Classes and Structures

*Classes* and *structures* are general-purpose, flexible constructs that become the building blocks of your program’s code. You define properties and methods to add functionality to your classes and structures by using exactly the same syntax as for constants, variables, and functions.

Unlike other programming languages, Swift does not require you to create separate interface and implementation files for custom classes and structures. In Swift, you define a class or a structure in a single file, and the external interface to that class or structure is automatically made available for other code to use.

NOTE

An instance of a *class* is traditionally known as an *object*. However, Swift classes and structures are much closer in functionality than in other languages, and much of this chapter describes functionality that can apply to instances of *either* a class or a structure type. Because of this, the more general term *instance* is used.

Comparing Classes and Structures

Classes and structures 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://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Properties.html#//apple_ref/doc/uid/TP40014097-CH14-ID254), [Methods](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Methods.html#//apple_ref/doc/uid/TP40014097-CH15-ID234), [Subscripts](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Subscripts.html#//apple_ref/doc/uid/TP40014097-CH16-ID305), [Initialization](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Initialization.html#//apple_ref/doc/uid/TP40014097-CH18-ID203), [Extensions](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Extensions.html#//apple_ref/doc/uid/TP40014097-CH24-ID151), and [Protocols](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html#//apple_ref/doc/uid/TP40014097-CH25-ID267).

Classes have additional capabilities that structures do not:

* 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.

For more information, see [Inheritance](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Inheritance.html#//apple_ref/doc/uid/TP40014097-CH17-ID193), [Type Casting](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TypeCasting.html#//apple_ref/doc/uid/TP40014097-CH22-ID338), [Deinitialization](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Deinitialization.html" \l "//apple_ref/doc/uid/TP40014097-CH19-ID142), and [Automatic Reference Counting](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html#//apple_ref/doc/uid/TP40014097-CH20-ID48).

NOTE

Structures are always copied when they are passed around in your code, and do not use reference counting.

Definition Syntax

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

1. class SomeClass {
2. // class definition goes here
3. }
4. struct SomeStructure {
5. // structure definition goes here
6. }

NOTE

Whenever you define a new class or structure, you effectively define a brand new Swift type. Give types UpperCamelCase names (such as SomeClass and SomeStructure here) to match the capitalization of standard Swift types (such as String, Int, and Bool). Conversely, always give properties and methods lowerCamelCasenames (such as frameRate and incrementCount) to differentiate them from type names.

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

1. struct Resolution {
2. var width = 0
3. var height = 0
4. }
5. class VideoMode {
6. var resolution = Resolution()
7. var interlaced = false
8. var frameRate = 0.0
9. var name: String?
10. }

The example above defines a new structure called Resolution, to describe a pixel-based display resolution. This structure has two stored properties called width and height. Stored properties are constants or variables that are bundled up and stored as part of the class or structure. These two properties are inferred to be of type Int by setting them to an initial integer value of 0.

The example above also defines a new class called VideoMode, to describe a specific video mode for video display. This class has four variable stored properties. The first, resolution, is initialized with a new Resolutionstructure instance, which infers a property type of Resolution. For the other three properties, new VideoModeinstances will be initialized with an interlaced setting of false (meaning “noninterlaced video”), a playback frame rate of 0.0, and an optional String value called name. The name property is automatically given a default value of nil, or “no name value”, because it is of an optional type.

Class and Structure Instances

The Resolution structure definition and the VideoMode class definition only describe what a Resolution or VideoMode will look like. They themselves do not describe a specific resolution or video mode. To do that, you need to create an instance of the structure or class.

The syntax for creating instances is very similar for both structures and classes:

1. let someResolution = Resolution()
2. let someVideoMode = VideoMode()

Structures and classes both use initializer syntax for new instances. The simplest form of initializer syntax uses the type name of the class or structure followed by empty parentheses, such as Resolution() or VideoMode(). This creates a new instance of the class or structure, with any properties initialized to their default values. Class and structure initialization is described in more detail in [Initialization](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Initialization.html#//apple_ref/doc/uid/TP40014097-CH18-ID203).

Accessing Properties

You can access the properties of an instance using *dot syntax*. In dot syntax, you write the property name immediately after the instance name, separated by a period (.), without any spaces:

1. print("The width of someResolution is \(someResolution.width)")
2. // Prints "The width of someResolution is 0"

In this example, someResolution.width refers to the width property of someResolution, and returns its default initial value of 0.

