralized System

In a company, with a Centralized System of Information and Data Management.

**1) Data is maintained at a central location and is shared with various Departments**

**2) Departments have access to information or data from other Departments**



In this case, all departments update a Central Information System.

* When Customer approaches the sales team to buy a product on an urgent basis. The Sales Team has real-time information access to the products in inventory which is updated by the Inventory Department in the Centralized System
* Sales Team respond to customer request on time, leading to Increased Revenue and Customer Delight.
* In case, manufacturing is required the Sales Team update the Centralized Database, so that all the department remain informed about the product status.
* Production Planning Department is **auto updated** by the Centralized Database for requirements. Production Planning Team checks the availability of the raw materials required via the Central Database, which is updated by the Inventory Department.
* Thus, Data Duplication is avoided, and accurate data is made available. The Shop Floor Team update their Man Power Status regularly in the Central Database, which can be accessed by the HR department.
* In case of shortage of workforce, HR team starts the recruitment process with considerable lead time to hire a suitable candidate at market price. Thus labor cost goes down.
* While vendors can directly submit their invoices to the Central Enterprise System, which can be accessed by the finance department. Thus, payments are made on time, and possible legal actions are avoided
* SAP software is a type of Centralized System. SAP Systems are the most popularly used in ERP software.

### Key benefits of the centralized system are:

* It eliminates the duplication, discontinuity, and redundancy in data
* Provides information across departments in real time.
* SAP System provides control over various business processes
* Increases productivity, better inventory management, promotes quality, reduced material cost, effective human resources management, reduced overheads boosts profits
* Better customer interaction and increased throughput. It also improves customer service
* Hence, a centralized enterprise management system is required.
* SAP Software is a centralized enterprise management system, also known as Enterprise Resource Planning.
* The meaning of the acronym SAP is **S**ystems **A**pplications and **P**roducts in Data Processing

## SAP Web IDE

Put the power of agile development in your developers’ hands by streamlining the end-to-end application lifecycle and extending your digital core with smarter, more intelligent applications.

### Application innovation and extension

Apply SAP Cloud Application Programming Model to develop full-stack applications and extend services for SAP S/4HANA Cloud.

### Mobile app development

Create, package, and deploy native apps with no-code tools or an SAP Fiori mobile app based on Apache Cordova and Kapsel.

### Creation and extension of SAP Fiori

Use code or a graphical editor to apply extension points for SAP Fiori and build mobile apps with development tools.

### Enriched user experiences

Use a software development kit to add application templates and capabilities or integrate them with external tools.

SAP Cloud Platform Mobile Services provides extensions for SAP Web IDE, so you can build end-to end-mobile applications. The Mobile Services App Development Tool allows you to model and generate OData services, create mobile native apps from metadata, and build content for SAP Mobile Cards. In addition, with the SAP Cloud Platform MDK and SDKs for iOS and Android, you can create and deploy native and hybrid mobile apps.

### Read-Only Computed Properties

A computed property with a getter but no setter is known as a read-only computed property. A read-only computed property always returns a value, and can be accessed through dot syntax, but cannot be set to a different value.

NOTE

You must declare computed properties—including read-only computed properties—as variable properties with the var keyword, because their value is not fixed. The let keyword is only used for constant properties, to indicate that their values cannot be changed once they are set as part of instance initialization.

You can simplify the declaration of a read-only computed property by removing the get keyword and its braces:

struct Cuboid {

var width = 0.0, height = 0.0, depth = 0.0

var volume: Double {

return width \* height \* depth

}

}

let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)

print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")

// Prints "the volume of fourByFiveByTwo is 40.0"

This example defines a new structure called Cuboid, which represents a 3D rectangular box with width, height, and depth properties. This structure also has a read-only computed property called volume, which calculates and returns the current volume of the cuboid. It doesn’t make sense for volume to be settable, because it would be ambiguous as to which values of width, height, and depth should be used for a particular volume value. Nonetheless, it is useful for a Cuboid to provide a read-only computed property to enable external users to discover its current calculated volume.

