# Subscripts

Classes, structures, and enumerations can define subscripts, which are shortcuts for accessing the member elements of a collection, list, or sequence. You use subscripts to set and retrieve values by index without needing separate methods for setting and retrieval. For example, you access elements in an Array instance as someArray[index] and elements in a Dictionary instance as someDictionary[key].

You can define multiple subscripts for a single type, and the appropriate subscript overload to use is selected based on the type of index value you pass to the subscript. Subscripts are not limited to a single dimension, and you can define subscripts with multiple input parameters to suit your custom type’s needs.

## Subscript Syntax

Subscripts enable you to query instances of a type by writing one or more values in square brackets after the instance name. Their syntax is similar to both instance method syntax and computed property syntax. You write subscript definitions with the subscript keyword, and specify one or more input parameters and a return type, in the same way as instance methods. Unlike instance methods, subscripts can be read-write or read-only. This behavior is communicated by a getter and setter in the same way as for computed properties:

1. subscript(index: Int) -> Int {
2. get {
3. // Return an appropriate subscript value here.
4. }
5. set(newValue) {
6. // Perform a suitable setting action here.
7. }
8. }

The type of newValue is the same as the return value of the subscript. As with computed properties, you can choose not to specify the setter’s (newValue) parameter. A default parameter called newValue is provided to your setter if you do not provide one yourself.

As with read-only computed properties, you can simplify the declaration of a read-only subscript by removing the get keyword and its braces:

1. subscript(index: Int) -> Int {
2. // Return an appropriate subscript value here.
3. }

Here’s an example of a read-only subscript implementation, which defines a TimesTable structure to represent an n-times-table of integers:

1. struct TimesTable {
2. let multiplier: Int
3. subscript(index: Int) -> Int {
4. return multiplier \* index
5. }
6. }
7. let threeTimesTable = TimesTable(multiplier: 3)
8. print("six times three is \(threeTimesTable[6])")
9. // Prints "six times three is 18"

In this example, a new instance of TimesTable is created to represent the three-times-table. This is indicated by passing a value of 3 to the structure’s initializer as the value to use for the instance’s multiplier parameter.

You can query the threeTimesTable instance by calling its subscript, as shown in the call to threeTimesTable[6]. This requests the sixth entry in the three-times-table, which returns a value of 18, or 3 times 6.

NOTE

An n-times-table is based on a fixed mathematical rule. It is not appropriate to set threeTimesTable[someIndex] to a new value, and so the subscript for TimesTable is defined as a read-only subscript.

## Subscript Usage

The exact meaning of “subscript” depends on the context in which it is used. Subscripts are typically used as a shortcut for accessing the member elements in a collection, list, or sequence. You are free to implement subscripts in the most appropriate way for your particular class or structure’s functionality.

For example, Swift’s Dictionary type implements a subscript to set and retrieve the values stored in a Dictionary instance. You can set a value in a dictionary by providing a key of the dictionary’s key type within subscript brackets, and assigning a value of the dictionary’s value type to the subscript:

1. var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
2. numberOfLegs["bird"] = 2

The example above defines a variable called numberOfLegs and initializes it with a dictionary literal containing three key-value pairs. The type of the numberOfLegs dictionary is inferred to be [String: Int]. After creating the dictionary, this example uses subscript assignment to add a String key of "bird" and an Int value of 2 to the dictionary.

For more information about Dictionary subscripting, see [Accessing and Modifying a Dictionary](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID116).

NOTE

Swift’s Dictionary type implements its key-value subscripting as a subscript that takes and returns an optional type. For the numberOfLegs dictionary above, the key-value subscript takes and returns a value of type Int?, or “optional int”. The Dictionary type uses an optional subscript type to model the fact that not every key will have a value, and to give a way to delete a value for a key by assigning a nil value for that key.

## Subscript Options

Subscripts can take any number of input parameters, and these input parameters can be of any type. Subscripts can also return any type. Subscripts can use variadic parameters, but they can’t use in-out parameters or provide default parameter values.

A class or structure can provide as many subscript implementations as it needs, and the appropriate subscript to be used will be inferred based on the types of the value or values that are contained within the subscript brackets at the point that the subscript is used. This definition of multiple subscripts is known as subscript overloading.

While it is most common for a subscript to take a single parameter, you can also define a subscript with multiple parameters if it is appropriate for your type. The following example defines a Matrix structure, which represents a two-dimensional matrix of Double values. The Matrix structure’s subscript takes two integer parameters:

1. struct Matrix {
2. let rows: Int, columns: Int
3. var grid: [Double]
4. init(rows: Int, columns: Int) {
5. self.rows = rows
6. self.columns = columns
7. grid = Array(repeating: 0.0, count: rows \* columns)
8. }
9. func indexIsValid(row: Int, column: Int) -> Bool {
10. return row >= 0 && row < rows && column >= 0 && column < columns
11. }
12. subscript(row: Int, column: Int) -> Double {
13. get {
14. assert(indexIsValid(row: row, column: column), "Index out of range")
15. return grid[(row \* columns) + column]
16. }
17. set {
18. assert(indexIsValid(row: row, column: column), "Index out of range")
19. grid[(row \* columns) + column] = newValue
20. }
21. }
22. }

Matrix provides an initializer that takes two parameters called rows and columns, and creates an array that is large enough to store rows \* columns values of type Double. Each position in the matrix is given an initial value of 0.0. To achieve this, the array’s size, and an initial cell value of 0.0, are passed to an array initializer that creates and initializes a new array of the correct size. This initializer is described in more detail in [Creating an Array with a Default Value](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID501).

You can construct a new Matrix instance by passing an appropriate row and column count to its initializer:

1. var matrix = Matrix(rows: 2, columns: 2)

The example above creates a new Matrix instance with two rows and two columns. The grid array for this Matrix instance is effectively a flattened version of the matrix, as read from top left to bottom right:

Values in the matrix can be set by passing row and column values into the subscript, separated by a comma:

1. matrix[0, 1] = 1.5
2. matrix[1, 0] = 3.2

These two statements call the subscript’s setter to set a value of 1.5 in the top right position of the matrix (where row is 0 and column is 1), and 3.2 in the bottom left position (where row is 1 and column is 0):

The Matrix subscript’s getter and setter both contain an assertion to check that the subscript’s row and column values are valid. To assist with these assertions, Matrix includes a convenience method called indexIsValid(row:column:), which checks whether the requested row and column are inside the bounds of the matrix:

1. func indexIsValid(row: Int, column: Int) -> Bool {
2. return row >= 0 && row < rows && column >= 0 && column < columns
3. }

An assertion is triggered if you try to access a subscript that is outside of the matrix bounds:

1. let someValue = matrix[2, 2]
2. // This triggers an assert, because [2, 2] is outside of the matrix bounds.

## Type Subscripts

Instance subscripts, as described above, are subscripts that you call on an instance of a particular type. You can also define subscripts that are called on the type itself. This kind of subscript is called a type subscript. You indicate a type subscript by writing the static keyword before the subscript keyword. Classes can use the class keyword instead, to allow subclasses to override the superclass’s implementation of that subscript. The example below shows how you define and call a type subscript:

1. enum Planet: Int {
2. case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune
3. static subscript(n: Int) -> Planet {
4. return Planet(rawValue: n)!
5. }
6. }
7. let mars = Planet[4]
8. print(mars)

# Inheritance

A class can inherit methods, properties, and other characteristics from another class. When one class inherits from another, the inheriting class is known as a subclass, and the class it inherits from is known as its superclass.

## Defining a Base Class

Any class that does not inherit from another class is known as a base class.

NOTE

Swift classes do not inherit from a universal base class. Classes you define without specifying a superclass automatically become base classes for you to build upon.

## Subclassing

Subclassing is the act of basing a new class on an existing class. The subclass inherits characteristics from the existing class, which you can then refine. You can also add new characteristics to the subclass.

To indicate that a subclass has a superclass, write the subclass name before the superclass name, separated by a colon:

class SomeSubclass: SomeSuperclass {

// subclass definition goes here

}

The following example defines a subclass called Bicycle, with a superclass of Vehicle:

class Bicycle: Vehicle {

var hasBasket = false

}

## Overriding

A subclass can provide its own custom implementation of an instance method, type method, instance property, type property, or subscript that it would otherwise inherit from a superclass. This is known as overriding.

To override a characteristic that would otherwise be inherited, you prefix your overriding definition with the override keyword. Doing so clarifies that you intend to provide an override and have not provided a matching definition by mistake. Overriding by accident can cause unexpected behavior, and any overrides without the override keyword are diagnosed as an error when your code is compiled.

The override keyword also prompts the Swift compiler to check that your overriding class’s superclass (or one of its parents) has a declaration that matches the one you provided for the override. This check ensures that your overriding definition is correct.

### Accessing Superclass Methods, Properties, and Subscripts

When you provide a method, property, or subscript override for a subclass, it is sometimes useful to use the existing superclass implementation as part of your override. For example, you can refine the behavior of that existing implementation, or store a modified value in an existing inherited variable.

Where this is appropriate, you access the superclass version of a method, property, or subscript by using the super prefix:

* An overridden method named someMethod() can call the superclass version of someMethod() by calling super.someMethod() within the overriding method implementation.
* An overridden property called someProperty can access the superclass version of someProperty as super.someProperty within the overriding getter or setter implementation.
* An overridden subscript for someIndex can access the superclass version of the same subscript as super[someIndex] from within the overriding subscript implementation.

#### Overriding Property Getters and Setters

You can provide a custom getter (and setter, if appropriate) to override any inherited property, regardless of whether the inherited property is implemented as a stored or computed property at source. The stored or computed nature of an inherited property is not known by a subclass—it only knows that the inherited property has a certain name and type. You must always state both the name and the type of the property you are overriding, to enable the compiler to check that your override matches a superclass property with the same name and type.

You can present an inherited read-only property as a read-write property by providing both a getter and a setter in your subclass property override. You cannot, however, present an inherited read-write property as a read-only property.

NOTE

If you provide a setter as part of a property override, you must also provide a getter for that override. If you don’t want to modify the inherited property’s value within the overriding getter, you can simply pass through the inherited value by returning super.someProperty from the getter, where someProperty is the name of the property you are overriding.

The following example defines a new class called Car, which is a subclass of Vehicle. The Car class introduces a new stored property called gear, with a default integer value of 1. The Car class also overrides the description property it inherits from Vehicle, to provide a custom description that includes the current gear:

1. class Car: Vehicle {
2. var gear = 1
3. override var description: String {
4. return super.description + " in gear \(gear)"
5. }
6. }

The override of the description property starts by calling super.description, which returns the Vehicle class’s description property. The Car class’s version of description then adds some extra text onto the end of this description to provide information about the current gear.

If you create an instance of the Car class and set its gear and currentSpeed properties, you can see that its description property returns the tailored description defined within the Car class:

1. let car = Car()
2. car.currentSpeed = 25.0
3. car.gear = 3
4. print("Car: \(car.description)")
5. // Car: traveling at 25.0 miles per hour in gear 3

#### Overriding Property Observers

You can use property overriding to add property observers to an inherited property. This enables you to be notified when the value of an inherited property changes, regardless of how that property was originally implemented. For more information on property observers, see [Property Observers](https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID262).

NOTE

You cannot add property observers to inherited constant stored properties or inherited read-only computed properties. The value of these properties cannot be set, and so it is not appropriate to provide a willSet or didSet implementation as part of an override.

Note also that you cannot provide both an overriding setter and an overriding property observer for the same property. If you want to observe changes to a property’s value, and you are already providing a custom setter for that property, you can simply observe any value changes from within the custom setter.

The following example defines a new class called AutomaticCar, which is a subclass of Car. The AutomaticCar class represents a car with an automatic gearbox, which automatically selects an appropriate gear to use based on the current speed:

1. class AutomaticCar: Car {
2. override var currentSpeed: Double {
3. didSet {
4. gear = Int(currentSpeed / 10.0) + 1
5. }
6. }
7. }

Whenever you set the currentSpeed property of an AutomaticCar instance, the property’s didSet observer sets the instance’s gear property to an appropriate choice of gear for the new speed. Specifically, the property observer chooses a gear that is the new currentSpeed value divided by 10, rounded down to the nearest integer, plus 1. A speed of 35.0 produces a gear of 4:

1. let automatic = AutomaticCar()
2. automatic.currentSpeed = 35.0
3. print("AutomaticCar: \(automatic.description)")
4. // AutomaticCar: traveling at 35.0 miles per hour in gear 4

## Preventing Overrides

You can prevent a method, property, or subscript from being overridden by marking it as final. Do this by writing the final modifier before the method, property, or subscript’s introducer keyword (such as final var, final func, final class func, and final subscript).

Any attempt to override a final method, property, or subscript in a subclass is reported as a compile-time error. Methods, properties, or subscripts that you add to a class in an extension can also be marked as final within the extension’s definition.

You can mark an entire class as final by writing the final modifier before the class keyword in its class definition (final class). Any attempt to subclass a final class is reported as a compile-time error.

# Initialization

Initialization is the process of preparing an instance of a class, structure, or enumeration for use. This process involves setting an initial value for each stored property on that instance and performing any other setup or initialization that is required before the new instance is ready for use.

You implement this initialization process by defining initializers, which are like special methods that can be called to create a new instance of a particular type. Unlike Objective-C initializers, Swift initializers do not return a value. Their primary role is to ensure that new instances of a type are correctly initialized before they are used for the first time.

Instances of class types can also implement a deinitializer, which performs any custom cleanup just before an instance of that class is deallocated

## Setting Initial Values for Stored Properties

Classes and structures must set all of their stored properties to an appropriate initial value by the time an instance of that class or structure is created. Stored properties cannot be left in an indeterminate state.

You can set an initial value for a stored property within an initializer, or by assigning a default property value as part of the property’s definition. These actions are described in the following sections.

NOTE

When you assign a default value to a stored property, or set its initial value within an initializer, the value of that property is set directly, without calling any property observers.

### Initializers

Initializers are called to create a new instance of a particular type. In its simplest form, an initializer is like an instance method with no parameters, written using the init keyword:

init() {

// perform some initialization here

}

The example below defines a new structure called Fahrenheit to store temperatures expressed in the Fahrenheit scale. The Fahrenheit structure has one stored property, temperature, which is of type Double:

struct Fahrenheit {

var temperature: Double

init() {

temperature = 32.0

}

}

var f = Fahrenheit()

print("The default temperature is \(f.temperature)° Fahrenheit")

// Prints "The default temperature is 32.0° Fahrenheit"

### Default Property Values

You can set the initial value of a stored property from within an initializer, as shown above. Alternatively, specify a default property value as part of the property’s declaration. You specify a default property value by assigning an initial value to the property when it is defined.

NOTE

If a property always takes the same initial value, provide a default value rather than setting a value within an initializer. The end result is the same, but the default value ties the property’s initialization more closely to its declaration. It makes for shorter, clearer initializers and enables you to infer the type of the property from its default value. The default value also makes it easier for you to take advantage of default initializers and initializer inheritance, as described later in this chapter.

You can write the Fahrenheit structure from above in a simpler form by providing a default value for its temperature property at the point that the property is declared:

struct Fahrenheit {

var temperature = 32.0

}

### Parameter Names and Argument Labels

As with function and method parameters, initialization parameters can have both a parameter name for use within the initializer’s body and an argument label for use when calling the initializer.

However, initializers do not have an identifying function name before their parentheses in the way that functions and methods do. Therefore, the names and types of an initializer’s parameters play a particularly important role in identifying which initializer should be called. Because of this, Swift provides an automatic argument label for every parameter in an initializer if you don’t provide one.

The following example defines a structure called Color, with three constant properties called red, green, and blue. These properties store a value between 0.0 and 1.0 to indicate the amount of red, green, and blue in the color.

Color provides an initializer with three appropriately named parameters of type Double for its red, green, and blue components. Color also provides a second initializer with a single white parameter, which is used to provide the same value for all three color components.

1. struct Color {
2. let red, green, blue: Double
3. init(red: Double, green: Double, blue: Double) {
4. self.red = red
5. self.green = green
6. self.blue = blue
7. }
8. init(white: Double) {
9. red = white
10. green = white
11. blue = white
12. }
13. }

Both initializers can be used to create a new Color instance, by providing named values for each initializer parameter:

1. let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
2. let halfGray = Color(white: 0.5)

Note that it is not possible to call these initializers without using argument labels. Argument labels must always be used in an initializer if they are defined, and omitting them is a compile-time error:

1. let veryGreen = Color(0.0, 1.0, 0.0)
2. // this reports a compile-time error - argument labels are required

### Initializer Parameters Without Argument Labels

If you do not want to use an argument label for an initializer parameter, write an underscore (\_) instead of an explicit argument label for that parameter to override the default behavior.