You can drill down into sub-properties, such as the width property in the resolution property of a VideoMode:

1. print("The width of someVideoMode is \(someVideoMode.resolution.width)")
2. // Prints "The width of someVideoMode is 0"

You can also use dot syntax to assign a new value to a variable property:

1. someVideoMode.resolution.width = 1280
2. print("The width of someVideoMode is now \(someVideoMode.resolution.width)")
3. // Prints "The width of someVideoMode is now 1280"

NOTE

Unlike Objective-C, Swift enables you to set sub-properties of a structure property directly. In the last example above, the width property of the resolution property of someVideoMode is set directly, without your needing to set the entire resolution property to a new value.

Memberwise Initializers for Structure Types

All structures have an automatically-generated *memberwise initializer*, which you can use to initialize the member properties of new structure instances. Initial values for the properties of the new instance can be passed to the memberwise initializer by name:

1. let vga = Resolution(width: 640, height: 480)

Unlike structures, class instances do not receive a default memberwise initializer. Initializers are described in more detail in [Initialization](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Initialization.html#//apple_ref/doc/uid/TP40014097-CH18-ID203).

Structures and Enumerations Are Value Types

A *value type* is a type whose value is *copied* when it is assigned to a variable or constant, or when it is passed to a function.

You’ve actually been using value types extensively throughout the previous chapters. In fact, all of the basic types in Swift—integers, floating-point numbers, Booleans, strings, arrays and dictionaries—are value types, and are implemented as structures behind the scenes.

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.

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 hdand 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 cinemainstance. The end result is two completely separate instances, which just happened to contain the same numeric values. Because they are separate instances, setting the width of cinema to 2048 doesn’t affect the width stored in hd.

The same behavior applies to enumerations:

1. enum CompassPoint {
2. case north, south, east, west
3. }
4. var currentDirection = CompassPoint.west
5. let rememberedDirection = currentDirection
6. currentDirection = .east
7. if rememberedDirection == .west {
8. print("The remembered direction is still .west")
9. }
10. // Prints "The remembered direction is still .west"

When rememberedDirection is assigned the value of currentDirection, it is actually set to a copy of that value. Changing the value of currentDirection thereafter does not 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 instead.

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 VideoModeclass. The video mode is assigned a copy of the HD resolution of 1920 by 1080 from before. It is set to be interlaced, and is given a name of "1080i". Finally, it is set to a frame rate of 25.0 frames per second.

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

1. let alsoTenEighty = tenEighty
2. alsoTenEighty.frameRate = 30.0

Because classes are reference types, tenEighty and alsoTenEighty actually both refer to the *same* VideoModeinstance. Effectively, they are just two different names for the same single instance.

Checking the frameRate property of tenEighty shows that it correctly reports the new frame rate of 30.0 from the underlying VideoMode instance:

1. print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
2. // Prints "The frameRate property of tenEighty is now 30.0"

Note that tenEighty and alsoTenEighty are declared as *constants*, rather than variables. However, you can still change tenEighty.frameRate and alsoTenEighty.frameRate because the values of the tenEighty and alsoTenEighty constants themselves do not actually change. tenEighty and alsoTenEighty themselves do not “store” the VideoMode instance—instead, they both *refer* to a VideoMode instance behind the scenes. It is the frameRate property of the underlying VideoMode that is changed, not the values of the constant references to that VideoMode.

Identity Operators

Because classes are reference types, it is possible for multiple constants and variables to refer to the same single instance of a class behind the scenes. (The same is not 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 if 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 ===) does not 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 classes and structures, it is your responsibility to decide what qualifies as two instances being “equal”. The process of defining your own implementations of the “equal to” and “not equal to” operators is described in [Equivalence Operators](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AdvancedOperators.html#//apple_ref/doc/uid/TP40014097-CH27-ID45).

Pointers

If you have experience with C, C++, or Objective-C, you may know that these languages use *pointers* to refer to addresses in memory. A Swift constant or variable that refers to an instance of some reference type is similar to a pointer in C, but is not a direct pointer to an address in memory, and does not require you to write an asterisk (\*) to indicate that you are creating a reference. Instead, these references are defined like any other constant or variable in Swift.

Choosing Between Classes and Structures

You can use both classes and structures to define custom data types to use as the building blocks of your program’s code.

However, structure instances are always passed by *value*, and class instances are always passed by *reference*. This means that they are suited to different kinds of tasks. As you consider the data constructs and functionality that you need for a project, decide whether each data construct should be defined as a class or as a structure.

As a general guideline, consider creating a structure when one or more of these conditions apply:

* The structure’s primary purpose is to encapsulate a few relatively simple data values.
* It is reasonable to expect that the encapsulated values will be copied rather than referenced when you assign or pass around an instance of that structure.
* Any properties stored by the structure are themselves value types, which would also be expected to be copied rather than referenced.
* The structure does not need to inherit properties or behavior from another existing type.