## Property Observers

Property observers observe and respond to changes in a property’s value. Property observers are called every time a property’s value is set, even if the new value is the same as the property’s current value.

You can add property observers to any stored properties you define, except for lazy stored properties. You can also add property observers to any inherited property (whether stored or computed) by overriding the property within a subclass. You don’t need to define property observers for nonoverridden computed properties, because you can observe and respond to changes to their value in the computed property’s setter. Property overriding is described in [Overriding](https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html#ID196).

You have the option to define either or both of these observers on a property:

* willSet is called just before the value is stored.
* didSet is called immediately after the new value is stored.

## Property Observers

Property observers observe and respond to changes in a property’s value. Property observers are called every time a property’s value is set, even if the new value is the same as the property’s current value.

You can add property observers to any stored properties you define, except for lazy stored properties. You can also add property observers to any inherited property (whether stored or computed) by overriding the property within a subclass. You don’t need to define property observers for nonoverridden computed properties, because you can observe and respond to changes to their value in the computed property’s setter. Property overriding is described in [Overriding](https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html#ID196).

You have the option to define either or both of these observers on a property:

* willSet is called just before the value is stored.
* didSet is called immediately after the new value is stored.

If you implement a willSet observer, it’s passed the new property value as a constant parameter. You can specify a name for this parameter as part of your willSet implementation. If you don’t write the parameter name and parentheses within your implementation, the parameter is made available with a default parameter name of newValue.

Similarly, if you implement a didSet observer, it’s passed a constant parameter containing the old property value. You can name the parameter or use the default parameter name of oldValue. If you assign a value to a property within its own didSet observer, the new value that you assign replaces the one that was just set.

NOTE

The willSet and didSet observers of superclass properties are called when a property is set in a subclass initializer, after the superclass initializer has been called. They are not called while a class is setting its own properties, before the superclass initializer has been called.

For more information about initializer delegation, see [Initializer Delegation for Value Types](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID215) and [Initializer Delegation for Class Types](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID219).

Here’s an example of willSet and didSet in action. The example below defines a new class called StepCounter, which tracks the total number of steps that a person takes while walking. This class might be used with input data from a pedometer or other step counter to keep track of a person’s exercise during their daily routine.

1. class StepCounter {
2. var totalSteps: Int = 0 {
3. willSet(newTotalSteps) {
4. print("About to set totalSteps to \(newTotalSteps)")
5. }
6. didSet {
7. if totalSteps > oldValue {
8. print("Added \(totalSteps - oldValue) steps")
9. }
10. }
11. }
12. }
13. let stepCounter = StepCounter()
14. stepCounter.totalSteps = 200
15. // About to set totalSteps to 200
16. // Added 200 steps
17. stepCounter.totalSteps = 360
18. // About to set totalSteps to 360
19. // Added 160 steps
20. stepCounter.totalSteps = 896
21. // About to set totalSteps to 896
22. // Added 536 steps

The StepCounter class declares a totalSteps property of type Int. This is a stored property with willSet and didSet observers.

The willSet and didSet observers for totalSteps are called whenever the property is assigned a new value. This is true even if the new value is the same as the current value.

This example’s willSet observer uses a custom parameter name of newTotalSteps for the upcoming new value. In this example, it simply prints out the value that is about to be set.

The didSet observer is called after the value of totalSteps is updated. It compares the new value of totalSteps against the old value. If the total number of steps has increased, a message is printed to indicate how many new steps have been taken. The didSet observer does not provide a custom parameter name for the old value, and the default name of oldValue is used instead.

NOTE

If you pass a property that has observers to a function as an in-out parameter, the willSet and didSet observers are always called. This is because of the copy-in copy-out memory model for in-out parameters: The value is always written back to the property at the end of the function. For a detailed discussion of the behavior of in-out parameters, see [In-Out Parameters](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID545).

## Property Wrappers

A property wrapper adds a layer of separation between code that manages how a property is stored and the code that defines a property. For example, if you have properties that provide thread-safety checks or store their underlying data in a database, you have to write that code on every property. When you use a property wrapper, you write the management code once when you define the wrapper, and then reuse that management code by applying it to multiple properties.