Here’s an expanded version of the Celsius example from [Initialization Parameters](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID208) above, with an additional initializer to create a new Celsius instance from a Double value that is already in the Celsius scale:

1. struct Celsius {
2. var temperatureInCelsius: Double
3. init(fromFahrenheit fahrenheit: Double) {
4. temperatureInCelsius = (fahrenheit - 32.0) / 1.8
5. }
6. init(fromKelvin kelvin: Double) {
7. temperatureInCelsius = kelvin - 273.15
8. }
9. init(\_ celsius: Double) {
10. temperatureInCelsius = celsius
11. }
12. }
13. let bodyTemperature = Celsius(37.0)
14. // bodyTemperature.temperatureInCelsius is 37.0

The initializer call Celsius(37.0) is clear in its intent without the need for an argument label. It is therefore appropriate to write this initializer as init(\_ celsius: Double) so that it can be called by providing an unnamed Double value.

### Optional Property Types

If your custom type has a stored property that is logically allowed to have “no value”—perhaps because its value cannot be set during initialization, or because it is allowed to have “no value” at some later point—declare the property with an optional type. Properties of optional type are automatically initialized with a value of nil, indicating that the property is deliberately intended to have “no value yet” during initialization.

The following example defines a class called SurveyQuestion, with an optional String property called response:

class SurveyQuestion {

var text: String

var response: String?

init(text: String) {

self.text = text

}

func ask() {

print(text)

}

}

let cheeseQuestion = SurveyQuestion(text: "Do you like cheese?")

cheeseQuestion.ask()

// Prints "Do you like cheese?"

cheeseQuestion.response = "Yes, I do like cheese."

The response to a survey question cannot be known until it is asked, and so the response property is declared with a type of String?, or “optional String”. It is automatically assigned a default value of nil, meaning “no string yet”, when a new instance of SurveyQuestion is initialized.

### Assigning Constant Properties During Initialization

You can assign a value to a constant property at any point during initialization, as long as it is set to a definite value by the time initialization finishes. Once a constant property is assigned a value, it can’t be further modified.

NOTE

For class instances, a constant property can be modified during initialization only by the class that introduces it. It cannot be modified by a subclass.

You can revise the SurveyQuestion example from above to use a constant property rather than a variable property for the text property of the question, to indicate that the question does not change once an instance of SurveyQuestion is created. Even though the text property is now a constant, it can still be set within the class’s initializer:

class SurveyQuestion {

let text: String

var response: String?

init(text: String) {

self.text = text

}

func ask() {

print(text)

}

}

let beetsQuestion = SurveyQuestion(text: "How about beets?")

beetsQuestion.ask()

// Prints "How about beets?"

beetsQuestion.response = "I also like beets. (But not with cheese.)"

## Default Initializers

Swift provides a default initializer for any structure or class that provides default values for all of its properties and does not provide at least one initializer itself. The default initializer simply creates a new instance with all of its properties set to their default values.

This example defines a class called ShoppingListItem, which encapsulates the name, quantity, and purchase state of an item in a shopping list:

class ShoppingListItem {

var name: String?

var quantity = 1

var purchased = false

}

var item = ShoppingListItem()

Because all properties of the ShoppingListItem class have default values, and because it is a base class with no superclass, ShoppingListItem automatically gains a default initializer implementation that creates a new instance with all of its properties set to their default values. (The name property is an optional String property, and so it automatically receives a default value of nil, even though this value is not written in the code.) The example above uses the default initializer for the ShoppingListItem class to create a new instance of the class with initializer syntax, written as ShoppingListItem(), and assigns this new instance to a variable called item.

### Memberwise Initializers for Structure Types

Structure types automatically receive a memberwise initializer if they don’t define any of their own custom initializers. Unlike a default initializer, the structure receives a memberwise initializer even if it has stored properties that don’t have default values.

The memberwise initializer is a shorthand way 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.

The example below defines a structure called Size with two properties called width and height. Both properties are inferred to be of type Double by assigning a default value of 0.0.

The Size structure automatically receives an init(width:height:) memberwise initializer, which you can use to initialize a new Size instance:

struct Size {

var width = 0.0, height = 0.0

}

let twoByTwo = Size(width: 2.0, height: 2.0)

When you call a memberwise initializer, you can omit values for any properties that have default values. In the example above, the Size structure has a default value for both its height and width properties. You can omit either property or both properties, and the initializer uses the default value for anything you omit—for example:

let zeroByTwo = Size(height: 2.0)

print(zeroByTwo.width, zeroByTwo.height)

// Prints "0.0 2.0"

let zeroByZero = Size()

print(zeroByZero.width, zeroByZero.height)

// Prints "0.0 0.0"

## Initializer Delegation for Value Types

Initializers can call other initializers to perform part of an instance’s initialization. This process, known as initializer delegation, avoids duplicating code across multiple initializers.

The rules for how initializer delegation works, and for what forms of delegation are allowed, are different for value types and class types. Value types (structures and enumerations) do not support inheritance, and so their initializer delegation process is relatively simple, because they can only delegate to another initializer that they provide themselves. Classes, however, can inherit from other classes, as described in [Inheritance](https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html). This means that classes have additional responsibilities for ensuring that all stored properties they inherit are assigned a suitable value during initialization. These responsibilities are described in [Class Inheritance and Initialization](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID216) below.

For value types, you use self.init to refer to other initializers from the same value type when writing your own custom initializers. You can call self.init only from within an initializer.

Note that if you define a custom initializer for a value type, you will no longer have access to the default initializer (or the memberwise initializer, if it is a structure) for that type. This constraint prevents a situation in which additional essential setup provided in a more complex initializer is accidentally circumvented by someone using one of the automatic initializers.

NOTE

If you want your custom value type to be initializable with the default initializer and memberwise initializer, and also with your own custom initializers, write your custom initializers in an extension rather than as part of the value type’s original implementation. For more information, see [Extensions](https://docs.swift.org/swift-book/LanguageGuide/Extensions.html).

The following example defines a custom Rect structure to represent a geometric rectangle. The example requires two supporting structures called Size and Point, both of which provide default values of 0.0 for all of their properties:

1. struct Size {
2. var width = 0.0, height = 0.0
3. }
4. struct Point {
5. var x = 0.0, y = 0.0
6. }

You can initialize the Rect structure below in one of three ways—by using its default zero-initialized origin and size property values, by providing a specific origin point and size, or by providing a specific center point and size. These initialization options are represented by three custom initializers that are part of the Rect structure’s definition:

1. struct Rect {
2. var origin = Point()
3. var size = Size()
4. init() {}
5. init(origin: Point, size: Size) {
6. self.origin = origin
7. self.size = size
8. }
9. init(center: Point, size: Size) {
10. let originX = center.x - (size.width / 2)
11. let originY = center.y - (size.height / 2)
12. self.init(origin: Point(x: originX, y: originY), size: size)
13. }
14. }

The first Rect initializer, init(), is functionally the same as the default initializer that the structure would have received if it did not have its own custom initializers. This initializer has an empty body, represented by an empty pair of curly braces {}. Calling this initializer returns a Rect instance whose origin and size properties are both initialized with the default values of Point(x: 0.0, y: 0.0) and Size(width: 0.0, height: 0.0) from their property definitions:

1. let basicRect = Rect()
2. // basicRect's origin is (0.0, 0.0) and its size is (0.0, 0.0)

The second Rect initializer, init(origin:size:), is functionally the same as the memberwise initializer that the structure would have received if it did not have its own custom initializers. This initializer simply assigns the origin and size argument values to the appropriate stored properties:

1. let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
2. size: Size(width: 5.0, height: 5.0))
3. // originRect's origin is (2.0, 2.0) and its size is (5.0, 5.0)

The third Rect initializer, init(center:size:), is slightly more complex. It starts by calculating an appropriate origin point based on a center point and a size value. It then calls (or delegates) to the init(origin:size:) initializer, which stores the new origin and size values in the appropriate properties:

1. let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
2. size: Size(width: 3.0, height: 3.0))
3. // centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)

The init(center:size:) initializer could have assigned the new values of origin and size to the appropriate properties itself. However, it is more convenient (and clearer in intent) for the init(center:size:) initializer to take advantage of an existing initializer that already provides exactly that functionality.

NOTE

For an alternative way to write this example without defining the init() and init(origin:size:) initializers yourself, see [Extensions](https://docs.swift.org/swift-book/LanguageGuide/Extensions.html).

## Class Inheritance and Initialization

All of a class’s stored properties—including any properties the class inherits from its superclass—must be assigned an initial value during initialization.

Swift defines two kinds of initializers for class types to help ensure all stored properties receive an initial value. These are known as designated initializers and convenience initializers.

### Designated Initializers and Convenience Initializers

Designated initializers are the primary initializers for a class. A designated initializer fully initializes all properties introduced by that class and calls an appropriate superclass initializer to continue the initialization process up the superclass chain.

Classes tend to have very few designated initializers, and it is quite common for a class to have only one. Designated initializers are “funnel” points through which initialization takes place, and through which the initialization process continues up the superclass chain.

Every class must have at least one designated initializer. In some cases, this requirement is satisfied by inheriting one or more designated initializers from a superclass, as described in [Automatic Initializer Inheritance](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID222) below.

Convenience initializers are secondary, supporting initializers for a class. You can define a convenience initializer to call a designated initializer from the same class as the convenience initializer with some of the designated initializer’s parameters set to default values. You can also define a convenience initializer to create an instance of that class for a specific use case or input value type.

You do not have to provide convenience initializers if your class does not require them. Create convenience initializers whenever a shortcut to a common initialization pattern will save time or make initialization of the class clearer in intent.

### Syntax for Designated and Convenience Initializers

Designated initializers for classes are written in the same way as simple initializers for value types:

1. init(parameters) {
2. statements
3. }

Convenience initializers are written in the same style, but with the convenience modifier placed before the init keyword, separated by a space:

1. convenience init(parameters) {
2. statements
3. }

### Initializer Delegation for Class Types

To simplify the relationships between designated and convenience initializers, Swift applies the following three rules for delegation calls between initializers:

**Rule 1**

A designated initializer must call a designated initializer from its immediate superclass.

**Rule 2**

A convenience initializer must call another initializer from the same class.

**Rule 3**

A convenience initializer must ultimately call a designated initializer.

A simple way to remember this is:

* Designated initializers must always delegate up.
* Convenience initializers must always delegate across.

These rules are illustrated in the figure below:

Here, the superclass has a single designated initializer and two convenience initializers. One convenience initializer calls another convenience initializer, which in turn calls the single designated initializer. This satisfies rules 2 and 3 from above. The superclass does not itself have a further superclass, and so rule 1 does not apply.

The subclass in this figure has two designated initializers and one convenience initializer. The convenience initializer must call one of the two designated initializers, because it can only call another initializer from the same class. This satisfies rules 2 and 3 from above. Both designated initializers must call the single designated initializer from the superclass, to satisfy rule 1 from above.

NOTE

These rules don’t affect how users of your classes create instances of each class. Any initializer in the diagram above can be used to create a fully initialized instance of the class they belong to. The rules only affect how you write the implementation of the class’s initializers.

The figure below shows a more complex class hierarchy for four classes. It illustrates how the designated initializers in this hierarchy act as “funnel” points for class initialization, simplifying the interrelationships among classes in the chain:

### Two-Phase Initialization

Class initialization in Swift is a two-phase process. In the first phase, each stored property is assigned an initial value by the class that introduced it. Once the initial state for every stored property has been determined, the second phase begins, and each class is given the opportunity to customize its stored properties further before the new instance is considered ready for use.

The use of a two-phase initialization process makes initialization safe, while still giving complete flexibility to each class in a class hierarchy. Two-phase initialization prevents property values from being accessed before they are initialized, and prevents property values from being set to a different value by another initializer unexpectedly.

NOTE

Swift’s two-phase initialization process is similar to initialization in Objective-C. The main difference is that during phase 1, Objective-C assigns zero or null values (such as 0 or nil) to every property. Swift’s initialization flow is more flexible in that it lets you set custom initial values, and can cope with types for which 0 or nil is not a valid default value.

Swift’s compiler performs four helpful safety-checks to make sure that two-phase initialization is completed without error:

**Safety check 1**

A designated initializer must ensure that all of the properties introduced by its class are initialized before it delegates up to a superclass initializer.

As mentioned above, the memory for an object is only considered fully initialized once the initial state of all of its stored properties is known. In order for this rule to be satisfied, a designated initializer must make sure that all of its own properties are initialized before it hands off up the chain.

**Safety check 2**

A designated initializer must delegate up to a superclass initializer before assigning a value to an inherited property. If it doesn’t, the new value the designated initializer assigns will be overwritten by the superclass as part of its own initialization.

**Safety check 3**

A convenience initializer must delegate to another initializer before assigning a value to any property (including properties defined by the same class). If it doesn’t, the new value the convenience initializer assigns will be overwritten by its own class’s designated initializer.

**Safety check 4**

An initializer cannot call any instance methods, read the values of any instance properties, or refer to self as a value until after the first phase of initialization is complete.

The class instance is not fully valid until the first phase ends. Properties can only be accessed, and methods can only be called, once the class instance is known to be valid at the end of the first phase.

Here’s how two-phase initialization plays out, based on the four safety checks above:

**Phase 1**

* A designated or convenience initializer is called on a class.
* Memory for a new instance of that class is allocated. The memory is not yet initialized.
* A designated initializer for that class confirms that all stored properties introduced by that class have a value. The memory for these stored properties is now initialized.
* The designated initializer hands off to a superclass initializer to perform the same task for its own stored properties.
* This continues up the class inheritance chain until the top of the chain is reached.
* Once the top of the chain is reached, and the final class in the chain has ensured that all of its stored properties have a value, the instance’s memory is considered to be fully initialized, and phase 1 is complete.

**Phase 2**

* Working back down from the top of the chain, each designated initializer in the chain has the option to customize the instance further. Initializers are now able to access self and can modify its properties, call its instance methods, and so on.
* Finally, any convenience initializers in the chain have the option to customize the instance and to work with self.

Here’s how phase 1 looks for an initialization call for a hypothetical subclass and superclass:

In this example, initialization begins with a call to a convenience initializer on the subclass. This convenience initializer cannot yet modify any properties. It delegates across to a designated initializer from the same class.

The designated initializer makes sure that all of the subclass’s properties have a value, as per safety check 1. It then calls a designated initializer on its superclass to continue the initialization up the chain.

The superclass’s designated initializer makes sure that all of the superclass properties have a value. There are no further superclasses to initialize, and so no further delegation is needed.

As soon as all properties of the superclass have an initial value, its memory is considered fully initialized, and phase 1 is complete.

Here’s how phase 2 looks for the same initialization call:

The superclass’s designated initializer now has an opportunity to customize the instance further (although it does not have to).

Once the superclass’s designated initializer is finished, the subclass’s designated initializer can perform additional customization (although again, it does not have to).

Finally, once the subclass’s designated initializer is finished, the convenience initializer that was originally called can perform additional customization.

### Initializer Inheritance and Overriding

Unlike subclasses in Objective-C, Swift subclasses do not inherit their superclass initializers by default. Swift’s approach prevents a situation in which a simple initializer from a superclass is inherited by a more specialized subclass and is used to create a new instance of the subclass that is not fully or correctly initialized.

NOTE

Superclass initializers are inherited in certain circumstances, but only when it is safe and appropriate to do so. For more information, see [Automatic Initializer Inheritance](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID222) below.

If you want a custom subclass to present one or more of the same initializers as its superclass, you can provide a custom implementation of those initializers within the subclass.

When you write a subclass initializer that matches a superclass designated initializer, you are effectively providing an override of that designated initializer. Therefore, you must write the override modifier before the subclass’s initializer definition. This is true even if you are overriding an automatically provided default initializer, as described in [Default Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID213).

As with an overridden property, method or subscript, the presence of the override modifier prompts Swift to check that the superclass has a matching designated initializer to be overridden, and validates that the parameters for your overriding initializer have been specified as intended.

NOTE

You always write the override modifier when overriding a superclass designated initializer, even if your subclass’s implementation of the initializer is a convenience initializer.

Conversely, if you write a subclass initializer that matches a superclass convenience initializer, that superclass convenience initializer can never be called directly by your subclass, as per the rules described above in [Initializer Delegation for Class Types](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID219). Therefore, your subclass is not (strictly speaking) providing an override of the superclass initializer. As a result, you do not write the override modifier when providing a matching implementation of a superclass convenience initializer.