Examples of good candidates for structures include:

* The size of a geometric shape, perhaps encapsulating a width property and a height property, both of type Double.
* A way to refer to ranges within a series, perhaps encapsulating a start property and a length property, both of type Int.
* A point in a 3D coordinate system, perhaps encapsulating x, y and z properties, each of type Double.

In all other cases, define a class, and create instances of that class to be managed and passed by reference. In practice, this means that most custom data constructs should be classes, not structures.

Assignment and Copy Behavior for Strings, Arrays, and Dictionaries

In Swift, many basic data types such as String, Array, and Dictionary are implemented as structures. This means that data such as strings, arrays, and dictionaries are copied when they are assigned to a new constant or variable, or when they are passed to a function or method.

This behavior is different from Foundation: NSString, NSArray, and NSDictionary are implemented as classes, not structures. Strings, arrays, and dictionaries in Foundation are always assigned and passed around as a reference to an existing instance, rather than as a copy.

NOTE

The description above refers to the “copying” of strings, arrays, and dictionaries. The behavior you see in your code will always be as if a copy took place. However, Swift only performs an *actual* copy behind the scenes when it is absolutely necessary to do so. Swift manages all value copying to ensure optimal performance, and you should not avoid assignment to try to preempt this optimization.

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.

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 varkeyword) 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://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Initialization.html#//apple_ref/doc/uid/TP40014097-CH18-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://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Initialization.html#//apple_ref/doc/uid/TP40014097-CH18-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 {
2. /\*
3. DataImporter is a class to import data from an external file.
4. The class is assumed to take a nontrivial amount of time to initialize.
5. \*/
6. var filename = "data.txt"
7. // the DataImporter class would provide data importing functionality here
8. }
9. class DataManager {
10. lazy var importer = DataImporter()
11. var data = [String]()
12. // the DataManager class would provide data management functionality here
13. }
14. let manager = DataManager()
15. manager.data.append("Some data")
16. manager.data.append("Some more data")
17. // 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 Stringvalues. 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:

1. print(manager.importer.filename)
2. // the DataImporter instance for the importer property has now been created
3. // 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.

1. struct Point {
2. var x = 0.0, y = 0.0
3. }
4. struct Size {
5. var width = 0.0, height = 0.0
6. }
7. struct Rect {
8. var origin = Point()
9. var size = Size()
10. var center: Point {
11. get {
12. let centerX = origin.x + (size.width / 2)
13. let centerY = origin.y + (size.height / 2)
14. return Point(x: centerX, y: centerY)
15. }
16. set(newCenter) {
17. origin.x = newCenter.x - (size.width / 2)
18. origin.y = newCenter.y - (size.height / 2)
19. }
20. }
21. }
22. var square = Rect(origin: Point(x: 0.0, y: 0.0),
23. size: Size(width: 10.0, height: 10.0))
24. let initialSquareCenter = square.center
25. square.center = Point(x: 15.0, y: 15.0)
26. print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
27. // Prints "square.origin is now at (10.0, 10.0)"

This example defines three structures for working with geometric shapes:

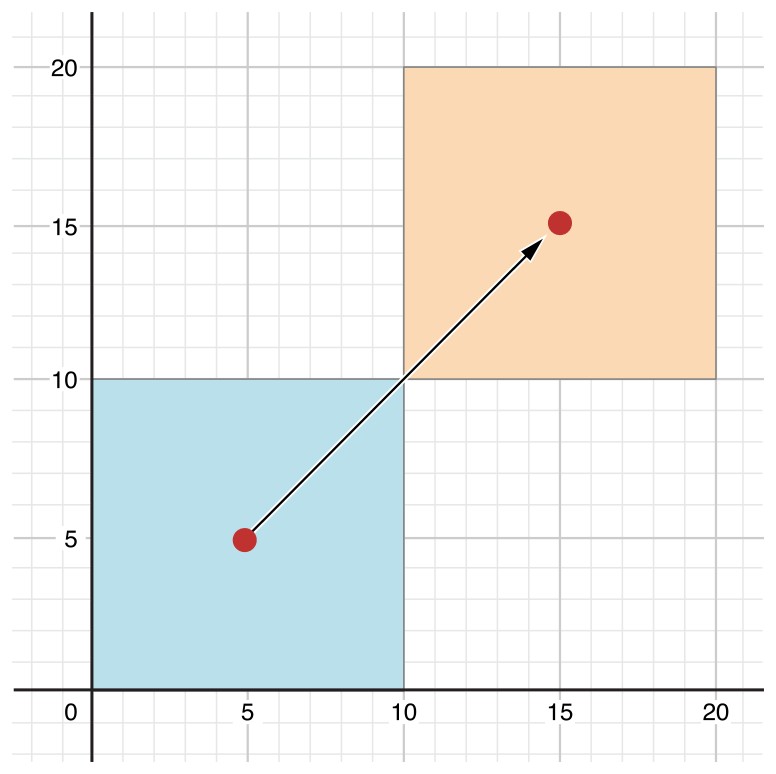
* Point encapsulates the x- and y-coordinate of a point.
* Size encapsulates a width and a height.
* Rect defines a rectangle by an origin point and a size.