To define a property wrapper, you make a structure, enumeration, or class that defines a wrappedValue property. In the code below, the TwelveOrLess structure ensures that the value it wraps always contains a number less than or equal to 12. If you ask it to store a larger number, it stores 12 instead.

1. @propertyWrapper
2. struct TwelveOrLess {
3. private var number = 0
4. var wrappedValue: Int {
5. get { return number }
6. set { number = min(newValue, 12) }
7. }
8. }

The setter ensures that new values are less than 12, and the getter returns the stored value.

NOTE

The declaration for number in the example above marks the variable as private, which ensures number is used only in the implementation of TwelveOrLess. Code that’s written anywhere else accesses the value using the getter and setter for wrappedValue, and can’t use number directly. For information about private, see [Access Control](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html).

You apply a wrapper to a property by writing the wrapper’s name before the property as an attribute. Here’s a structure that stores a small rectangle, using the same (rather arbitrary) definition of “small” that’s implemented by the TwelveOrLess property wrapper:

1. struct SmallRectangle {
2. @TwelveOrLess var height: Int
3. @TwelveOrLess var width: Int
4. }
5. var rectangle = SmallRectangle()
6. print(rectangle.height)
7. // Prints "0"
8. rectangle.height = 10
9. print(rectangle.height)
10. // Prints "10"
11. rectangle.height = 24
12. print(rectangle.height)
13. // Prints "12"

The height and width properties get their initial values from the definition of TwelveOrLess, which sets TwelveOrLess.number to zero. Storing the number 10 into rectangle.height succeeds because it’s a small number. Trying to store 24 actually stores a value of 12 instead, because 24 is too large for the property setter’s rule.

When you apply a wrapper to a property, the compiler synthesizes code that provides storage for the wrapper and code that provides access to the property through the wrapper. (The property wrapper is responsible for storing the wrapped value, so there’s no synthesized code for that.) You could write code that uses the behavior of a property wrapper, without taking advantage of the special attribute syntax. For example, here’s a version of SmallRectangle from the previous code listing that wraps its properties in the TwelveOrLess structure explicitly, instead of writing @TwelveOrLess as an attribute:

1. struct SmallRectangle {
2. private var \_height = TwelveOrLess()
3. private var \_width = TwelveOrLess()
4. var height: Int {
5. get { return \_height.wrappedValue }
6. set { \_height.wrappedValue = newValue }
7. }
8. var width: Int {
9. get { return \_width.wrappedValue }
10. set { \_width.wrappedValue = newValue }
11. }
12. }

The \_height and \_width properties store an instance of the property wrapper, TwelveOrLess. The getter and setter for height and width wrap access to the wrappedValue property.

### Setting Initial Values for Wrapped Properties

The code in the examples above sets the initial value for the wrapped property by giving number an initial value in the definition of TwelveOrLess. Code that uses this property wrapper, can’t specify a different initial value for a property that’s wrapped by TwelveOrLess—for example, the definition of SmallRectangle can’t give height or width initial values. To support setting an initial value or other customization, the property wrapper needs to add an initializer. Here’s an expanded version of TwelveOrLess called SmallNumber that defines initializers that set the wrapped and maximum value:

1. @propertyWrapper
2. struct SmallNumber {
3. private var maximum: Int
4. private var number: Int
5. var wrappedValue: Int {
6. get { return number }
7. set { number = min(newValue, maximum) }
8. }
9. init() {
10. maximum = 12
11. number = 0
12. }
13. init(wrappedValue: Int) {
14. maximum = 12
15. number = min(wrappedValue, maximum)
16. }
17. init(wrappedValue: Int, maximum: Int) {
18. self.maximum = maximum
19. number = min(wrappedValue, maximum)
20. }
21. }

The definition of SmallNumber includes three initializers—init(), init(wrappedValue:), and init(wrappedValue:maximum:)—which the examples below use to set the wrapped value and the maximum value. For information about initialization and initializer syntax, see [Initialization](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html).