The example below defines a base class called Vehicle. This base class declares a stored property called numberOfWheels, with a default Int value of 0. The numberOfWheels property is used by a computed property called description to create a String description of the vehicle’s characteristics:

1. class Vehicle {
2. var numberOfWheels = 0
3. var description: String {
4. return "\(numberOfWheels) wheel(s)"
5. }
6. }

The Vehicle class provides a default value for its only stored property, and does not provide any custom initializers itself. As a result, it automatically receives a default initializer, as described in [Default Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID213). The default initializer (when available) is always a designated initializer for a class, and can be used to create a new Vehicle instance with a numberOfWheels of 0:

1. let vehicle = Vehicle()
2. print("Vehicle: \(vehicle.description)")
3. // Vehicle: 0 wheel(s)

The next example defines a subclass of Vehicle called Bicycle:

1. class Bicycle: Vehicle {
2. override init() {
3. super.init()
4. numberOfWheels = 2
5. }
6. }

The Bicycle subclass defines a custom designated initializer, init(). This designated initializer matches a designated initializer from the superclass of Bicycle, and so the Bicycle version of this initializer is marked with the override modifier.

The init() initializer for Bicycle starts by calling super.init(), which calls the default initializer for the Bicycle class’s superclass, Vehicle. This ensures that the numberOfWheels inherited property is initialized by Vehicle before Bicycle has the opportunity to modify the property. After calling super.init(), the original value of numberOfWheels is replaced with a new value of 2.

If you create an instance of Bicycle, you can call its inherited description computed property to see how its numberOfWheels property has been updated:

1. let bicycle = Bicycle()
2. print("Bicycle: \(bicycle.description)")
3. // Bicycle: 2 wheel(s)

If a subclass initializer performs no customization in phase 2 of the initialization process, and the superclass has a zero-argument designated initializer, you can omit a call to super.init() after assigning values to all of the subclass’s stored properties.

This example defines another subclass of Vehicle, called Hoverboard. In its initializer, the Hoverboard class sets only its color property. Instead of making an explicit call to super.init(), this initializer relies on an implicit call to its superclass’s initializer to complete the process.

1. class Hoverboard: Vehicle {
2. var color: String
3. init(color: String) {
4. self.color = color
5. // super.init() implicitly called here
6. }
7. override var description: String {
8. return "\(super.description) in a beautiful \(color)"
9. }
10. }

An instance of Hoverboard uses the default number of wheels supplied by the Vehicle initializer.

1. let hoverboard = Hoverboard(color: "silver")
2. print("Hoverboard: \(hoverboard.description)")
3. // Hoverboard: 0 wheel(s) in a beautiful silver

NOTE

Subclasses can modify inherited variable properties during initialization, but can not modify inherited constant properties.

### Automatic Initializer Inheritance

As mentioned above, subclasses do not inherit their superclass initializers by default. However, superclass initializers are automatically inherited if certain conditions are met. In practice, this means that you do not need to write initializer overrides in many common scenarios, and can inherit your superclass initializers with minimal effort whenever it is safe to do so.

Assuming that you provide default values for any new properties you introduce in a subclass, the following two rules apply:

**Rule 1**

If your subclass doesn’t define any designated initializers, it automatically inherits all of its superclass designated initializers.

**Rule 2**

If your subclass provides an implementation of all of its superclass designated initializers—either by inheriting them as per rule 1, or by providing a custom implementation as part of its definition—then it automatically inherits all of the superclass convenience initializers.

These rules apply even if your subclass adds further convenience initializers.

NOTE

A subclass can implement a superclass designated initializer as a subclass convenience initializer as part of satisfying rule 2.

### Designated and Convenience Initializers in Action

The following example shows designated initializers, convenience initializers, and automatic initializer inheritance in action. This example defines a hierarchy of three classes called Food, RecipeIngredient, and ShoppingListItem, and demonstrates how their initializers interact.

The base class in the hierarchy is called Food, which is a simple class to encapsulate the name of a foodstuff. The Food class introduces a single String property called name and provides two initializers for creating Food instances:

1. class Food {
2. var name: String
3. init(name: String) {
4. self.name = name
5. }
6. convenience init() {
7. self.init(name: "[Unnamed]")
8. }
9. }

The figure below shows the initializer chain for the Food class:

Classes do not have a default memberwise initializer, and so the Food class provides a designated initializer that takes a single argument called name. This initializer can be used to create a new Food instance with a specific name:

1. let namedMeat = Food(name: "Bacon")
2. // namedMeat's name is "Bacon"

The init(name: String) initializer from the Food class is provided as a designated initializer, because it ensures that all stored properties of a new Food instance are fully initialized. The Food class does not have a superclass, and so the init(name: String) initializer does not need to call super.init() to complete its initialization.

The Food class also provides a convenience initializer, init(), with no arguments. The init() initializer provides a default placeholder name for a new food by delegating across to the Food class’s init(name: String) with a name value of [Unnamed]:

1. let mysteryMeat = Food()
2. // mysteryMeat's name is "[Unnamed]"

The second class in the hierarchy is a subclass of Food called RecipeIngredient. The RecipeIngredient class models an ingredient in a cooking recipe. It introduces an Int property called quantity (in addition to the name property it inherits from Food) and defines two initializers for creating RecipeIngredient instances:

1. class RecipeIngredient: Food {
2. var quantity: Int
3. init(name: String, quantity: Int) {
4. self.quantity = quantity
5. super.init(name: name)
6. }
7. override convenience init(name: String) {
8. self.init(name: name, quantity: 1)
9. }
10. }

The figure below shows the initializer chain for the RecipeIngredient class:

The RecipeIngredient class has a single designated initializer, init(name: String, quantity: Int), which can be used to populate all of the properties of a new RecipeIngredient instance. This initializer starts by assigning the passed quantity argument to the quantity property, which is the only new property introduced by RecipeIngredient. After doing so, the initializer delegates up to the init(name: String) initializer of the Food class. This process satisfies safety check 1 from [Two-Phase Initialization](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID220) above.

RecipeIngredient also defines a convenience initializer, init(name: String), which is used to create a RecipeIngredient instance by name alone. This convenience initializer assumes a quantity of 1 for any RecipeIngredient instance that is created without an explicit quantity. The definition of this convenience initializer makes RecipeIngredient instances quicker and more convenient to create, and avoids code duplication when creating several single-quantity RecipeIngredient instances. This convenience initializer simply delegates across to the class’s designated initializer, passing in a quantity value of 1.

The init(name: String) convenience initializer provided by RecipeIngredient takes the same parameters as the init(name: String) designated initializer from Food. Because this convenience initializer overrides a designated initializer from its superclass, it must be marked with the override modifier (as described in [Initializer Inheritance and Overriding](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID221)).

Even though RecipeIngredient provides the init(name: String) initializer as a convenience initializer, RecipeIngredient has nonetheless provided an implementation of all of its superclass’s designated initializers. Therefore, RecipeIngredient automatically inherits all of its superclass’s convenience initializers too.

In this example, the superclass for RecipeIngredient is Food, which has a single convenience initializer called init(). This initializer is therefore inherited by RecipeIngredient. The inherited version of init() functions in exactly the same way as the Food version, except that it delegates to the RecipeIngredient version of init(name: String) rather than the Food version.

All three of these initializers can be used to create new RecipeIngredient instances:

1. let oneMysteryItem = RecipeIngredient()
2. let oneBacon = RecipeIngredient(name: "Bacon")
3. let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)

The third and final class in the hierarchy is a subclass of RecipeIngredient called ShoppingListItem. The ShoppingListItem class models a recipe ingredient as it appears in a shopping list.

Every item in the shopping list starts out as “unpurchased”. To represent this fact, ShoppingListItem introduces a Boolean property called purchased, with a default value of false. ShoppingListItem also adds a computed description property, which provides a textual description of a ShoppingListItem instance:

1. class ShoppingListItem: RecipeIngredient {
2. var purchased = false
3. var description: String {
4. var output = "\(quantity) x \(name)"
5. output += purchased ? " ✔" : " ✘"
6. return output
7. }
8. }

NOTE

ShoppingListItem does not define an initializer to provide an initial value for purchased, because items in a shopping list (as modeled here) always start out unpurchased.

Because it provides a default value for all of the properties it introduces and does not define any initializers itself, ShoppingListItem automatically inherits all of the designated and convenience initializers from its superclass.

The figure below shows the overall initializer chain for all three classes:

You can use all three of the inherited initializers to create a new ShoppingListItem instance:

1. var breakfastList = [
2. ShoppingListItem(),
3. ShoppingListItem(name: "Bacon"),
4. ShoppingListItem(name: "Eggs", quantity: 6),
5. ]
6. breakfastList[0].name = "Orange juice"
7. breakfastList[0].purchased = true
8. for item in breakfastList {
9. print(item.description)
10. }
11. // 1 x Orange juice ✔
12. // 1 x Bacon ✘
13. // 6 x Eggs ✘

Here, a new array called breakfastList is created from an array literal containing three new ShoppingListItem instances. The type of the array is inferred to be [ShoppingListItem]. After the array is created, the name of the ShoppingListItem at the start of the array is changed from "[Unnamed]" to "Orange juice" and it is marked as having been purchased. Printing the description of each item in the array shows that their default states have been set as expected.

## Failable Initializers

It is sometimes useful to define a class, structure, or enumeration for which initialization can fail. This failure might be triggered by invalid initialization parameter values, the absence of a required external resource, or some other condition that prevents initialization from succeeding.

To cope with initialization conditions that can fail, define one or more failable initializers as part of a class, structure, or enumeration definition. You write a failable initializer by placing a question mark after the init keyword (init?).

NOTE

You cannot define a failable and a nonfailable initializer with the same parameter types and names.

A failable initializer creates an optional value of the type it initializes. You write return nil within a failable initializer to indicate a point at which initialization failure can be triggered.

NOTE

Strictly speaking, initializers do not return a value. Rather, their role is to ensure that self is fully and correctly initialized by the time that initialization ends. Although you write return nil to trigger an initialization failure, you do not use the return keyword to indicate initialization success.

For instance, failable initializers are implemented for numeric type conversions. To ensure conversion between numeric types maintains the value exactly, use the init(exactly:) initializer. If the type conversion cannot maintain the value, the initializer fails.

1. let wholeNumber: Double = 12345.0
2. let pi = 3.14159
3. if let valueMaintained = Int(exactly: wholeNumber) {
4. print("\(wholeNumber) conversion to Int maintains value of \(valueMaintained)")
5. }
6. // Prints "12345.0 conversion to Int maintains value of 12345"
7. let valueChanged = Int(exactly: pi)
8. // valueChanged is of type Int?, not Int
9. if valueChanged == nil {
10. print("\(pi) conversion to Int does not maintain value")
11. }
12. // Prints "3.14159 conversion to Int does not maintain value"

The example below defines a structure called Animal, with a constant String property called species. The Animal structure also defines a failable initializer with a single parameter called species. This initializer checks if the species value passed to the initializer is an empty string. If an empty string is found, an initialization failure is triggered. Otherwise, the species property’s value is set, and initialization succeeds:

1. struct Animal {
2. let species: String
3. init?(species: String) {
4. if species.isEmpty { return nil }
5. self.species = species
6. }
7. }

You can use this failable initializer to try to initialize a new Animal instance and to check if initialization succeeded:

1. let someCreature = Animal(species: "Giraffe")
2. // someCreature is of type Animal?, not Animal
3. if let giraffe = someCreature {
4. print("An animal was initialized with a species of \(giraffe.species)")
5. }
6. // Prints "An animal was initialized with a species of Giraffe"

If you pass an empty string value to the failable initializer’s species parameter, the initializer triggers an initialization failure:

1. let anonymousCreature = Animal(species: "")
2. // anonymousCreature is of type Animal?, not Animal
3. if anonymousCreature == nil {
4. print("The anonymous creature could not be initialized")
5. }
6. // Prints "The anonymous creature could not be initialized"

NOTE

Checking for an empty string value (such as "" rather than "Giraffe") is not the same as checking for nil to indicate the absence of an optional String value. In the example above, an empty string ("") is a valid, non-optional String. However, it is not appropriate for an animal to have an empty string as the value of its species property. To model this restriction, the failable initializer triggers an initialization failure if an empty string is found.

### Failable Initializers for Enumerations

You can use a failable initializer to select an appropriate enumeration case based on one or more parameters. The initializer can then fail if the provided parameters do not match an appropriate enumeration case.

The example below defines an enumeration called TemperatureUnit, with three possible states (kelvin, celsius, and fahrenheit). A failable initializer is used to find an appropriate enumeration case for a Character value representing a temperature symbol:

1. enum TemperatureUnit {
2. case kelvin, celsius, fahrenheit
3. init?(symbol: Character) {
4. switch symbol {
5. case "K":
6. self = .kelvin
7. case "C":
8. self = .celsius
9. case "F":
10. self = .fahrenheit
11. default:
12. return nil
13. }
14. }
15. }

You can use this failable initializer to choose an appropriate enumeration case for the three possible states and to cause initialization to fail if the parameter does not match one of these states:

1. let fahrenheitUnit = TemperatureUnit(symbol: "F")
2. if fahrenheitUnit != nil {
3. print("This is a defined temperature unit, so initialization succeeded.")
4. }
5. // Prints "This is a defined temperature unit, so initialization succeeded."
6. let unknownUnit = TemperatureUnit(symbol: "X")
7. if unknownUnit == nil {
8. print("This is not a defined temperature unit, so initialization failed.")
9. }
10. // Prints "This is not a defined temperature unit, so initialization failed."

### Failable Initializers for Enumerations with Raw Values

Enumerations with raw values automatically receive a failable initializer, init?(rawValue:), that takes a parameter called rawValue of the appropriate raw-value type and selects a matching enumeration case if one is found, or triggers an initialization failure if no matching value exists.

You can rewrite the TemperatureUnit example from above to use raw values of type Character and to take advantage of the init?(rawValue:) initializer:

1. enum TemperatureUnit: Character {
2. case kelvin = "K", celsius = "C", fahrenheit = "F"
3. }
4. let fahrenheitUnit = TemperatureUnit(rawValue: "F")
5. if fahrenheitUnit != nil {
6. print("This is a defined temperature unit, so initialization succeeded.")
7. }
8. // Prints "This is a defined temperature unit, so initialization succeeded."
9. let unknownUnit = TemperatureUnit(rawValue: "X")
10. if unknownUnit == nil {
11. print("This is not a defined temperature unit, so initialization failed.")
12. }
13. // Prints "This is not a defined temperature unit, so initialization failed."

### Propagation of Initialization Failure

A failable initializer of a class, structure, or enumeration can delegate across to another failable initializer from the same class, structure, or enumeration. Similarly, a subclass failable initializer can delegate up to a superclass failable initializer.

In either case, if you delegate to another initializer that causes initialization to fail, the entire initialization process fails immediately, and no further initialization code is executed.

NOTE

A failable initializer can also delegate to a nonfailable initializer. Use this approach if you need to add a potential failure state to an existing initialization process that does not otherwise fail.

The example below defines a subclass of Product called CartItem. The CartItem class models an item in an online shopping cart. CartItem introduces a stored constant property called quantity and ensures that this property always has a value of at least 1:

1. class Product {
2. let name: String
3. init?(name: String) {
4. if name.isEmpty { return nil }
5. self.name = name
6. }
7. }
8. class CartItem: Product {
9. let quantity: Int
10. init?(name: String, quantity: Int) {
11. if quantity < 1 { return nil }
12. self.quantity = quantity
13. super.init(name: name)
14. }
15. }

The failable initializer for CartItem starts by validating that it has received a quantity value of 1 or more. If the quantity is invalid, the entire initialization process fails immediately and no further initialization code is executed. Likewise, the failable initializer for Product checks the name value, and the initializer process fails immediately if name is the empty string.

If you create a CartItem instance with a nonempty name and a quantity of 1 or more, initialization succeeds:

1. if let twoSocks = CartItem(name: "sock", quantity: 2) {
2. print("Item: \(twoSocks.name), quantity: \(twoSocks.quantity)")
3. }
4. // Prints "Item: sock, quantity: 2"

If you try to create a CartItem instance with a quantity value of 0, the CartItem initializer causes initialization to fail:

1. if let zeroShirts = CartItem(name: "shirt", quantity: 0) {
2. print("Item: \(zeroShirts.name), quantity: \(zeroShirts.quantity)")
3. } else {
4. print("Unable to initialize zero shirts")
5. }
6. // Prints "Unable to initialize zero shirts"

Similarly, if you try to create a CartItem instance with an empty name value, the superclass Product initializer causes initialization to fail:

1. if let oneUnnamed = CartItem(name: "", quantity: 1) {
2. print("Item: \(oneUnnamed.name), quantity: \(oneUnnamed.quantity)")
3. } else {
4. print("Unable to initialize one unnamed product")
5. }
6. // Prints "Unable to initialize one unnamed product"

### Overriding a Failable Initializer

You can override a superclass failable initializer in a subclass, just like any other initializer. Alternatively, you can override a superclass failable initializer with a subclass nonfailable initializer. This enables you to define a subclass for which initialization cannot fail, even though initialization of the superclass is allowed to fail.