The Rect structure also provides a computed property called center. The current center position of a Rect can always be determined from its origin and size, and so you don’t need to store the center point as an explicit Point value. Instead, Rect defines a custom getter and setter for a computed variable called center, to enable you to work with the rectangle’s center as if it were a real stored property.

The preceding example creates a new Rect variable called square. The square variable is initialized with an origin point of (0, 0), and a width and height of 10. This square is represented by the blue square in the diagram below.

The square variable’s center property is then accessed through dot syntax (square.center), which causes the getter for center to be called, to retrieve the current property value. Rather than returning an existing value, the getter actually calculates and returns a new Point to represent the center of the square. As can be seen above, the getter correctly returns a center point of (5, 5).

The center property is then set to a new value of (15, 15), which moves the square up and to the right, to the new position shown by the orange square in the diagram below. Setting the center property calls the setter for center, which modifies the x and y values of the stored origin property, and moves the square to its new position.



Shorthand Setter Declaration

If a computed property’s setter does not define a name for the new value to be set, a default name of newValueis used. Here’s an alternative version of the Rect structure, which 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. }

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:

1. struct Cuboid {
2. var width = 0.0, height = 0.0, depth = 0.0
3. var volume: Double {
4. return width \* height \* depth
5. }
6. }
7. let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
8. print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
9. // 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://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Inheritance.html#//apple_ref/doc/uid/TP40014097-CH17-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://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Initialization.html#//apple_ref/doc/uid/TP40014097-CH18-ID215) and [Initializer Delegation for Class Types](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Initialization.html#//apple_ref/doc/uid/TP40014097-CH18-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 totalStepsagainst 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 didSetobservers 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://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Declarations.html#//apple_ref/doc/uid/TP40014097-CH34-ID545).

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://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Properties.html#//apple_ref/doc/uid/TP40014097-CH14-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.

You can also define properties that belong to the type itself, not to any one instance of that type. There will only ever be one copy of these properties, no matter how many instances of that type you create. These kinds of properties are called *type properties*.

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).

Stored type properties can be variables or constants. Computed type properties are always declared as variable properties, in the same way as computed instance properties.

NOTE

Unlike stored instance properties, you must always give stored type properties a default value. This is because the type itself does not have an initializer that can assign a value to a stored type property at initialization time.

Stored type properties are lazily initialized on their first access. They are guaranteed to be initialized only once, even when accessed by multiple threads simultaneously, and they do not need to be marked with the lazy modifier.

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.

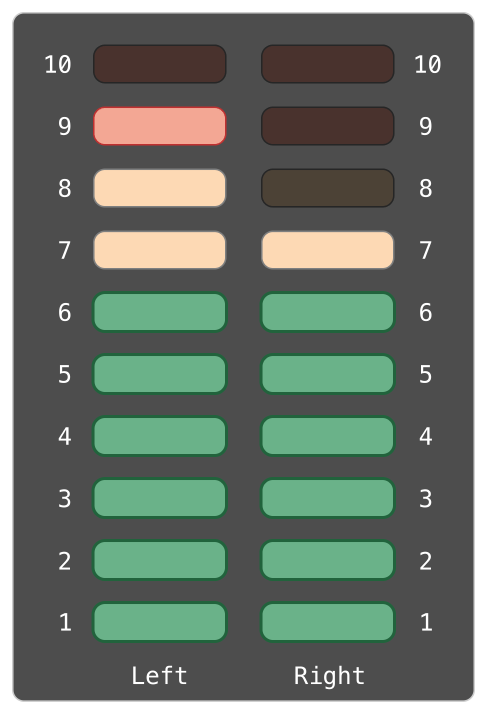
Querying and Setting Type Properties

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 10for 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 currentLevelproperty is capped to the maximum value of 10, and the maxInputLevelForAllChannels type property is updated to equal 10:

1. rightChannel.currentLevel = 11
2. print(rightChannel.currentLevel)
3. // Prints "10"
4. print(AudioChannel.maxInputLevelForAllChannels)
5. // 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.