When you apply a wrapper to a property and you don’t specify an initial value, Swift uses the init() initializer to set up the wrapper. For example:

1. struct ZeroRectangle {
2. @SmallNumber var height: Int
3. @SmallNumber var width: Int
4. }
5. var zeroRectangle = ZeroRectangle()
6. print(zeroRectangle.height, zeroRectangle.width)
7. // Prints "0 0"

The instances of SmallNumber that wrap height and width are created by calling SmallNumber(). The code inside that initializer sets the initial wrapped value and the initial maximum value, using the default values of zero and 12. The property wrapper still provides all of the initial values, like the earlier example that used TwelveOrLess in SmallRectangle. Unlike that example, SmallNumber also supports writing those initial values as part of declaring the property.

When you specify an initial value for the property, Swift uses the init(wrappedValue:) initializer to set up the wrapper. For example:

1. struct UnitRectangle {
2. @SmallNumber var height: Int = 1
3. @SmallNumber var width: Int = 1
4. }
5. var unitRectangle = UnitRectangle()
6. print(unitRectangle.height, unitRectangle.width)
7. // Prints "1 1"

When you write = 1 on a property with a wrapper, that’s translated into a call to the init(wrappedValue:) initializer. The instances of SmallNumber that wrap height and width are created by calling SmallNumber(wrappedValue: 1). The initializer uses the wrapped value that’s specified here, and it uses the default maximum value of 12.

When you write arguments in parentheses after the custom attribute, Swift uses the initializer that accepts those arguments to set up the wrapper. For example, if you provide an initial value and a maximum value, Swift uses the init(wrappedValue:maximum:) initializer:

1. struct NarrowRectangle {
2. @SmallNumber(wrappedValue: 2, maximum: 5) var height: Int
3. @SmallNumber(wrappedValue: 3, maximum: 4) var width: Int
4. }
5. var narrowRectangle = NarrowRectangle()
6. print(narrowRectangle.height, narrowRectangle.width)
7. // Prints "2 3"
8. narrowRectangle.height = 100
9. narrowRectangle.width = 100
10. print(narrowRectangle.height, narrowRectangle.width)
11. // Prints "5 4"

The instance of SmallNumber that wraps height is created by calling SmallNumber(wrappedValue: 2, maximum: 5), and the instance that wraps width is created by calling SmallNumber(wrappedValue: 3, maximum: 4).

By including arguments to the property wrapper, you can set up the initial state in the wrapper or pass other options to the wrapper when it’s created. This syntax is the most general way to use a property wrapper. You can provide whatever arguments you need to the attribute, and they’re passed to the initializer.

When you include property wrapper arguments, you can also specify an initial value using assignment. Swift treats the assignment like a wrappedValue argument and uses the initializer that accepts the arguments you include. For example:

1. struct MixedRectangle {
2. @SmallNumber var height: Int = 1
3. @SmallNumber(maximum: 9) var width: Int = 2
4. }
5. var mixedRectangle = MixedRectangle()
6. print(mixedRectangle.height)
7. // Prints "1"
8. mixedRectangle.height = 20
9. print(mixedRectangle.height)
10. // Prints "12"

The instance of SmallNumber that wraps height is created by calling SmallNumber(wrappedValue: 1), which uses the default maximum value of 12. The instance that wraps width is created by calling SmallNumber(wrappedValue: 2, maximum: 9).

### Projecting a Value From a Property Wrapper

In addition to the wrapped value, a property wrapper can expose additional functionality by defining a projected value—for example, a property wrapper that manages access to a database can expose a flushDatabaseConnection() method on its projected value. The name of the projected value is the same as the wrapped value, except it begins with a dollar sign ($). Because your code can’t define properties that start with $ the projected value never interferes with properties you define.

In the SmallNumber example above, if you try to set the property to a number that’s too large, the property wrapper adjusts the number before storing it. The code below adds a projectedValue property to the SmallNumber structure to keep track of whether the property wrapper adjusted the new value for the property before storing that new value.