Note that if you override a failable superclass initializer with a nonfailable subclass initializer, the only way to delegate up to the superclass initializer is to force-unwrap the result of the failable superclass initializer.

NOTE

You can override a failable initializer with a nonfailable initializer but not the other way around.

The example below defines a class called Document. This class models a document that can be initialized with a name property that is either a nonempty string value or nil, but cannot be an empty string:

1. class Document {
2. var name: String?
3. // this initializer creates a document with a nil name value
4. init() {}
5. // this initializer creates a document with a nonempty name value
6. init?(name: String) {
7. if name.isEmpty { return nil }
8. self.name = name
9. }
10. }

The next example defines a subclass of Document called AutomaticallyNamedDocument. The AutomaticallyNamedDocument subclass overrides both of the designated initializers introduced by Document. These overrides ensure that an AutomaticallyNamedDocument instance has an initial name value of "[Untitled]" if the instance is initialized without a name, or if an empty string is passed to the init(name:) initializer:

1. class AutomaticallyNamedDocument: Document {
2. override init() {
3. super.init()
4. self.name = "[Untitled]"
5. }
6. override init(name: String) {
7. super.init()
8. if name.isEmpty {
9. self.name = "[Untitled]"
10. } else {
11. self.name = name
12. }
13. }
14. }

The AutomaticallyNamedDocument overrides its superclass’s failable init?(name:) initializer with a nonfailable init(name:) initializer. Because AutomaticallyNamedDocument copes with the empty string case in a different way than its superclass, its initializer does not need to fail, and so it provides a nonfailable version of the initializer instead.

You can use forced unwrapping in an initializer to call a failable initializer from the superclass as part of the implementation of a subclass’s nonfailable initializer. For example, the UntitledDocument subclass below is always named "[Untitled]", and it uses the failable init(name:) initializer from its superclass during initialization.

1. class UntitledDocument: Document {
2. override init() {
3. super.init(name: "[Untitled]")!
4. }
5. }

In this case, if the init(name:) initializer of the superclass were ever called with an empty string as the name, the forced unwrapping operation would result in a runtime error. However, because it’s called with a string constant, you can see that the initializer won’t fail, so no runtime error can occur in this case.

### The init! Failable Initializer

You typically define a failable initializer that creates an optional instance of the appropriate type by placing a question mark after the init keyword (init?). Alternatively, you can define a failable initializer that creates an implicitly unwrapped optional instance of the appropriate type. Do this by placing an exclamation mark after the init keyword (init!) instead of a question mark.

You can delegate from init? to init! and vice versa, and you can override init? with init! and vice versa. You can also delegate from init to init!, although doing so will trigger an assertion if the init! initializer causes initialization to fail.

## Required Initializers

Write the required modifier before the definition of a class initializer to indicate that every subclass of the class must implement that initializer:

1. class SomeClass {
2. required init() {
3. // initializer implementation goes here
4. }
5. }

You must also write the required modifier before every subclass implementation of a required initializer, to indicate that the initializer requirement applies to further subclasses in the chain. You do not write the override modifier when overriding a required designated initializer:

1. class SomeSubclass: SomeClass {
2. required init() {
3. // subclass implementation of the required initializer goes here
4. }
5. }

NOTE

You do not have to provide an explicit implementation of a required initializer if you can satisfy the requirement with an inherited initializer.

## Setting a Default Property Value with a Closure or Function

If a stored property’s default value requires some customization or setup, you can use a closure or global function to provide a customized default value for that property. Whenever a new instance of the type that the property belongs to is initialized, the closure or function is called, and its return value is assigned as the property’s default value.

These kinds of closures or functions typically create a temporary value of the same type as the property, tailor that value to represent the desired initial state, and then return that temporary value to be used as the property’s default value.

Here’s a skeleton outline of how a closure can be used to provide a default property value:

1. class SomeClass {
2. let someProperty: SomeType = {
3. // create a default value for someProperty inside this closure
4. // someValue must be of the same type as SomeType
5. return someValue
6. }()
7. }

Note that the closure’s end curly brace is followed by an empty pair of parentheses. This tells Swift to execute the closure immediately. If you omit these parentheses, you are trying to assign the closure itself to the property, and not the return value of the closure.

NOTE

If you use a closure to initialize a property, remember that the rest of the instance has not yet been initialized at the point that the closure is executed. This means that you cannot access any other property values from within your closure, even if those properties have default values. You also cannot use the implicit self property, or call any of the instance’s methods.

The example below defines a structure called Chessboard, which models a board for the game of chess. Chess is played on an 8 x 8 board, with alternating black and white squares.

To represent this game board, the Chessboard structure has a single property called boardColors, which is an array of 64 Bool values. A value of true in the array represents a black square and a value of false represents a white square. The first item in the array represents the top left square on the board and the last item in the array represents the bottom right square on the board.

The boardColors array is initialized with a closure to set up its color values:

1. struct Chessboard {
2. let boardColors: [Bool] = {
3. var temporaryBoard = [Bool]()
4. var isBlack = false
5. for i in 1...8 {
6. for j in 1...8 {
7. temporaryBoard.append(isBlack)
8. isBlack = !isBlack
9. }
10. isBlack = !isBlack
11. }
12. return temporaryBoard
13. }()
14. func squareIsBlackAt(row: Int, column: Int) -> Bool {
15. return boardColors[(row \* 8) + column]
16. }
17. }

Whenever a new Chessboard instance is created, the closure is executed, and the default value of boardColors is calculated and returned. The closure in the example above calculates and sets the appropriate color for each square on the board in a temporary array called temporaryBoard, and returns this temporary array as the closure’s return value once its setup is complete. The returned array value is stored in boardColors and can be queried with the squareIsBlackAt(row:column:) utility function:

1. let board = Chessboard()
2. print(board.squareIsBlackAt(row: 0, column: 1))
3. // Prints "true"
4. print(board.squareIsBlackAt(row: 7, column: 7))
5. // Prints "false"

# Deinitialization

A deinitializer is called immediately before a class instance is deallocated. You write deinitializers with the deinit keyword.

. However, when you are working with your own resources, you might need to perform some additional cleanup yourself. For example, if you create a custom class to open a file and write some data to it, you might need to close the file before the class instance is deallocated.

Class definitions can have at most one deinitializer per class. The deinitializer does not take any parameters and is written without parentheses:

deinit {

// perform the deinitialization

}

Superclass deinitializers are inherited by their subclasses, and the superclass deinitializer is called automatically at the end of a subclass deinitializer implementation. Superclass deinitializers are always called, even if a subclass does not provide its own deinitializer.

Because an instance is not deallocated until after its deinitializer is called, a deinitializer can access all properties of the instance it is called on and can modify its behavior based on those properties (such as looking up the name of a file that needs to be closed).

## Deinitializers in Action

Here’s an example of a deinitializer in action. This example defines two new types, Bank and Player, for a simple game. The Bank class manages a made-up currency, which can never have more than 10,000 coins in circulation. There can only ever be one Bank in the game, and so the Bank is implemented as a class with type properties and methods to store and manage its current state:

1. class Bank {
2. static var coinsInBank = 10\_000
3. static func distribute(coins numberOfCoinsRequested: Int) -> Int {
4. let numberOfCoinsToVend = min(numberOfCoinsRequested, coinsInBank)
5. coinsInBank -= numberOfCoinsToVend
6. return numberOfCoinsToVend
7. }
8. static func receive(coins: Int) {
9. coinsInBank += coins
10. }
11. }

Bank keeps track of the current number of coins it holds with its coinsInBank property. It also offers two methods—distribute(coins:) and receive(coins:)—to handle the distribution and collection of coins.

The distribute(coins:) method checks that there are enough coins in the bank before distributing them. If there are not enough coins, Bank returns a smaller number than the number that was requested (and returns zero if no coins are left in the bank). It returns an integer value to indicate the actual number of coins that were provided.

The receive(coins:) method simply adds the received number of coins back into the bank’s coin store.

The Player class describes a player in the game. Each player has a certain number of coins stored in their purse at any time. This is represented by the player’s coinsInPurse property:

1. class Player {
2. var coinsInPurse: Int
3. init(coins: Int) {
4. coinsInPurse = Bank.distribute(coins: coins)
5. }
6. func win(coins: Int) {
7. coinsInPurse += Bank.distribute(coins: coins)
8. }
9. deinit {
10. Bank.receive(coins: coinsInPurse)
11. }
12. }

Each Player instance is initialized with a starting allowance of a specified number of coins from the bank during initialization, although a Player instance may receive fewer than that number if not enough coins are available.

The Player class defines a win(coins:) method, which retrieves a certain number of coins from the bank and adds them to the player’s purse. The Player class also implements a deinitializer, which is called just before a Player instance is deallocated. Here, the deinitializer simply returns all of the player’s coins to the bank:

1. var playerOne: Player? = Player(coins: 100)
2. print("A new player has joined the game with \(playerOne!.coinsInPurse) coins")
3. // Prints "A new player has joined the game with 100 coins"
4. print("There are now \(Bank.coinsInBank) coins left in the bank")
5. // Prints "There are now 9900 coins left in the bank"

A new Player instance is created, with a request for 100 coins if they are available. This Player instance is stored in an optional Player variable called playerOne. An optional variable is used here, because players can leave the game at any point. The optional lets you track whether there is currently a player in the game.

Because playerOne is an optional, it is qualified with an exclamation mark (!) when its coinsInPurse property is accessed to print its default number of coins, and whenever its win(coins:) method is called:

playerOne!.win(coins: 2\_000)

print("PlayerOne won 2000 coins & now has \(playerOne!.coinsInPurse) coins")

// Prints "PlayerOne won 2000 coins & now has 2100 coins"

print("The bank now only has \(Bank.coinsInBank) coins left")

// Prints "The bank now only has 7900 coins left"

Here, the player has won 2,000 coins. The player’s purse now contains 2,100 coins, and the bank has only 7,900 coins left.

playerOne = nil

print("PlayerOne has left the game")

// Prints "PlayerOne has left the game"

print("The bank now has \(Bank.coinsInBank) coins")

// Prints "The bank now has 10000 coins"

The player has now left the game. This is indicated by setting the optional playerOne variable to nil, meaning “no Player instance.” At the point that this happens, the playerOne variable’s reference to the Player instance is broken. No other properties or variables are still referring to the Player instance, and so it is deallocated in order to free up its memory. Just before this happens, its deinitializer is called automatically, and its coins are returned to the bank.

# Optional Chaining

Optional chaining is a process for querying and calling properties, methods, and subscripts on an optional that might currently be nil. If the optional contains a value, the property, method, or subscript call succeeds; if the optional is nil, the property, method, or subscript call returns nil. Multiple queries can be chained together, and the entire chain fails gracefully if any link in the chain is nil.

NOTE

Optional chaining in Swift is similar to messaging nil in Objective-C, but in a way that works for any type, and that can be checked for success or failure.

## Optional Chaining as an Alternative to Forced Unwrapping

You specify optional chaining by placing a question mark (?) after the optional value on which you wish to call a property, method or subscript if the optional is non-nil. This is very similar to placing an exclamation mark (!) after an optional value to force the unwrapping of its value. The main difference is that optional chaining fails gracefully when the optional is nil, whereas forced unwrapping triggers a runtime error when the optional is nil.

To reflect the fact that optional chaining can be called on a nil value, the result of an optional chaining call is always an optional value, even if the property, method, or subscript you are querying returns a non-optional value. You can use this optional return value to check whether the optional chaining call was successful (the returned optional contains a value), or did not succeed due to a nil value in the chain (the returned optional value is nil).

Specifically, the result of an optional chaining call is of the same type as the expected return value, but wrapped in an optional. A property that normally returns an Int will return an Int? when accessed through optional chaining.

The next several code snippets demonstrate how optional chaining differs from forced unwrapping and enables you to check for success.

First, two classes called Person and Residence are defined:

1. class Person {
2. var residence: Residence?
3. }
4. class Residence {
5. var numberOfRooms = 1
6. }

Residence instances have a single Int property called numberOfRooms, with a default value of 1. Person instances have an optional residence property of type Residence?.

If you create a new Person instance, its residence property is default initialized to nil, by virtue of being optional. In the code below, john has a residence property value of nil:

1. let john = Person()

If you try to access the numberOfRooms property of this person’s residence, by placing an exclamation mark after residence to force the unwrapping of its value, you trigger a runtime error, because there is no residence value to unwrap:

1. let roomCount = john.residence!.numberOfRooms
2. // this triggers a runtime error

The code above succeeds when john.residence has a non-nil value and will set roomCount to an Int value containing the appropriate number of rooms. However, this code always triggers a runtime error when residence is nil, as illustrated above.

Optional chaining provides an alternative way to access the value of numberOfRooms. To use optional chaining, use a question mark in place of the exclamation mark:

1. if let roomCount = john.residence?.numberOfRooms {
2. print("John's residence has \(roomCount) room(s).")
3. } else {
4. print("Unable to retrieve the number of rooms.")
5. }
6. // Prints "Unable to retrieve the number of rooms."

This tells Swift to “chain” on the optional residence property and to retrieve the value of numberOfRooms if residence exists.

Because the attempt to access numberOfRooms has the potential to fail, the optional chaining attempt returns a value of type Int?, or “optional Int”. When residence is nil, as in the example above, this optional Int will also be nil, to reflect the fact that it was not possible to access numberOfRooms. The optional Int is accessed through optional binding to unwrap the integer and assign the non-optional value to the roomCount variable.

Note that this is true even though numberOfRooms is a non-optional Int. The fact that it is queried through an optional chain means that the call to numberOfRooms will always return an Int? instead of an Int.

You can assign a Residence instance to john.residence, so that it no longer has a nil value:

1. john.residence = Residence()

john.residence now contains an actual Residence instance, rather than nil. If you try to access numberOfRooms with the same optional chaining as before, it will now return an Int? that contains the default numberOfRooms value of 1:

if let roomCount = john.residence?.numberOfRooms {

print("John's residence has \(roomCount) room(s).")

} else {

print("Unable to retrieve the number of rooms.")

}

// Prints "John's residence has 1 room(s)."

## Defining Model Classes for Optional Chaining

You can use optional chaining with calls to properties, methods, and subscripts that are more than one level deep. This enables you to drill down into subproperties within complex models of interrelated types, and to check whether it is possible to access properties, methods, and subscripts on those subproperties.

The code snippets below define four model classes for use in several subsequent examples, including examples of multilevel optional chaining. These classes expand upon the Person and Residence model from above by adding a Room and Address class, with associated properties, methods, and subscripts.

The Person class is defined in the same way as before:

1. class Person {
2. var residence: Residence?
3. }

The Residence class is more complex than before. This time, the Residence class defines a variable property called rooms, which is initialized with an empty array of type [Room]:

1. class Residence {
2. var rooms = [Room]()
3. var numberOfRooms: Int {
4. return rooms.count
5. }
6. subscript(i: Int) -> Room {
7. get {
8. return rooms[i]
9. }
10. set {
11. rooms[i] = newValue
12. }
13. }
14. func printNumberOfRooms() {
15. print("The number of rooms is \(numberOfRooms)")
16. }
17. var address: Address?
18. }

Because this version of Residence stores an array of Room instances, its numberOfRooms property is implemented as a computed property, not a stored property. The computed numberOfRooms property simply returns the value of the count property from the rooms array.

As a shortcut to accessing its rooms array, this version of Residence provides a read-write subscript that provides access to the room at the requested index in the rooms array.

This version of Residence also provides a method called printNumberOfRooms, which simply prints the number of rooms in the residence.

Finally, Residence defines an optional property called address, with a type of Address?. The Address class type for this property is defined below.

The Room class used for the rooms array is a simple class with one property called name, and an initializer to set that property to a suitable room name:

1. class Room {
2. let name: String
3. init(name: String) { self.name = name }
4. }

The final class in this model is called Address. This class has three optional properties of type String?. The first two properties, buildingName and buildingNumber, are alternative ways to identify a particular building as part of an address. The third property, street, is used to name the street for that address:

class Address {

var buildingName: String?

var buildingNumber: String?

var street: String?

func buildingIdentifier() -> String? {

if let buildingNumber = buildingNumber, let street = street {

return "\(buildingNumber) \(street)"

} else if buildingName != nil {

return buildingName

} else {

return nil

}

}

}

The Address class also provides a method called buildingIdentifier(), which has a return type of String?. This method checks the properties of the address and returns buildingName if it has a value, or buildingNumber concatenated with street if both have values, or nil otherwise.