The fact that structures and enumerations can define methods in Swift is a major difference from C and 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.

Instance Methods

*Instance methods* are functions that belong to instances of a particular class, structure, or enumeration. They support the functionality of those instances, either by providing ways to access and modify instance properties, or by providing functionality related to the instance’s purpose. Instance methods have exactly the same syntax as functions, as described in [Functions](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Functions.html#//apple_ref/doc/uid/TP40014097-CH10-ID158).

You write an instance method within the opening and closing braces of the type it belongs to. An instance method has implicit access to all other instance methods and properties of that type. An instance method can be called only on a specific instance of the type it belongs to. It cannot be called in isolation without an existing instance.

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.

The Counter class also declares a variable property, count, to keep track of the current counter value.

You call instance methods with the same dot syntax as properties:

1. let counter = Counter()
2. // the initial counter value is 0
3. counter.increment()
4. // the counter's value is now 1
5. counter.increment(by: 5)
6. // the counter's value is now 6
7. counter.reset()
8. // the counter's value is now 0

Function parameters can have both a name (for use within the function’s body) and an argument label (for use when calling the function), as described in [Function Argument Labels and Parameter Names](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Functions.html#//apple_ref/doc/uid/TP40014097-CH10-ID166). The same is true for method parameters, because methods are just functions that are associated with a type.

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:

1. func increment() {
2. self.count += 1
3. }

In practice, you don’t need to write self in your code very often. If you don’t explicitly write self, Swift assumes that you are referring to a property or method of the current instance whenever you use a known property or method name within a method. This assumption is demonstrated by the use of count (rather than self.count) inside the three instance methods for Counter.

The main exception to this rule occurs when a parameter name for an instance method has the same name as a property of that instance. In this situation, the parameter name takes precedence, and it becomes necessary to refer to the property in a more qualified way. You use the self property to distinguish between the parameter name and the property name.

Here, self disambiguates between a method parameter called x and an instance property that is also called x:

1. struct Point {
2. var x = 0.0, y = 0.0
3. func isToTheRightOf(x: Double) -> Bool {
4. return self.x > x
5. }
6. }
7. let somePoint = Point(x: 4.0, y: 5.0)
8. if somePoint.isToTheRightOf(x: 1.0) {
9. print("This point is to the right of the line where x == 1.0")
10. }
11. // Prints "This point is to the right of the line where x == 1.0"

Without the self prefix, Swift would assume that both uses of x referred to the method parameter called x.

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:

1. struct Point {
2. var x = 0.0, y = 0.0
3. mutating func moveBy(x deltaX: Double, y deltaY: Double) {
4. x += deltaX
5. y += deltaY
6. }
7. }
8. var somePoint = Point(x: 1.0, y: 1.0)
9. somePoint.moveBy(x: 2.0, y: 3.0)
10. print("The point is now at (\(somePoint.x), \(somePoint.y))")
11. // Prints "The point is now at (3.0, 4.0)"

The Point structure above defines a mutating moveBy(x:y:) method, which moves a Point instance by a certain amount. Instead of returning a new point, this method actually modifies the point on which it is called. The mutating keyword is added to its definition to enable it to modify its properties.

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://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Properties.html#//apple_ref/doc/uid/TP40014097-CH14-ID256):

1. let fixedPoint = Point(x: 3.0, y: 3.0)
2. fixedPoint.moveBy(x: 2.0, y: 3.0)
3. // 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:

1. struct Point {
2. var x = 0.0, y = 0.0
3. mutating func moveBy(x deltaX: Double, y deltaY: Double) {
4. self = Point(x: x + deltaX, y: y + deltaY)
5. }
6. }

This version of the mutating moveBy(x:y:) method creates a brand 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:

1. enum TriStateSwitch {
2. case off, low, high
3. mutating func next() {
4. switch self {
5. case .off:
6. self = .low
7. case .low:
8. self = .high
9. case .high:
10. self = .off
11. }
12. }
13. }
14. var ovenLight = TriStateSwitch.low
15. ovenLight.next()
16. // ovenLight is now equal to .high
17. ovenLight.next()
18. // 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 are called 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 may also use the class keyword 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:

1. class SomeClass {
2. class func someTypeMethod() {
3. // type method implementation goes here
4. }
5. }
6. SomeClass.someTypeMethod()

Within 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://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Attributes.html#//apple_ref/doc/uid/TP40014097-CH35-ID347).

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"