1. @propertyWrapper
2. struct SmallNumber {
3. private var number = 0
4. var projectedValue = false
5. var wrappedValue: Int {
6. get { return number }
7. set {
8. if newValue > 12 {
9. number = 12
10. projectedValue = true
11. } else {
12. number = newValue
13. projectedValue = false
14. }
15. }
16. }
17. }
18. struct SomeStructure {
19. @SmallNumber var someNumber: Int
20. }
21. var someStructure = SomeStructure()
22. someStructure.someNumber = 4
23. print(someStructure.$someNumber)
24. // Prints "false"
25. someStructure.someNumber = 55
26. print(someStructure.$someNumber)
27. // Prints "true"

Writing s.$someNumber accesses the wrapper’s projected value. After storing a small number like four, the value of s.$someNumber is false. However, the projected value is true after trying to store a number that’s too large, like 55.

A property wrapper can return a value of any type as its projected value. In this example, the property wrapper exposes only one piece of information—whether the number was adjusted—so it exposes that Boolean value as its projected value. A wrapper that needs to expose more information can return an instance of some other data type, or it can return self to expose the instance of the wrapper as its projected value.

When you access a projected value from code that’s part of the type, like a property getter or an instance method, you can omit self. before the property name, just like accessing other properties. The code in the following example refers to the projected value of the wrapper around height and width as $height and $width:

1. enum Size {
2. case small, large
3. }
4. struct SizedRectangle {
5. @SmallNumber var height: Int
6. @SmallNumber var width: Int
7. mutating func resize(to size: Size) -> Bool {
8. switch size {
9. case .small:
10. height = 10
11. width = 20
12. case .large:
13. height = 100
14. width = 100
15. }
16. return $height || $width
17. }
18. }

Because property wrapper syntax is just syntactic sugar for a property with a getter and a setter, accessing height and width behaves the same as accessing any other property. For example, the code in resize(to:) accesses height and width using their property wrapper. If you call resize(to: .large), the switch case for .large sets the rectangle’s height and width to 100. The wrapper prevents the value of those properties from being larger than 12, and it sets the projected value to true, to record the fact that it adjusted their values. At the end of resize(to:), the return statement checks $height and $width to determine whether the property wrapper adjusted either height or width.

## Global and Local Variables

The capabilities described above for computing and observing properties are also available to global variables and local variables. Global variables are variables that are defined outside of any function, method, closure, or type context. Local variables are variables that are defined within a function, method, or closure context.

The global and local variables you have encountered in previous chapters have all been stored variables. Stored variables, like stored properties, provide storage for a value of a certain type and allow that value to be set and retrieved.

However, you can also define computed variables and define observers for stored variables, in either a global or local scope. Computed variables calculate their value, rather than storing it, and they are written in the same way as computed properties.

NOTE

Global constants and variables are always computed lazily, in a similar manner to [Lazy Stored Properties](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID257). Unlike lazy stored properties, global constants and variables do not need to be marked with the lazy modifier.

Local constants and variables are never computed lazily.

## Type Properties

Instance properties are properties that belong to an instance of a particular type. Every time you create a new instance of that type, it has its own set of property values, separate from any other instance.

Type properties are useful for defining values that are universal to all instances of a particular type, such as a constant property that all instances can use (like a static constant in C), or a variable property that stores a value that is global to all instances of that type (like a static variable in C).

### Type Property Syntax

In C and Objective-C, you define static constants and variables associated with a type as global static variables. In Swift, however, type properties are written as part of the type’s definition, within the type’s outer curly braces, and each type property is explicitly scoped to the type it supports.

You define type properties with the static keyword. For computed type properties for class types, you can use the class keyword instead to allow subclasses to override the superclass’s implementation. The example below shows the syntax for stored and computed type properties:

1. struct SomeStructure {
2. static var storedTypeProperty = "Some value."
3. static var computedTypeProperty: Int {
4. return 1
5. }
6. }
7. enum SomeEnumeration {
8. static var storedTypeProperty = "Some value."
9. static var computedTypeProperty: Int {
10. return 6
11. }
12. }
13. class SomeClass {
14. static var storedTypeProperty = "Some value."
15. static var computedTypeProperty: Int {
16. return 27
17. }
18. class var overrideableComputedTypeProperty: Int {
19. return 107
20. }
21. }

NOTE

The computed type property examples above are for read-only computed type properties, but you can also define read-write computed type properties with the same syntax as for computed instance properties.