## Accessing Properties Through Optional Chaining

As demonstrated in [Optional Chaining as an Alternative to Forced Unwrapping](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html#ID246), you can use optional chaining to access a property on an optional value, and to check if that property access is successful.

Use the classes defined above to create a new Person instance, and try to access its numberOfRooms property as before:

1. let john = Person()
2. if let roomCount = john.residence?.numberOfRooms {
3. print("John's residence has \(roomCount) room(s).")
4. } else {
5. print("Unable to retrieve the number of rooms.")
6. }
7. // Prints "Unable to retrieve the number of rooms."

Because john.residence is nil, this optional chaining call fails in the same way as before.

You can also attempt to set a property’s value through optional chaining:

1. let someAddress = Address()
2. someAddress.buildingNumber = "29"
3. someAddress.street = "Acacia Road"
4. john.residence?.address = someAddress

In this example, the attempt to set the address property of john.residence will fail, because john.residence is currently nil.

The assignment is part of the optional chaining, which means none of the code on the right-hand side of the = operator is evaluated. In the previous example, it’s not easy to see that someAddress is never evaluated, because accessing a constant doesn’t have any side effects. The listing below does the same assignment, but it uses a function to create the address. The function prints “Function was called” before returning a value, which lets you see whether the right-hand side of the = operator was evaluated.

func createAddress() -> Address {

print("Function was called.")

let someAddress = Address()

someAddress.buildingNumber = "29"

someAddress.street = "Acacia Road"

return someAddress

}

john.residence?.address = createAddress()

You can tell that the createAddress() function isn’t called, because nothing is printed.

## Calling Methods Through Optional Chaining

You can use optional chaining to call a method on an optional value, and to check whether that method call is successful. You can do this even if that method does not define a return value.

The printNumberOfRooms() method on the Residence class prints the current value of numberOfRooms. Here’s how the method looks:

1. func printNumberOfRooms() {
2. print("The number of rooms is \(numberOfRooms)")
3. }

This method does not specify a return type. However, functions and methods with no return type have an implicit return type of Void, as described in [Functions Without Return Values](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID163). This means that they return a value of (), or an empty tuple.

If you call this method on an optional value with optional chaining, the method’s return type will be Void?, not Void, because return values are always of an optional type when called through optional chaining. This enables you to use an if statement to check whether it was possible to call the printNumberOfRooms() method, even though the method does not itself define a return value. Compare the return value from the printNumberOfRooms call against nil to see if the method call was successful:

1. if john.residence?.printNumberOfRooms() != nil {
2. print("It was possible to print the number of rooms.")
3. } else {
4. print("It was not possible to print the number of rooms.")
5. }
6. // Prints "It was not possible to print the number of rooms."

The same is true if you attempt to set a property through optional chaining. The example above in [Accessing Properties Through Optional Chaining](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html#ID248) attempts to set an address value for john.residence, even though the residence property is nil. Any attempt to set a property through optional chaining returns a value of type Void?, which enables you to compare against nil to see if the property was set successfully:

if (john.residence?.address = someAddress) != nil {

print("It was possible to set the address.")

} else {

print("It was not possible to set the address.")

}

// Prints "It was not possible to set the address."

## Accessing Subscripts Through Optional Chaining

You can use optional chaining to try to retrieve and set a value from a subscript on an optional value, and to check whether that subscript call is successful.

NOTE

When you access a subscript on an optional value through optional chaining, you place the question mark before the subscript’s brackets, not after. The optional chaining question mark always follows immediately after the part of the expression that is optional.

The example below tries to retrieve the name of the first room in the rooms array of the john.residence property using the subscript defined on the Residence class. Because john.residence is currently nil, the subscript call fails:

1. if let firstRoomName = john.residence?[0].name {
2. print("The first room name is \(firstRoomName).")
3. } else {
4. print("Unable to retrieve the first room name.")
5. }
6. // Prints "Unable to retrieve the first room name."

The optional chaining question mark in this subscript call is placed immediately after john.residence, before the subscript brackets, because john.residence is the optional value on which optional chaining is being attempted.

Similarly, you can try to set a new value through a subscript with optional chaining:

1. john.residence?[0] = Room(name: "Bathroom")

This subscript setting attempt also fails, because residence is currently nil.

If you create and assign an actual Residence instance to john.residence, with one or more Room instances in its rooms array, you can use the Residence subscript to access the actual items in the rooms array through optional chaining:

1. let johnsHouse = Residence()
2. johnsHouse.rooms.append(Room(name: "Living Room"))
3. johnsHouse.rooms.append(Room(name: "Kitchen"))
4. john.residence = johnsHouse
5. if let firstRoomName = john.residence?[0].name {
6. print("The first room name is \(firstRoomName).")
7. } else {
8. print("Unable to retrieve the first room name.")
9. }
10. // Prints "The first room name is Living Room."

### Accessing Subscripts of Optional Type

If a subscript returns a value of optional type—such as the key subscript of Swift’s Dictionary type—place a question mark after the subscript’s closing bracket to chain on its optional return value:

1. var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]
2. testScores["Dave"]?[0] = 91
3. testScores["Bev"]?[0] += 1
4. testScores["Brian"]?[0] = 72
5. // the "Dave" array is now [91, 82, 84] and the "Bev" array is now [80, 94, 81]

The example above defines a dictionary called testScores, which contains two key-value pairs that map a String key to an array of Int values. The example uses optional chaining to set the first item in the "Dave" array to 91; to increment the first item in the "Bev" array by 1; and to try to set the first item in an array for a key of "Brian". The first two calls succeed, because the testScores dictionary contains keys for "Dave" and "Bev". The third call fails, because the testScores dictionary does not contain a key for "Brian".

## Linking Multiple Levels of Chaining

You can link together multiple levels of optional chaining to drill down to properties, methods, and subscripts deeper within a model. However, multiple levels of optional chaining do not add more levels of optionality to the returned value.

To put it another way:

* If the type you are trying to retrieve is not optional, it will become optional because of the optional chaining.
* If the type you are trying to retrieve is already optional, it will not become more optional because of the chaining.

Therefore:

* If you try to retrieve an Int value through optional chaining, an Int? is always returned, no matter how many levels of chaining are used.
* Similarly, if you try to retrieve an Int? value through optional chaining, an Int? is always returned, no matter how many levels of chaining are used.

The example below tries to access the street property of the address property of the residence property of john. There are two levels of optional chaining in use here, to chain through the residence and address properties, both of which are of optional type:

1. if let johnsStreet = john.residence?.address?.street {
2. print("John's street name is \(johnsStreet).")
3. } else {
4. print("Unable to retrieve the address.")
5. }
6. // Prints "Unable to retrieve the address."

The value of john.residence currently contains a valid Residence instance. However, the value of john.residence.address is currently nil. Because of this, the call to john.residence?.address?.street fails.

Note that in the example above, you are trying to retrieve the value of the street property. The type of this property is String?. The return value of john.residence?.address?.street is therefore also String?, even though two levels of optional chaining are applied in addition to the underlying optional type of the property.

If you set an actual Address instance as the value for john.residence.address, and set an actual value for the address’s street property, you can access the value of the street property through multilevel optional chaining:

let johnsAddress = Address()

johnsAddress.buildingName = "The Larches"

johnsAddress.street = "Laurel Street"

john.residence?.address = johnsAddress

if let johnsStreet = john.residence?.address?.street {

print("John's street name is \(johnsStreet).")

} else {

print("Unable to retrieve the address.")

}

// Prints "John's street name is Laurel Street."

In this example, the attempt to set the address property of john.residence will succeed, because the value of john.residence currently contains a valid Residence instance.

## Chaining on Methods with Optional Return Values

The previous example shows how to retrieve the value of a property of optional type through optional chaining. You can also use optional chaining to call a method that returns a value of optional type, and to chain on that method’s return value if needed.

The example below calls the Address class’s buildingIdentifier() method through optional chaining. This method returns a value of type String?. As described above, the ultimate return type of this method call after optional chaining is also String?:

1. if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
2. print("John's building identifier is \(buildingIdentifier).")
3. }
4. // Prints "John's building identifier is The Larches."

If you want to perform further optional chaining on this method’s return value, place the optional chaining question mark after the method’s parentheses:

1. if let beginsWithThe =
2. john.residence?.address?.buildingIdentifier()?.hasPrefix("The") {
3. if beginsWithThe {
4. print("John's building identifier begins with \"The\".")
5. } else {
6. print("John's building identifier does not begin with \"The\".")
7. }
8. }
9. // Prints "John's building identifier begins with "The"."

NOTE

In the example above, you place the optional chaining question mark after the parentheses, because the optional value you are chaining on is the buildingIdentifier() method’s return value, and not the buildingIdentifier() method itself.

There are four ways to handle errors in Swift. You can propagate the error from a function to the code that calls that function, handle the error using a do-catch statement, handle the error as an optional value, or assert that the error will not occur. Each approach is described in a section below.

NOTE

Error handling in Swift resembles exception handling in other languages, with the use of the try, catch and throw keywords. Unlike exception handling in many languages—including Objective-C—error handling in Swift does not involve unwinding the call stack, a process that can be computationally expensive. As such, the performance characteristics of a throw statement are comparable to those of a return statement.

### Adding the SDK Libraries to the Xcode Project

SAP Cloud Platform SDK for iOS contains functionality in libraries that can be added to Xcode projects. There are two types of libraries, ones that are specific to a simulator or device and ones that can be used in both instances (fat). The fat libraries are useful during development, but when you submit the app to the Apple App Store, you must not use fat libraries.

An overview of the SAP Predictive service APIs provided as web services in the cloud.

***Warning:****The HANA database in your HCP trial account is stopped automatically every 12 hours. So you need to ensure that your HANA database is running before executing each unit, especially if you don’t work on this course continuously.****If the SAP HANA multitenant database remains in a stopped state for two weeks, it will be deleted from your HCP trial account!****In the SAP HANA Cloud Platform Cockpit you can see when it will be stopped or deleted.*

# [“Install Spotify” can't be opened because Apple cannot check it for malicious software](https://apple.stackexchange.com/questions/366542/install-spotify-cant-be-opened-because-apple-cannot-check-it-for-malicious-so)

## for macOS Catalina

I think codesign sometimes doesn't work for the **notarization** issue recently, so you should use xattr to remove the quarantine:

$ xattr -d com.apple.quarantine <app-path>

I have got a conclusion. When installing app from 3rd party on different macOS, you can try different ways when it comes up to Move to Trash issue. Meanwhile, I think this is a common issue for 3rd party apps, so you may change the title question to something that contains Cannot be opened or Move to trash as well.

# Error Handling

There are four ways to handle errors in Swift. You can propagate the error from a function to the code that calls that function, handle the error using a do-catch statement, handle the error as an optional value, or assert that the error will not occur. Each approach is described in a section below.

### Propagating Errors Using Throwing Functions

To indicate that a function, method, or initializer can throw an error, you write the throws keyword in the function’s declaration after its parameters. A function marked with throws is called a throwing function. If the function specifies a return type, you write the throws keyword before the return arrow (->).

func canThrowErrors() throws -> String

func cannotThrowErrors() -> String

A throwing function propagates errors that are thrown inside of it to the scope from which it’s called.

NOTE

Only throwing functions can propagate errors. Any errors thrown inside a nonthrowing function must be handled inside the function.

In the example below, the VendingMachine class has a vend(itemNamed:) method that throws an appropriate VendingMachineError if the requested item is not available, is out of stock, or has a cost that exceeds the current deposited amount:

1. struct Item {
2. var price: Int
3. var count: Int
4. }
5. class VendingMachine {
6. var inventory = [
7. "Candy Bar": Item(price: 12, count: 7),
8. "Chips": Item(price: 10, count: 4),
9. "Pretzels": Item(price: 7, count: 11)
10. ]
11. var coinsDeposited = 0
12. func vend(itemNamed name: String) throws {
13. guard let item = inventory[name] else {
14. throw VendingMachineError.invalidSelection
15. }
16. guard item.count > 0 else {
17. throw VendingMachineError.outOfStock
18. }
19. guard item.price <= coinsDeposited else {
20. throw VendingMachineError.insufficientFunds(coinsNeeded: item.price - coinsDeposited)
21. }
22. coinsDeposited -= item.price
23. var newItem = item
24. newItem.count -= 1
25. inventory[name] = newItem
26. print("Dispensing \(name)")
27. }
28. }

The implementation of the vend(itemNamed:) method uses guard statements to exit the method early and throw appropriate errors if any of the requirements for purchasing a snack aren’t met. Because a throw statement immediately transfers program control, an item will be vended only if all of these requirements are met.

Because the vend(itemNamed:) method propagates any errors it throws, any code that calls this method must either handle the errors—using a do-catch statement, try?, or try!—or continue to propagate them. For example, the buyFavoriteSnack(person:vendingMachine:) in the example below is also a throwing function, and any errors that the vend(itemNamed:) method throws will propagate up to the point where the buyFavoriteSnack(person:vendingMachine:) function is called.

1. let favoriteSnacks = [
2. "Alice": "Chips",
3. "Bob": "Licorice",
4. "Eve": "Pretzels",
5. ]
6. func buyFavoriteSnack(person: String, vendingMachine: VendingMachine) throws {
7. let snackName = favoriteSnacks[person] ?? "Candy Bar"
8. try vendingMachine.vend(itemNamed: snackName)
9. }

In this example, the buyFavoriteSnack(person: vendingMachine:) function looks up a given person’s favorite snack and tries to buy it for them by calling the vend(itemNamed:) method. Because the vend(itemNamed:) method can throw an error, it’s called with the try keyword in front of it.

Throwing initializers can propagate errors in the same way as throwing functions. For example, the initializer for the PurchasedSnack structure in the listing below calls a throwing function as part of the initialization process, and it handles any errors that it encounters by propagating them to its caller.

1. struct PurchasedSnack {
2. let name: String
3. init(name: String, vendingMachine: VendingMachine) throws {
4. try vendingMachine.vend(itemNamed: name)
5. self.name = name
6. }
7. }

### Handling Errors Using Do-Catch

You use a do-catch statement to handle errors by running a block of code. If an error is thrown by the code in the do clause, it is matched against the catch clauses to determine which one of them can handle the error.

Here is the general form of a do-catch statement:

do {

try expression

statements

} catch pattern 1 {

statements

} catch pattern 2 where condition {

statements

} catch {

statements

}

For example, the following code matches against all three cases of the VendingMachineError enumeration.

1. var vendingMachine = VendingMachine()
2. vendingMachine.coinsDeposited = 8
3. do {
4. try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
5. print("Success! Yum.")
6. } catch VendingMachineError.invalidSelection {
7. print("Invalid Selection.")
8. } catch VendingMachineError.outOfStock {
9. print("Out of Stock.")
10. } catch VendingMachineError.insufficientFunds(let coinsNeeded) {
11. print("Insufficient funds. Please insert an additional \(coinsNeeded) coins.")
12. } catch {
13. print("Unexpected error: \(error).")
14. }
15. // Prints "Insufficient funds. Please insert an additional 2 coins."

In the above example, the buyFavoriteSnack(person:vendingMachine:) function is called in a try expression, because it can throw an error. If an error is thrown, execution immediately transfers to the catch clauses, which decide whether to allow propagation to continue. If no pattern is matched, the error gets caught by the final catch clause and is bound to a local error constant. If no error is thrown, the remaining statements in the do statement are executed.

The catch clauses don’t have to handle every possible error that the code in the do clause can throw. If none of the catch clauses handle the error, the error propagates to the surrounding scope. However, the propagated error must be handled by *some* surrounding scope. In a nonthrowing function, an enclosing do-catch clause must handle the error. In a throwing function, either an enclosing do-catch clause or the caller must handle the error. If the error propagates to the top-level scope without being handled, you’ll get a runtime error.

For example, the above example can be written so any error that isn’t a VendingMachineError is instead caught by the calling function:

func nourish(with item: String) throws {

do {

try vendingMachine.vend(itemNamed: item)

} catch is VendingMachineError {

print("Invalid selection, out of stock, or not enough money.")

}

}

do {

try nourish(with: "Beet-Flavored Chips")

} catch {

print("Unexpected non-vending-machine-related error: \(error)")

}

1. // Prints "Invalid selection, out of stock, or not enough money."

In the nourish(with:) function, if vend(itemNamed:) throws an error that’s one of the cases of the VendingMachineError enumeration, nourish(with:) handles the error by printing a message. Otherwise, nourish(with:) propagates the error to its call site. The error is then caught by the general catch clause.

### Converting Errors to Optional Values

You use try? to handle an error by converting it to an optional value. If an error is thrown while evaluating the try? expression, the value of the expression is nil. For example, in the following code x and y have the same value and behavior:

func someThrowingFunction() throws -> Int {

// ...