### Querying and Setting Type Properties

Type properties are queried and set with dot syntax, just like instance properties. However, type properties are queried and set on the type, not on an instance of that type. For example:

1. print(SomeStructure.storedTypeProperty)
2. // Prints "Some value."
3. SomeStructure.storedTypeProperty = "Another value."
4. print(SomeStructure.storedTypeProperty)
5. // Prints "Another value."
6. print(SomeEnumeration.computedTypeProperty)
7. // Prints "6"
8. print(SomeClass.computedTypeProperty)
9. // Prints "27"

The examples that follow use two stored type properties as part of a structure that models an audio level meter for a number of audio channels. Each channel has an integer audio level between 0 and 10 inclusive.

The figure below illustrates how two of these audio channels can be combined to model a stereo audio level meter. When a channel’s audio level is 0, none of the lights for that channel are lit. When the audio level is 10, all of the lights for that channel are lit. In this figure, the left channel has a current level of 9, and the right channel has a current level of 7:

The audio channels described above are represented by instances of the AudioChannel structure:

1. struct AudioChannel {
2. static let thresholdLevel = 10
3. static var maxInputLevelForAllChannels = 0
4. var currentLevel: Int = 0 {
5. didSet {
6. if currentLevel > AudioChannel.thresholdLevel {
7. // cap the new audio level to the threshold level
8. currentLevel = AudioChannel.thresholdLevel
9. }
10. if currentLevel > AudioChannel.maxInputLevelForAllChannels {
11. // store this as the new overall maximum input level
12. AudioChannel.maxInputLevelForAllChannels = currentLevel
13. }
14. }
15. }
16. }

The AudioChannel structure defines two stored type properties to support its functionality. The first, thresholdLevel, defines the maximum threshold value an audio level can take. This is a constant value of 10 for all AudioChannel instances. If an audio signal comes in with a higher value than 10, it will be capped to this threshold value (as described below).

The second type property is a variable stored property called maxInputLevelForAllChannels. This keeps track of the maximum input value that has been received by any AudioChannel instance. It starts with an initial value of 0.

The AudioChannel structure also defines a stored instance property called currentLevel, which represents the channel’s current audio level on a scale of 0 to 10.

The currentLevel property has a didSet property observer to check the value of currentLevel whenever it is set. This observer performs two checks:

* If the new value of currentLevel is greater than the allowed thresholdLevel, the property observer caps currentLevel to thresholdLevel.
* If the new value of currentLevel (after any capping) is higher than any value previously received by any AudioChannel instance, the property observer stores the new currentLevel value in the maxInputLevelForAllChannels type property.

NOTE

In the first of these two checks, the didSet observer sets currentLevel to a different value. This does not, however, cause the observer to be called again.

You can use the AudioChannel structure to create two new audio channels called leftChannel and rightChannel, to represent the audio levels of a stereo sound system:

1. var leftChannel = AudioChannel()
2. var rightChannel = AudioChannel()

If you set the currentLevel of the left channel to 7, you can see that the maxInputLevelForAllChannels type property is updated to equal 7:

1. leftChannel.currentLevel = 7
2. print(leftChannel.currentLevel)
3. // Prints "7"
4. print(AudioChannel.maxInputLevelForAllChannels)
5. // Prints "7"

If you try to set the currentLevel of the right channel to 11, you can see that the right channel’s currentLevel property is capped to the maximum value of 10, and the maxInputLevelForAllChannels type property is updated to equal 10:

rightChannel.currentLevel = 11

print(rightChannel.currentLevel)

// Prints "10"

print(AudioChannel.maxInputLevelForAllChannels)

// Prints "10"

# Methods

Methods are functions that are associated with a particular type. Classes, structures, and enumerations can all define instance methods, which encapsulate specific tasks and functionality for working with an instance of a given type. Classes, structures, and enumerations can also define type methods, which are associated with the type itself. Type methods are similar to class methods in Objective-C.

In Objective-C, classes are the only types that can define methods. In Swift, you can choose whether to define a class, structure, or enumeration, and still have the flexibility to define methods on the type you create.

Here’s an example that defines a simple Counter class, which can be used to count the number of times an action occurs:

1. class Counter {
2. var count = 0
3. func increment() {
4. count += 1
5. }
6. func increment(by amount: Int) {
7. count += amount
8. }
9. func reset() {
10. count = 0
11. }
12. }

The Counter class defines three instance methods:

* increment() increments the counter by 1.
* increment(by: Int) increments the counter by a specified integer amount.
* reset() resets the counter to zero.

You call instance methods with the same dot syntax as properties:

let counter = Counter()

// the initial counter value is 0

counter.increment()

// the counter's value is now 1

counter.increment(by: 5)

// the counter's value is now 6

counter.reset()

// the counter's value is now 0

### The self Property

Every instance of a type has an implicit property called self, which is exactly equivalent to the instance itself. You use the self property to refer to the current instance within its own instance methods.

The increment() method in the example above could have been written like this:

func increment() {

self.count += 1

}

### Modifying Value Types from Within Instance Methods

Structures and enumerations are value types. By default, the properties of a value type cannot be modified from within its instance methods.

However, if you need to modify the properties of your structure or enumeration within a particular method, you can opt in to mutating behavior for that method. The method can then mutate (that is, change) its properties from within the method, and any changes that it makes are written back to the original structure when the method ends. The method can also assign a completely new instance to its implicit self property, and this new instance will replace the existing one when the method ends.

You can opt in to this behavior by placing the mutating keyword before the func keyword for that method:

struct Point {

var x = 0.0, y = 0.0

mutating func moveBy(x deltaX: Double, y deltaY: Double) {

x += deltaX

y += deltaY

}

}

var somePoint = Point(x: 1.0, y: 1.0)

somePoint.moveBy(x: 2.0, y: 3.0)

print("The point is now at (\(somePoint.x), \(somePoint.y))")

// Prints "The point is now at (3.0, 4.0)"

Note that you cannot call a mutating method on a constant of structure type, because its properties cannot be changed, even if they are variable properties, as described in [Stored Properties of Constant Structure Instances](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID256):

let fixedPoint = Point(x: 3.0, y: 3.0)

fixedPoint.moveBy(x: 2.0, y: 3.0)

// this will report an error

### Assigning to self Within a Mutating Method

Mutating methods can assign an entirely new instance to the implicit self property. The Point example shown above could have been written in the following way instead:

struct Point {

var x = 0.0, y = 0.0

mutating func moveBy(x deltaX: Double, y deltaY: Double) {

self = Point(x: x + deltaX, y: y + deltaY)

}

}

This version of the mutating moveBy(x:y:) method creates a new structure whose x and y values are set to the target location. The end result of calling this alternative version of the method will be exactly the same as for calling the earlier version.

Mutating methods for enumerations can set the implicit self parameter to be a different case from the same enumeration:

enum TriStateSwitch {

case off, low, high

mutating func next() {

switch self {

case .off:

self = .low

case .low:

self = .high

case .high:

self = .off

}

}

}

var ovenLight = TriStateSwitch.low

ovenLight.next()

// ovenLight is now equal to .high

ovenLight.next()

// ovenLight is now equal to .off

This example defines an enumeration for a three-state switch. The switch cycles between three different power states (off, low and high) every time its next() method is called.

## Type Methods

Instance methods, as described above, are methods that you call on an instance of a particular type. You can also define methods that are called on the type itself. These kinds of methods are called type methods. You indicate type methods by writing the static keyword before the method’s func keyword. Classes can use the class keyword instead, to allow subclasses to override the superclass’s implementation of that method.

NOTE

In Objective-C, you can define type-level methods only for Objective-C classes. In Swift, you can define type-level methods for all classes, structures, and enumerations. Each type method is explicitly scoped to the type it supports.