}

let x = try? someThrowingFunction()

let y: Int?

do {

y = try someThrowingFunction()

} catch {

y = nil

}

If someThrowingFunction() throws an error, the value of x and y is nil. Otherwise, the value of x and y is the value that the function returned. Note that x and y are an optional of whatever type someThrowingFunction() returns. Here the function returns an integer, so x and y are optional integers.

Using try? lets you write concise error handling code when you want to handle all errors in the same way. For example, the following code uses several approaches to fetch data, or returns nil if all of the approaches fail.

func fetchData() -> Data? {

if let data = try? fetchDataFromDisk() { return data }

if let data = try? fetchDataFromServer() { return data }

return nil

}

### Disabling Error Propagation

Sometimes you know a throwing function or method won’t, in fact, throw an error at runtime. On those occasions, you can write try! before the expression to disable error propagation and wrap the call in a runtime assertion that no error will be thrown. If an error actually is thrown, you’ll get a runtime error.

For example, the following code uses a loadImage(atPath:) function, which loads the image resource at a given path or throws an error if the image can’t be loaded. In this case, because the image is shipped with the application, no error will be thrown at runtime, so it is appropriate to disable error propagation.

1. let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")

## Specifying Cleanup Actions

You use a defer statement to execute a set of statements just before code execution leaves the current block of code. This statement lets you do any necessary cleanup that should be performed regardless of how execution leaves the current block of code—whether it leaves because an error was thrown or because of a statement such as return or break. For example, you can use a defer statement to ensure that file descriptors are closed and manually allocated memory is freed.

A defer statement defers execution until the current scope is exited. This statement consists of the defer keyword and the statements to be executed later. The deferred statements may not contain any code that would transfer control out of the statements, such as a break or a return statement, or by throwing an error. Deferred actions are executed in the reverse of the order that they’re written in your source code. That is, the code in the first defer statement executes last, the code in the second defer statement executes second to last, and so on. The last defer statement in source code order executes first.

1. func processFile(filename: String) throws {
2. if exists(filename) {
3. let file = open(filename)
4. defer {
5. close(file)
6. }
7. while let line = try file.readline() {
8. // Work with the file.
9. }
10. // close(file) is called here, at the end of the scope.
11. }
12. }

The above example uses a defer statement to ensure that the open(\_:) function has a corresponding call to close(\_:).

NOTE

You can use a defer statement even when no error handling code is involved.

# Type Casting

*Type casting* is a way to check the type of an instance, or to treat that instance as a different superclass or subclass from somewhere else in its own class hierarchy.

Type casting in Swift is implemented with the is and as operators. These two operators provide a simple and expressive way to check the type of a value or cast a value to a different type.

## Defining a Class Hierarchy for Type Casting

You can use type casting with a hierarchy of classes and subclasses to check the type of a particular class instance and to cast that instance to another class within the same hierarchy. The three code snippets below define a hierarchy of classes and an array containing instances of those classes, for use in an example of type casting.

The first snippet defines a new base class called MediaItem. This class provides basic functionality for any kind of item that appears in a digital media library. Specifically, it declares a name property of type String, and an init name initializer. (It is assumed that all media items, including all movies and songs, will have a name.)

1. class MediaItem {
2. var name: String
3. init(name: String) {
4. self.name = name
5. }
6. }

The next snippet defines two subclasses of MediaItem. The first subclass, Movie, encapsulates additional information about a movie or film. It adds a director property on top of the base MediaItem class, with a corresponding initializer. The second subclass, Song, adds an artist property and initializer on top of the base class:

1. class Movie: MediaItem {
2. var director: String
3. init(name: String, director: String) {
4. self.director = director
5. super.init(name: name)
6. }
7. }
8. class Song: MediaItem {
9. var artist: String
10. init(name: String, artist: String) {
11. self.artist = artist
12. super.init(name: name)
13. }
14. }

The final snippet creates a constant array called library, which contains two Movie instances and three Song instances. The type of the library array is inferred by initializing it with the contents of an array literal. Swift’s type checker is able to deduce that Movie and Song have a common superclass of MediaItem, and so it infers a type of [MediaItem] for the library array:

1. let library = [
2. Movie(name: "Casablanca", director: "Michael Curtiz"),
3. Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
4. Movie(name: "Citizen Kane", director: "Orson Welles"),
5. Song(name: "The One And Only", artist: "Chesney Hawkes"),
6. Song(name: "Never Gonna Give You Up", artist: "Rick Astley")
7. ]
8. // the type of "library" is inferred to be [MediaItem]

The items stored in library are still Movie and Song instances behind the scenes. However, if you iterate over the contents of this array, the items you receive back are typed as MediaItem, and not as Movie or Song. In order to work with them as their native type, you need to check their type, or downcast them to a different type, as described below.

## Checking Type

Use the type check operator (is) to check whether an instance is of a certain subclass type. The type check operator returns true if the instance is of that subclass type and false if it is not.

The example below defines two variables, movieCount and songCount, which count the number of Movie and Song instances in the library array:

1. var movieCount = 0
2. var songCount = 0
3. for item in library {
4. if item is Movie {
5. movieCount += 1
6. } else if item is Song {
7. songCount += 1
8. }
9. }
10. print("Media library contains \(movieCount) movies and \(songCount) songs")
11. // Prints "Media library contains 2 movies and 3 songs"

This example iterates through all items in the library array. On each pass, the for-in loop sets the item constant to the next MediaItem in the array.

item is Movie returns true if the current MediaItem is a Movie instance and false if it is not. Similarly, item is Song checks whether the item is a Song instance. At the end of the for-in loop, the values of movieCount and songCount contain a count of how many MediaItem instances were found of each type.

## Downcasting

A constant or variable of a certain class type may actually refer to an instance of a subclass behind the scenes. Where you believe this is the case, you can try to downcast to the subclass type with a type cast operator (as? or as!).

Because downcasting can fail, the type cast operator comes in two different forms. The conditional form, as?, returns an optional value of the type you are trying to downcast to. The forced form, as!, attempts the downcast and force-unwraps the result as a single compound action.

Use the conditional form of the type cast operator (as?) when you are not sure if the downcast will succeed. This form of the operator will always return an optional value, and the value will be nil if the downcast was not possible. This enables you to check for a successful downcast.

Use the forced form of the type cast operator (as!) only when you are sure that the downcast will always succeed. This form of the operator will trigger a runtime error if you try to downcast to an incorrect class type.

The example below iterates over each MediaItem in library, and prints an appropriate description for each item. To do this, it needs to access each item as a true Movie or Song, and not just as a MediaItem. This is necessary in order for it to be able to access the director or artist property of a Movie or Song for use in the description.

In this example, each item in the array might be a Movie, or it might be a Song. You don’t know in advance which actual class to use for each item, and so it is appropriate to use the conditional form of the type cast operator (as?) to check the downcast each time through the loop:

1. for item in library {
2. if let movie = item as? Movie {
3. print("Movie: \(movie.name), dir. \(movie.director)")
4. } else if let song = item as? Song {
5. print("Song: \(song.name), by \(song.artist)")
6. }
7. }
8. // Movie: Casablanca, dir. Michael Curtiz
9. // Song: Blue Suede Shoes, by Elvis Presley
10. // Movie: Citizen Kane, dir. Orson Welles
11. // Song: The One And Only, by Chesney Hawkes
12. // Song: Never Gonna Give You Up, by Rick Astley

The example starts by trying to downcast the current item as a Movie. Because item is a MediaItem instance, it’s possible that it might be a Movie; equally, it’s also possible that it might be a Song, or even just a base MediaItem. Because of this uncertainty, the as? form of the type cast operator returns an optional value when attempting to downcast to a subclass type. The result of item as? Movie is of type Movie?, or “optional Movie”.

Downcasting to Movie fails when applied to the Song instances in the library array. To cope with this, the example above uses optional binding to check whether the optional Movie actually contains a value (that is, to find out whether the downcast succeeded.) This optional binding is written “if let movie = item as? Movie”, which can be read as:

“Try to access item as a Movie. If this is successful, set a new temporary constant called movie to the value stored in the returned optional Movie.”

If the downcasting succeeds, the properties of movie are then used to print a description for that Movie instance, including the name of its director. A similar principle is used to check for Song instances, and to print an appropriate description (including artist name) whenever a Song is found in the library.

NOTE

Casting does not actually modify the instance or change its values. The underlying instance remains the same; it is simply treated and accessed as an instance of the type to which it has been cast.

## Type Casting for Any and AnyObject

Swift provides two special types for working with nonspecific types:

* Any can represent an instance of any type at all, including function types.
* AnyObject can represent an instance of any class type.

Use Any and AnyObject only when you explicitly need the behavior and capabilities they provide. It is always better to be specific about the types you expect to work with in your code.

Here’s an example of using Any to work with a mix of different types, including function types and nonclass types. The example creates an array called things, which can store values of type Any:

1. var things = [Any]()
2. things.append(0)
3. things.append(0.0)
4. things.append(42)
5. things.append(3.14159)
6. things.append("hello")
7. things.append((3.0, 5.0))
8. things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))
9. things.append({ (name: String) -> String in "Hello, \(name)" })

The things array contains two Int values, two Double values, a String value, a tuple of type (Double, Double), the movie “Ghostbusters”, and a closure expression that takes a String value and returns another String value.

To discover the specific type of a constant or variable that is known only to be of type Any or AnyObject, you can use an is or as pattern in a switch statement’s cases. The example below iterates over the items in the things array and queries the type of each item with a switch statement. Several of the switch statement’s cases bind their matched value to a constant of the specified type to enable its value to be printed:

1. for thing in things {
2. switch thing {
3. case 0 as Int:
4. print("zero as an Int")
5. case 0 as Double:
6. print("zero as a Double")
7. case let someInt as Int:
8. print("an integer value of \(someInt)")
9. case let someDouble as Double where someDouble > 0:
10. print("a positive double value of \(someDouble)")
11. case is Double:
12. print("some other double value that I don't want to print")
13. case let someString as String:
14. print("a string value of \"\(someString)\"")
15. case let (x, y) as (Double, Double):
16. print("an (x, y) point at \(x), \(y)")
17. case let movie as Movie:
18. print("a movie called \(movie.name), dir. \(movie.director)")
19. case let stringConverter as (String) -> String:
20. print(stringConverter("Michael"))
21. default:
22. print("something else")
23. }
24. }
25. // zero as an Int
26. // zero as a Double
27. // an integer value of 42
28. // a positive double value of 3.14159
29. // a string value of "hello"
30. // an (x, y) point at 3.0, 5.0
31. // a movie called Ghostbusters, dir. Ivan Reitman
32. // Hello, Michael

NOTE

The Any type represents values of any type, including optional types. Swift gives you a warning if you use an optional value where a value of type Any is expected. If you really do need to use an optional value as an Any value, you can use the as operator to explicitly cast the optional to Any, as shown below.

1. let optionalNumber: Int? = 3
2. things.append(optionalNumber) // Warning
3. things.append(optionalNumber as Any) // No warning

# Nested Types

Enumerations are often created to support a specific class or structure’s functionality. Similarly, it can be convenient to define utility classes and structures purely for use within the context of a more complex type. To accomplish this, Swift enables you to define nested types, whereby you nest supporting enumerations, classes, and structures within the definition of the type they support.

To nest a type within another type, write its definition within the outer braces of the type it supports. Types can be nested to as many levels as are required.

## Nested Types in Action

The example below defines a structure called BlackjackCard, which models a playing card as used in the game of Blackjack. The BlackjackCard structure contains two nested enumeration types called Suit and Rank.

In Blackjack, the Ace cards have a value of either one or eleven. This feature is represented by a structure called Values, which is nested within the Rank enumeration:

1. struct BlackjackCard {
2. // nested Suit enumeration
3. enum Suit: Character {
4. case spades = "♠", hearts = "♡", diamonds = "♢", clubs = "♣"
5. }
6. // nested Rank enumeration
7. enum Rank: Int {
8. case two = 2, three, four, five, six, seven, eight, nine, ten
9. case jack, queen, king, ace
10. struct Values {
11. let first: Int, second: Int?
12. }
13. var values: Values {
14. switch self {
15. case .ace:
16. return Values(first: 1, second: 11)
17. case .jack, .queen, .king:
18. return Values(first: 10, second: nil)
19. default:
20. return Values(first: self.rawValue, second: nil)
21. }
22. }
23. }
24. // BlackjackCard properties and methods
25. let rank: Rank, suit: Suit
26. var description: String {
27. var output = "suit is \(suit.rawValue),"
28. output += " value is \(rank.values.first)"
29. if let second = rank.values.second {
30. output += " or \(second)"
31. }
32. return output
33. }
34. }

The Suit enumeration describes the four common playing card suits, together with a raw Character value to represent their symbol.

The Rank enumeration describes the thirteen possible playing card ranks, together with a raw Int value to represent their face value. (This raw Int value is not used for the Jack, Queen, King, and Ace cards.)

As mentioned above, the Rank enumeration defines a further nested structure of its own, called Values. This structure encapsulates the fact that most cards have one value, but the Ace card has two values. The Values structure defines two properties to represent this:

* first, of type Int
* second, of type Int?, or “optional Int”

Rank also defines a computed property, values, which returns an instance of the Values structure. This computed property considers the rank of the card and initializes a new Values instance with appropriate values based on its rank. It uses special values for jack, queen, king, and ace. For the numeric cards, it uses the rank’s raw Int value.

The BlackjackCard structure itself has two properties—rank and suit. It also defines a computed property called description, which uses the values stored in rank and suit to build a description of the name and value of the card. The description property uses optional binding to check whether there is a second value to display, and if so, inserts additional description detail for that second value.

Because BlackjackCard is a structure with no custom initializers, it has an implicit memberwise initializer, as described in [Memberwise Initializers for Structure Types](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID214). You can use this initializer to initialize a new constant called theAceOfSpades:

1. let theAceOfSpades = BlackjackCard(rank: .ace, suit: .spades)
2. print("theAceOfSpades: \(theAceOfSpades.description)")
3. // Prints "theAceOfSpades: suit is ♠, value is 1 or 11"

Even though Rank and Suit are nested within BlackjackCard, their type can be inferred from context, and so the initialization of this instance is able to refer to the enumeration cases by their case names (.ace and .spades) alone. In the example above, the description property correctly reports that the Ace of Spades has a value of 1 or 11.

## Referring to Nested Types

To use a nested type outside of its definition context, prefix its name with the name of the type it is nested within:

1. let heartsSymbol = BlackjackCard.Suit.hearts.rawValue
2. // heartsSymbol is "♡"

For the example above, this enables the names of Suit, Rank, and Values to be kept deliberately short, because their names are naturally qualified by the context in which they are defined.

# Extensions

*Extensions* add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you do not have access to the original source code (known as *retroactive modeling*). Extensions are similar to categories in Objective-C. (Unlike Objective-C categories, Swift extensions do not have names.)

Extensions in Swift can:

* Add computed instance properties and computed type properties
* Define instance methods and type methods
* Provide new initializers
* Define subscripts
* Define and use new nested types
* Make an existing type conform to a protocol

In Swift, you can even extend a protocol to provide implementations of its requirements or add additional functionality that conforming types can take advantage of.

Extensions can add new functionality to a type, but they cannot override existing functionality.

## Extension Syntax

Declare extensions with the extension keyword:

1. extension SomeType {
2. // new functionality to add to SomeType goes here
3. }

An extension can extend an existing type to make it adopt one or more protocols. To add protocol conformance, you write the protocol names the same way as you write them for a class or structure:

1. extension SomeType: SomeProtocol, AnotherProtocol {
2. // implementation of protocol requirements goes here
3. }

An extension can be used to extend an existing generic type, as described in [Extending a Generic Type](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID185). You can also extend a generic type to conditionally add functionality,

If you define an extension to add new functionality to an existing type, the new functionality will be available on all existing instances of that type, even if they were created before the extension was defined.

## Computed Properties

Extensions can add computed instance properties and computed type properties to existing types. This example adds five computed instance properties to Swift’s built-in Double type, to provide basic support for working with distance units:

1. extension Double {
2. var km: Double { return self \* 1\_000.0 }
3. var m: Double { return self }
4. var cm: Double { return self / 100.0 }
5. var mm: Double { return self / 1\_000.0 }
6. var ft: Double { return self / 3.28084 }
7. }
8. let oneInch = 25.4.mm
9. print("One inch is \(oneInch) meters")
10. // Prints "One inch is 0.0254 meters"
11. let threeFeet = 3.ft
12. print("Three feet is \(threeFeet) meters")
13. // Prints "Three feet is 0.914399970739201 meters"

These computed properties express that a Double value should be considered as a certain unit of length. Although they are implemented as computed properties, the names of these properties can be appended to a floating-point literal value with dot syntax, as a way to use that literal value to perform distance conversions.

In this example, a Double value of 1.0 is considered to represent “one meter”. This is why the m computed property returns self—the expression 1.m is considered to calculate a Double value of 1.0.

Other units require some conversion to be expressed as a value measured in meters. One kilometer is the same as 1,000 meters, so the km computed property multiplies the value by 1\_000.00 to convert into a number expressed in meters. Similarly, there are 3.28084 feet in a meter, and so the ft computed property divides the underlying Double value by 3.28084, to convert it from feet to meters.

These properties are read-only computed properties, and so they are expressed without the get keyword, for brevity. Their return value is of type Double, and can be used within mathematical calculations wherever a Double is accepted:

1. let aMarathon = 42.km + 195.m
2. print("A marathon is \(aMarathon) meters long")
3. // Prints "A marathon is 42195.0 meters long"

NOTE

Extensions can add new computed properties, but they cannot add stored properties, or add property observers to existing properties.

## Initializers

Extensions can add new initializers to existing types. This enables you to extend other types to accept your own custom types as initializer parameters, or to provide additional initialization options that were not included as part of the type’s original implementation.

Extensions can add new convenience initializers to a class, but they cannot add new designated initializers or deinitializers to a class. Designated initializers and deinitializers must always be provided by the original class implementation.

If you use an extension to add an initializer to a value type that provides default values for all of its stored properties and does not define any custom initializers, you can call the default initializer and memberwise initializer for that value type from within your extension’s initializer. This wouldn’t be the case if you had written the initializer as part of the value type’s original implementation, as described in [Initializer Delegation for Value Types](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID215).

If you use an extension to add an initializer to a structure that was declared in another module, the new initializer can’t access self until it calls an initializer from the defining module.

The example below defines a custom Rect structure to represent a geometric rectangle. The example also defines two supporting structures called Size and Point, both of which provide default values of 0.0 for all of their properties:

1. struct Size {
2. var width = 0.0, height = 0.0
3. }
4. struct Point {
5. var x = 0.0, y = 0.0
6. }
7. struct Rect {
8. var origin = Point()
9. var size = Size()
10. }

Because the Rect structure provides default values for all of its properties, it receives a default initializer and a memberwise initializer automatically, as described in [Default Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID213). These initializers can be used to create new Rect instances:

1. let defaultRect = Rect()
2. let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
3. size: Size(width: 5.0, height: 5.0))

You can extend the Rect structure to provide an additional initializer that takes a specific center point and size:

1. extension Rect {
2. init(center: Point, size: Size) {
3. let originX = center.x - (size.width / 2)
4. let originY = center.y - (size.height / 2)
5. self.init(origin: Point(x: originX, y: originY), size: size)
6. }
7. }

This new initializer starts by calculating an appropriate origin point based on the provided center point and size value. The initializer then calls the structure’s automatic memberwise initializer init(origin:size:), which stores the new origin and size values in the appropriate properties:

1. let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
2. size: Size(width: 3.0, height: 3.0))
3. // centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)

NOTE

If you provide a new initializer with an extension, you are still responsible for making sure that each instance is fully initialized once the initializer completes.

## Methods

Extensions can add new instance methods and type methods to existing types. The following example adds a new instance method called repetitions to the Int type:

1. extension Int {
2. func repetitions(task: () -> Void) {
3. for \_ in 0..<self {
4. task()
5. }
6. }
7. }

The repetitions(task:) method takes a single argument of type () -> Void, which indicates a function that has no parameters and does not return a value.

After defining this extension, you can call the repetitions(task:) method on any integer to perform a task that many number of times:

1. 3.repetitions {
2. print("Hello!")
3. }
4. // Hello!
5. // Hello!
6. // Hello!

### Mutating Instance Methods

Instance methods added with an extension can also modify (or mutate) the instance itself. Structure and enumeration methods that modify self or its properties must mark the instance method as mutating, just like mutating methods from an original implementation.

The example below adds a new mutating method called square to Swift’s Int type, which squares the original value:

1. extension Int {
2. mutating func square() {
3. self = self \* self
4. }
5. }
6. var someInt = 3
7. someInt.square()
8. // someInt is now 9

## Subscripts

Extensions can add new subscripts to an existing type. This example adds an integer subscript to Swift’s built-in Int type. This subscript [n] returns the decimal digit n places in from the right of the number:

* 123456789[0] returns 9
* 123456789[1] returns 8

…and so on:

1. extension Int {
2. subscript(digitIndex: Int) -> Int {
3. var decimalBase = 1
4. for \_ in 0..<digitIndex {
5. decimalBase \*= 10
6. }
7. return (self / decimalBase) % 10
8. }
9. }
10. 746381295[0]
11. // returns 5
12. 746381295[1]
13. // returns 9
14. 746381295[2]
15. // returns 2
16. 746381295[8]
17. // returns 7

If the Int value does not have enough digits for the requested index, the subscript implementation returns 0, as if the number had been padded with zeros to the left:

1. 746381295[9]
2. // returns 0, as if you had requested:
3. 0746381295[9]

## Nested Types

Extensions can add new nested types to existing classes, structures, and enumerations:

1. extension Int {
2. enum Kind {
3. case negative, zero, positive
4. }
5. var kind: Kind {
6. switch self {
7. case 0:
8. return .zero
9. case let x where x > 0:
10. return .positive
11. default:
12. return .negative
13. }
14. }
15. }

This example adds a new nested enumeration to Int. This enumeration, called Kind, expresses the kind of number that a particular integer represents. Specifically, it expresses whether the number is negative, zero, or positive.

This example also adds a new computed instance property to Int, called kind, which returns the appropriate Kind enumeration case for that integer.

The nested enumeration can now be used with any Int value:

1. func printIntegerKinds(\_ numbers: [Int]) {
2. for number in numbers {
3. switch number.kind {
4. case .negative:
5. print("- ", terminator: "")
6. case .zero:
7. print("0 ", terminator: "")
8. case .positive:
9. print("+ ", terminator: "")
10. }
11. }
12. print("")
13. }
14. printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
15. // Prints "+ + - 0 - 0 + "

This function, printIntegerKinds(\_:), takes an input array of Int values and iterates over those values in turn. For each integer in the array, the function considers the kind computed property for that integer, and prints an appropriate description.

NOTE

number.kind is already known to be of type Int.Kind. Because of this, all of the Int.Kind case values can be written in shorthand form inside the switch statement, such as .negative rather than Int.Kind.negative.

# Protocols

A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to conform to that protocol.

In addition to specifying requirements that conforming types must implement, you can extend a protocol to implement some of these requirements or to implement additional functionality that conforming types can take advantage of.

## Protocol Syntax

You define protocols in a very similar way to classes, structures, and enumerations:

1. protocol SomeProtocol {
2. // protocol definition goes here
3. }

Custom types state that they adopt a particular protocol by placing the protocol’s name after the type’s name, separated by a colon, as part of their definition. Multiple protocols can be listed, and are separated by commas:

1. struct SomeStructure: FirstProtocol, AnotherProtocol {
2. // structure definition goes here
3. }

If a class has a superclass, list the superclass name before any protocols it adopts, followed by a comma:

1. class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
2. // class definition goes here
3. }

## Property Requirements

A protocol can require any conforming type to provide an instance property or type property with a particular name and type. The protocol doesn’t specify whether the property should be a stored property or a computed property—it only specifies the required property name and type. The protocol also specifies whether each property must be gettable or gettable and settable.

If a protocol requires a property to be gettable and settable, that property requirement can’t be fulfilled by a constant stored property or a read-only computed property. If the protocol only requires a property to be gettable, the requirement can be satisfied by any kind of property, and it’s valid for the property to be also settable if this is useful for your own code.

Property requirements are always declared as variable properties, prefixed with the var keyword. Gettable and settable properties are indicated by writing { get set } after their type declaration, and gettable properties are indicated by writing { get }.

1. protocol SomeProtocol {
2. var mustBeSettable: Int { get set }
3. var doesNotNeedToBeSettable: Int { get }
4. }

Always prefix type property requirements with the static keyword when you define them in a protocol. This rule pertains even though type property requirements can be prefixed with the class or static keyword when implemented by a class:

1. protocol AnotherProtocol {
2. static var someTypeProperty: Int { get set }
3. }

Here’s an example of a protocol with a single instance property requirement:

1. protocol FullyNamed {
2. var fullName: String { get }
3. }

The FullyNamed protocol requires a conforming type to provide a fully qualified name. The protocol doesn’t specify anything else about the nature of the conforming type—it only specifies that the type must be able to provide a full name for itself. The protocol states that any FullyNamed type must have a gettable instance property called fullName, which is of type String.

Here’s an example of a simple structure that adopts and conforms to the FullyNamed protocol:

1. struct Person: FullyNamed {
2. var fullName: String
3. }
4. let john = Person(fullName: "John Appleseed")
5. // john.fullName is "John Appleseed"

This example defines a structure called Person, which represents a specific named person. It states that it adopts the FullyNamed protocol as part of the first line of its definition.

Each instance of Person has a single stored property called fullName, which is of type String. This matches the single requirement of the FullyNamed protocol, and means that Person has correctly conformed to the protocol. (Swift reports an error at compile-time if a protocol requirement is not fulfilled.)

Here’s a more complex class, which also adopts and conforms to the FullyNamed protocol:

1. class Starship: FullyNamed {
2. var prefix: String?
3. var name: String
4. init(name: String, prefix: String? = nil) {
5. self.name = name
6. self.prefix = prefix
7. }
8. var fullName: String {
9. return (prefix != nil ? prefix! + " " : "") + name
10. }
11. }
12. var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
13. // ncc1701.fullName is "USS Enterprise"

This class implements the fullName property requirement as a computed read-only property for a starship. Each Starship class instance stores a mandatory name and an optional prefix. The fullName property uses the prefix value if it exists, and prepends it to the beginning of name to create a full name for the starship.

## Method Requirements

Protocols can require specific instance methods and type methods to be implemented by conforming types. These methods are written as part of the protocol’s definition in exactly the same way as for normal instance and type methods, but without curly braces or a method body. Variadic parameters are allowed, subject to the same rules as for normal methods. Default values, however, can’t be specified for method parameters within a protocol’s definition.

As with type property requirements, you always prefix type method requirements with the static keyword when they’re defined in a protocol. This is true even though type method requirements are prefixed with the class or static keyword when implemented by a class:

1. protocol SomeProtocol {
2. static func someTypeMethod()
3. }

The following example defines a protocol with a single instance method requirement:

1. protocol RandomNumberGenerator {
2. func random() -> Double
3. }

This protocol, RandomNumberGenerator, requires any conforming type to have an instance method called random, which returns a Double value whenever it’s called. Although it’s not specified as part of the protocol, it’s assumed that this value will be a number from 0.0 up to (but not including) 1.0.

The RandomNumberGenerator protocol doesn’t make any assumptions about how each random number will be generated—it simply requires the generator to provide a standard way to generate a new random number.

Here’s an implementation of a class that adopts and conforms to the RandomNumberGenerator protocol. This class implements a pseudorandom number generator algorithm known as a linear congruential generator:

1. class LinearCongruentialGenerator: RandomNumberGenerator {
2. var lastRandom = 42.0
3. let m = 139968.0
4. let a = 3877.0
5. let c = 29573.0
6. func random() -> Double {
7. lastRandom = ((lastRandom \* a + c)
8. .truncatingRemainder(dividingBy:m))
9. return lastRandom / m
10. }
11. }
12. let generator = LinearCongruentialGenerator()
13. print("Here's a random number: \(generator.random())")
14. // Prints "Here's a random number: 0.3746499199817101"
15. print("And another one: \(generator.random())")
16. // Prints "And another one: 0.729023776863283"

## Mutating Method Requirements

It’s sometimes necessary for a method to modify (or mutate) the instance it belongs to. For instance methods on value types (that is, structures and enumerations) you place the mutating keyword before a method’s func keyword to indicate that the method is allowed to modify the instance it belongs to and any properties of that instance. This process is described in [Modifying Value Types from Within Instance Methods](https://docs.swift.org/swift-book/LanguageGuide/Methods.html#ID239).

If you define a protocol instance method requirement that is intended to mutate instances of any type that adopts the protocol, mark the method with the mutating keyword as part of the protocol’s definition. This enables structures and enumerations to adopt the protocol and satisfy that method requirement.

NOTE

If you mark a protocol instance method requirement as mutating, you don’t need to write the mutating keyword when writing an implementation of that method for a class. The mutating keyword is only used by structures and enumerations.

The example below defines a protocol called Togglable, which defines a single instance method requirement called toggle. As its name suggests, the toggle() method is intended to toggle or invert the state of any conforming type, typically by modifying a property of that type.

The toggle() method is marked with the mutating keyword as part of the Togglable protocol definition, to indicate that the method is expected to mutate the state of a conforming instance when it’s called:

1. protocol Togglable {
2. mutating func toggle()
3. }

If you implement the Togglable protocol for a structure or enumeration, that structure or enumeration can conform to the protocol by providing an implementation of the toggle() method that is also marked as mutating.

The example below defines an enumeration called OnOffSwitch. This enumeration toggles between two states, indicated by the enumeration cases on and off. The enumeration’s toggle implementation is marked as mutating, to match the Togglable protocol’s requirements:

1. enum OnOffSwitch: Togglable {
2. case off, on
3. mutating func toggle() {
4. switch self {
5. case .off:
6. self = .on
7. case .on:
8. self = .off
9. }
10. }
11. }
12. var lightSwitch = OnOffSwitch.off
13. lightSwitch.toggle()
14. // lightSwitch is now equal to .on

## Initializer Requirements

Protocols can require specific initializers to be implemented by conforming types. You write these initializers as part of the protocol’s definition in exactly the same way as for normal initializers, but without curly braces or an initializer body:

1. protocol SomeProtocol {
2. init(someParameter: Int)
3. }

### Class Implementations of Protocol Initializer Requirements

You can implement a protocol initializer requirement on a conforming class as either a designated initializer or a convenience initializer. In both cases, you must mark the initializer implementation with the required modifier:

1. class SomeClass: SomeProtocol {
2. required init(someParameter: Int) {
3. // initializer implementation goes here
4. }
5. }

The use of the required modifier ensures that you provide an explicit or inherited implementation of the initializer requirement on all subclasses of the conforming class, such that they also conform to the protocol.

For more information on required initializers, see [Required Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID231).

NOTE

You don’t need to mark protocol initializer implementations with the required modifier on classes that are marked with the final modifier, because final classes can’t subclassed. For more about the final modifier, see [Preventing Overrides](https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html#ID202).

If a subclass overrides a designated initializer from a superclass, and also implements a matching initializer requirement from a protocol, mark the initializer implementation with both the required and override modifiers:

1. protocol SomeProtocol {
2. init()
3. }
4. class SomeSuperClass {
5. init() {
6. // initializer implementation goes here
7. }
8. }
9. class SomeSubClass: SomeSuperClass, SomeProtocol {
10. // "required" from SomeProtocol conformance; "override" from SomeSuperClass
11. required override init() {
12. // initializer implementation goes here
13. }
14. }

### Failable Initializer Requirements

Protocols can define failable initializer requirements for conforming types, as defined in [Failable Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID224).

A failable initializer requirement can be satisfied by a failable or nonfailable initializer on a conforming type. A nonfailable initializer requirement can be satisfied by a nonfailable initializer or an implicitly unwrapped failable initializer.

## Protocols as Types

Protocols don’t actually implement any functionality themselves. Nonetheless, you can use protocols as a fully fledged types in your code. Using a protocol as a type is sometimes called an existential type, which comes from the phrase “there exists a type T such that T conforms to the protocol”.

You can use a protocol in many places where other types are allowed, including:

* As a parameter type or return type in a function, method, or initializer
* As the type of a constant, variable, or property
* As the type of items in an array, dictionary, or other container

NOTE

Because protocols are types, begin their names with a capital letter (such as FullyNamed and RandomNumberGenerator) to match the names of other types in Swift (such as Int, String, and Double).

Here’s an example of a protocol used as a type:

1. class Dice {
2. let sides: Int
3. let generator: RandomNumberGenerator
4. init(sides: Int, generator: RandomNumberGenerator) {
5. self.sides = sides
6. self.generator = generator
7. }
8. func roll() -> Int {
9. return Int(generator.random() \* Double(sides)) + 1
10. }
11. }

This example defines a new class called Dice, which represents an n-sided dice for use in a board game. Dice instances have an integer property called sides, which represents how many sides they have, and a property called generator, which provides a random number generator from which to create dice roll values.