Type methods are called with dot syntax, like instance methods. However, you call type methods on the type, not on an instance of that type. Here’s how you call a type method on a class called SomeClass:

class SomeClass {

class func someTypeMethod() {

// type method implementation goes here

}

}

SomeClass.someTypeMethod()

ithin the body of a type method, the implicit self property refers to the type itself, rather than an instance of that type. This means that you can use self to disambiguate between type properties and type method parameters, just as you do for instance properties and instance method parameters.

More generally, any unqualified method and property names that you use within the body of a type method will refer to other type-level methods and properties. A type method can call another type method with the other method’s name, without needing to prefix it with the type name. Similarly, type methods on structures and enumerations can access type properties by using the type property’s name without a type name prefix.

The example below defines a structure called LevelTracker, which tracks a player’s progress through the different levels or stages of a game. It is a single-player game, but can store information for multiple players on a single device.

All of the game’s levels (apart from level one) are locked when the game is first played. Every time a player finishes a level, that level is unlocked for all players on the device. The LevelTracker structure uses type properties and methods to keep track of which levels of the game have been unlocked. It also tracks the current level for an individual player.

1. struct LevelTracker {
2. static var highestUnlockedLevel = 1
3. var currentLevel = 1
4. static func unlock(\_ level: Int) {
5. if level > highestUnlockedLevel { highestUnlockedLevel = level }
6. }
7. static func isUnlocked(\_ level: Int) -> Bool {
8. return level <= highestUnlockedLevel
9. }
10. @discardableResult
11. mutating func advance(to level: Int) -> Bool {
12. if LevelTracker.isUnlocked(level) {
13. currentLevel = level
14. return true
15. } else {
16. return false
17. }
18. }
19. }

The LevelTracker structure keeps track of the highest level that any player has unlocked. This value is stored in a type property called highestUnlockedLevel.

LevelTracker also defines two type functions to work with the highestUnlockedLevel property. The first is a type function called unlock(\_:), which updates the value of highestUnlockedLevel whenever a new level is unlocked. The second is a convenience type function called isUnlocked(\_:), which returns true if a particular level number is already unlocked. (Note that these type methods can access the highestUnlockedLevel type property without your needing to write it as LevelTracker.highestUnlockedLevel.)

In addition to its type property and type methods, LevelTracker tracks an individual player’s progress through the game. It uses an instance property called currentLevel to track the level that a player is currently playing.

To help manage the currentLevel property, LevelTracker defines an instance method called advance(to:). Before updating currentLevel, this method checks whether the requested new level is already unlocked. The advance(to:) method returns a Boolean value to indicate whether or not it was actually able to set currentLevel. Because it’s not necessarily a mistake for code that calls the advance(to:) method to ignore the return value, this function is marked with the @discardableResult attribute. For more information about this attribute, see [Attributes](https://docs.swift.org/swift-book/ReferenceManual/Attributes.html).

The LevelTracker structure is used with the Player class, shown below, to track and update the progress of an individual player:

1. class Player {
2. var tracker = LevelTracker()
3. let playerName: String
4. func complete(level: Int) {
5. LevelTracker.unlock(level + 1)
6. tracker.advance(to: level + 1)
7. }
8. init(name: String) {
9. playerName = name
10. }
11. }

The Player class creates a new instance of LevelTracker to track that player’s progress. It also provides a method called complete(level:), which is called whenever a player completes a particular level. This method unlocks the next level for all players and updates the player’s progress to move them to the next level. (The Boolean return value of advance(to:) is ignored, because the level is known to have been unlocked by the call to LevelTracker.unlock(\_:) on the previous line.)

You can create an instance of the Player class for a new player, and see what happens when the player completes level one:

1. var player = Player(name: "Argyrios")
2. player.complete(level: 1)
3. print("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")
4. // Prints "highest unlocked level is now 2"

If you create a second player, whom you try to move to a level that is not yet unlocked by any player in the game, the attempt to set the player’s current level fails:

1. player = Player(name: "Beto")
2. if player.tracker.advance(to: 6) {
3. print("player is now on level 6")
4. } else {
5. print("level 6 has not yet been unlocked")
6. }
7. // Prints "level 6 has not yet been unlocked"