The generator property is of type RandomNumberGenerator. Therefore, you can set it to an instance of any type that adopts the RandomNumberGenerator protocol. Nothing else is required of the instance you assign to this property, except that the instance must adopt the RandomNumberGenerator protocol. Because its type is RandomNumberGenerator, code inside the Dice class can only interact with generator in ways that apply to all generators that conform to this protocol. That means it can’t use any methods or properties that are defined by the underlying type of the generator. However, you can downcast from a protocol type to an underlying type in the same way you can downcast from a superclass to a subclass, as discussed in [Downcasting](https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html#ID341).

Dice also has an initializer, to set up its initial state. This initializer has a parameter called generator, which is also of type RandomNumberGenerator. You can pass a value of any conforming type in to this parameter when initializing a new Dice instance.

Dice provides one instance method, roll, which returns an integer value between 1 and the number of sides on the dice. This method calls the generator’s random() method to create a new random number between 0.0 and 1.0, and uses this random number to create a dice roll value within the correct range. Because generator is known to adopt RandomNumberGenerator, it’s guaranteed to have a random() method to call.

Here’s how the Dice class can be used to create a six-sided dice with a LinearCongruentialGenerator instance as its random number generator:

1. var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
2. for \_ in 1...5 {
3. print("Random dice roll is \(d6.roll())")
4. }
5. // Random dice roll is 3
6. // Random dice roll is 5
7. // Random dice roll is 4
8. // Random dice roll is 5
9. // Random dice roll is 4

## Delegation

Delegation is a design pattern that enables a class or structure to hand off (or delegate) some of its responsibilities to an instance of another type. This design pattern is implemented by defining a protocol that encapsulates the delegated responsibilities, such that a conforming type (known as a delegate) is guaranteed to provide the functionality that has been delegated. Delegation can be used to respond to a particular action, or to retrieve data from an external source without needing to know the underlying type of that source.

The example below defines two protocols for use with dice-based board games:

1. protocol DiceGame {
2. var dice: Dice { get }
3. func play()
4. }
5. protocol DiceGameDelegate: AnyObject {
6. func gameDidStart(\_ game: DiceGame)
7. func game(\_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)
8. func gameDidEnd(\_ game: DiceGame)
9. }

The DiceGame protocol is a protocol that can be adopted by any game that involves dice.

The DiceGameDelegate protocol can be adopted to track the progress of a DiceGame. To prevent strong reference cycles, delegates are declared as weak references. For information about weak references, see [Strong Reference Cycles Between Class Instances](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID51). Marking the protocol as class-only lets the SnakesAndLadders class later in this chapter declare that its delegate must use a weak reference. A class-only protocol is marked by its inheritance from AnyObject, as discussed in [Class-Only Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID281).

Here’s a version of the Snakes and Ladders game originally introduced in [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html). This version is adapted to use a Dice instance for its dice-rolls; to adopt the DiceGame protocol; and to notify a DiceGameDelegate about its progress:

1. class SnakesAndLadders: DiceGame {
2. let finalSquare = 25
3. let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
4. var square = 0
5. var board: [Int]
6. init() {
7. board = Array(repeating: 0, count: finalSquare + 1)
8. board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
9. board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
10. }
11. weak var delegate: DiceGameDelegate?
12. func play() {
13. square = 0
14. delegate?.gameDidStart(self)
15. gameLoop: while square != finalSquare {
16. let diceRoll = dice.roll()
17. delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
18. switch square + diceRoll {
19. case finalSquare:
20. break gameLoop
21. case let newSquare where newSquare > finalSquare:
22. continue gameLoop
23. default:
24. square += diceRoll
25. square += board[square]
26. }
27. }
28. delegate?.gameDidEnd(self)
29. }
30. }

For a description of the Snakes and Ladders gameplay, see [Break](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html#ID137).

This version of the game is wrapped up as a class called SnakesAndLadders, which adopts the DiceGame protocol. It provides a gettable dice property and a play() method in order to conform to the protocol. (The dice property is declared as a constant property because it doesn’t need to change after initialization, and the protocol only requires that it must be gettable.)

The Snakes and Ladders game board setup takes place within the class’s init() initializer. All game logic is moved into the protocol’s play method, which uses the protocol’s required dice property to provide its dice roll values.

Note that the delegate property is defined as an optional DiceGameDelegate, because a delegate isn’t required in order to play the game. Because it’s of an optional type, the delegate property is automatically set to an initial value of nil. Thereafter, the game instantiator has the option to set the property to a suitable delegate. Because the DiceGameDelegate protocol is class-only, you can declare the delegate to be weak to prevent reference cycles.

DiceGameDelegate provides three methods for tracking the progress of a game. These three methods have been incorporated into the game logic within the play() method above, and are called when a new game starts, a new turn begins, or the game ends.

Because the delegate property is an optional DiceGameDelegate, the play() method uses optional chaining each time it calls a method on the delegate. If the delegate property is nil, these delegate calls fail gracefully and without error. If the delegate property is non-nil, the delegate methods are called, and are passed the SnakesAndLadders instance as a parameter.

This next example shows a class called DiceGameTracker, which adopts the DiceGameDelegate protocol:

1. class DiceGameTracker: DiceGameDelegate {
2. var numberOfTurns = 0
3. func gameDidStart(\_ game: DiceGame) {
4. numberOfTurns = 0
5. if game is SnakesAndLadders {
6. print("Started a new game of Snakes and Ladders")
7. }
8. print("The game is using a \(game.dice.sides)-sided dice")
9. }
10. func game(\_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
11. numberOfTurns += 1
12. print("Rolled a \(diceRoll)")
13. }
14. func gameDidEnd(\_ game: DiceGame) {
15. print("The game lasted for \(numberOfTurns) turns")
16. }
17. }

DiceGameTracker implements all three methods required by DiceGameDelegate. It uses these methods to keep track of the number of turns a game has taken. It resets a numberOfTurns property to zero when the game starts, increments it each time a new turn begins, and prints out the total number of turns once the game has ended.

The implementation of gameDidStart(\_:) shown above uses the game parameter to print some introductory information about the game that is about to be played. The game parameter has a type of DiceGame, not SnakesAndLadders, and so gameDidStart(\_:) can access and use only methods and properties that are implemented as part of the DiceGame protocol. However, the method is still able to use type casting to query the type of the underlying instance. In this example, it checks whether game is actually an instance of SnakesAndLadders behind the scenes, and prints an appropriate message if so.

The gameDidStart(\_:) method also accesses the dice property of the passed game parameter. Because game is known to conform to the DiceGame protocol, it’s guaranteed to have a dice property, and so the gameDidStart(\_:) method is able to access and print the dice’s sides property, regardless of what kind of game is being played.

Here’s how DiceGameTracker looks in action:

1. let tracker = DiceGameTracker()
2. let game = SnakesAndLadders()
3. game.delegate = tracker
4. game.play()
5. // Started a new game of Snakes and Ladders
6. // The game is using a 6-sided dice
7. // Rolled a 3
8. // Rolled a 5
9. // Rolled a 4
10. // Rolled a 5
11. // The game lasted for 4 turns

## Adding Protocol Conformance with an Extension

You can extend an existing type to adopt and conform to a new protocol, even if you don’t have access to the source code for the existing type. Extensions can add new properties, methods, and subscripts to an existing type, and are therefore able to add any requirements that a protocol may demand. For more about extensions, see [Extensions](https://docs.swift.org/swift-book/LanguageGuide/Extensions.html).

NOTE

Existing instances of a type automatically adopt and conform to a protocol when that conformance is added to the instance’s type in an extension.

For example, this protocol, called TextRepresentable, can be implemented by any type that has a way to be represented as text. This might be a description of itself, or a text version of its current state:

1. protocol TextRepresentable {
2. var textualDescription: String { get }
3. }

The Dice class from above can be extended to adopt and conform to TextRepresentable:

1. extension Dice: TextRepresentable {
2. var textualDescription: String {
3. return "A \(sides)-sided dice"
4. }
5. }

This extension adopts the new protocol in exactly the same way as if Dice had provided it in its original implementation. The protocol name is provided after the type name, separated by a colon, and an implementation of all requirements of the protocol is provided within the extension’s curly braces.

Any Dice instance can now be treated as TextRepresentable:

1. let d12 = Dice(sides: 12, generator: LinearCongruentialGenerator())
2. print(d12.textualDescription)
3. // Prints "A 12-sided dice"

Similarly, the SnakesAndLadders game class can be extended to adopt and conform to the TextRepresentable protocol:

1. extension SnakesAndLadders: TextRepresentable {
2. var textualDescription: String {
3. return "A game of Snakes and Ladders with \(finalSquare) squares"
4. }
5. }
6. print(game.textualDescription)
7. // Prints "A game of Snakes and Ladders with 25 squares"

### Conditionally Conforming to a Protocol

A generic type may be able to satisfy the requirements of a protocol only under certain conditions, such as when the type’s generic parameter conforms to the protocol. You can make a generic type conditionally conform to a protocol by listing constraints when extending the type. Write these constraints after the name of the protocol you’re adopting by writing a generic where clause. For more about generic where clauses, see [Generic Where Clauses](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID192).

The following extension makes Array instances conform to the TextRepresentable protocol whenever they store elements of a type that conforms to TextRepresentable.

1. extension Array: TextRepresentable where Element: TextRepresentable {
2. var textualDescription: String {
3. let itemsAsText = self.map { $0.textualDescription }
4. return "[" + itemsAsText.joined(separator: ", ") + "]"
5. }
6. }
7. let myDice = [d6, d12]
8. print(myDice.textualDescription)
9. // Prints "[A 6-sided dice, A 12-sided dice]"

### Declaring Protocol Adoption with an Extension

If a type already conforms to all of the requirements of a protocol, but has not yet stated that it adopts that protocol, you can make it adopt the protocol with an empty extension:

1. struct Hamster {
2. var name: String
3. var textualDescription: String {
4. return "A hamster named \(name)"
5. }
6. }
7. extension Hamster: TextRepresentable {}

Instances of Hamster can now be used wherever TextRepresentable is the required type:

1. let simonTheHamster = Hamster(name: "Simon")
2. let somethingTextRepresentable: TextRepresentable = simonTheHamster
3. print(somethingTextRepresentable.textualDescription)
4. // Prints "A hamster named Simon"

NOTE

Types don’t automatically adopt a protocol just by satisfying its requirements. They must always explicitly declare their adoption of the protocol.

## Collections of Protocol Types

A protocol can be used as the type to be stored in a collection such as an array or a dictionary, as mentioned in [Protocols as Types](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID275). This example creates an array of TextRepresentable things:

1. let things: [TextRepresentable] = [game, d12, simonTheHamster]

It’s now possible to iterate over the items in the array, and print each item’s textual description:

1. for thing in things {
2. print(thing.textualDescription)
3. }
4. // A game of Snakes and Ladders with 25 squares
5. // A 12-sided dice
6. // A hamster named Simon

Note that the thing constant is of type TextRepresentable. It’s not of type Dice, or DiceGame, or Hamster, even if the actual instance behind the scenes is of one of those types. Nonetheless, because it’s of type TextRepresentable, and anything that is TextRepresentable is known to have a textualDescription property, it’s safe to access thing.textualDescription each time through the loop.

## Protocol Inheritance

A protocol can inherit one or more other protocols and can add further requirements on top of the requirements it inherits. The syntax for protocol inheritance is similar to the syntax for class inheritance, but with the option to list multiple inherited protocols, separated by commas:

1. protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
2. // protocol definition goes here
3. }

Here’s an example of a protocol that inherits the TextRepresentable protocol from above:

1. protocol PrettyTextRepresentable: TextRepresentable {
2. var prettyTextualDescription: String { get }
3. }

This example defines a new protocol, PrettyTextRepresentable, which inherits from TextRepresentable. Anything that adopts PrettyTextRepresentable must satisfy all of the requirements enforced by TextRepresentable, plus the additional requirements enforced by PrettyTextRepresentable. In this example, PrettyTextRepresentable adds a single requirement to provide a gettable property called prettyTextualDescription that returns a String.

The SnakesAndLadders class can be extended to adopt and conform to PrettyTextRepresentable:

1. extension SnakesAndLadders: PrettyTextRepresentable {
2. var prettyTextualDescription: String {
3. var output = textualDescription + ":\n"
4. for index in 1...finalSquare {
5. switch board[index] {
6. case let ladder where ladder > 0:
7. output += "▲ "
8. case let snake where snake < 0:
9. output += "▼ "
10. default:
11. output += "○ "
12. }
13. }
14. return output
15. }
16. }

This extension states that it adopts the PrettyTextRepresentable protocol and provides an implementation of the prettyTextualDescription property for the SnakesAndLadders type. Anything that is PrettyTextRepresentable must also be TextRepresentable, and so the implementation of prettyTextualDescription starts by accessing the textualDescription property from the TextRepresentable protocol to begin an output string. It appends a colon and a line break, and uses this as the start of its pretty text representation. It then iterates through the array of board squares, and appends a geometric shape to represent the contents of each square:

* If the square’s value is greater than 0, it’s the base of a ladder, and is represented by ▲.
* If the square’s value is less than 0, it’s the head of a snake, and is represented by ▼.
* Otherwise, the square’s value is 0, and it’s a “free” square, represented by ○.

The prettyTextualDescription property can now be used to print a pretty text description of any SnakesAndLadders instance:

1. print(game.prettyTextualDescription)
2. // A game of Snakes and Ladders with 25 squares:
3. // ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼ ○ ▼ ○

## Class-Only Protocols

You can limit protocol adoption to class types (and not structures or enumerations) by adding the AnyObject protocol to a protocol’s inheritance list.

1. protocol SomeClassOnlyProtocol: AnyObject, SomeInheritedProtocol {
2. // class-only protocol definition goes here
3. }

In the example above, SomeClassOnlyProtocol can only be adopted by class types. It’s a compile-time error to write a structure or enumeration definition that tries to adopt SomeClassOnlyProtocol.

NOTE

Use a class-only protocol when the behavior defined by that protocol’s requirements assumes or requires that a conforming type has reference semantics rather than value semantics. For more about reference and value semantics, see [Structures and Enumerations Are Value Types](https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html#ID88) and [Classes Are Reference Types](https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html#ID89).

## Protocol Composition

It can be useful to require a type to conform to multiple protocols at the same time. You can combine multiple protocols into a single requirement with a protocol composition. Protocol compositions behave as if you defined a temporary local protocol that has the combined requirements of all protocols in the composition. Protocol compositions don’t define any new protocol types.

Protocol compositions have the form SomeProtocol & AnotherProtocol. You can list as many protocols as you need, separating them with ampersands (&). In addition to its list of protocols, a protocol composition can also contain one class type, which you can use to specify a required superclass.

Here’s an example that combines two protocols called Named and Aged into a single protocol composition requirement on a function parameter:

1. protocol Named {
2. var name: String { get }
3. }
4. protocol Aged {
5. var age: Int { get }
6. }
7. struct Person: Named, Aged {
8. var name: String
9. var age: Int
10. }
11. func wishHappyBirthday(to celebrator: Named & Aged) {
12. print("Happy birthday, \(celebrator.name), you're \(celebrator.age)!")
13. }
14. let birthdayPerson = Person(name: "Malcolm", age: 21)
15. wishHappyBirthday(to: birthdayPerson)
16. // Prints "Happy birthday, Malcolm, you're 21!"

In this example, the Named protocol has a single requirement for a gettable String property called name. The Aged protocol has a single requirement for a gettable Int property called age. Both protocols are adopted by a structure called Person.

The example also defines a wishHappyBirthday(to:) function. The type of the celebrator parameter is Named & Aged, which means “any type that conforms to both the Named and Aged protocols.” It doesn’t matter which specific type is passed to the function, as long as it conforms to both of the required protocols.

The example then creates a new Person instance called birthdayPerson and passes this new instance to the wishHappyBirthday(to:) function. Because Person conforms to both protocols, this call is valid, and the wishHappyBirthday(to:) function can print its birthday greeting.

Here’s an example that combines the Named protocol from the previous example with a Location class:

1. class Location {
2. var latitude: Double
3. var longitude: Double
4. init(latitude: Double, longitude: Double) {
5. self.latitude = latitude
6. self.longitude = longitude
7. }
8. }
9. class City: Location, Named {
10. var name: String
11. init(name: String, latitude: Double, longitude: Double) {
12. self.name = name
13. super.init(latitude: latitude, longitude: longitude)
14. }
15. }
16. func beginConcert(in location: Location & Named) {
17. print("Hello, \(location.name)!")
18. }
19. let seattle = City(name: "Seattle", latitude: 47.6, longitude: -122.3)
20. beginConcert(in: seattle)
21. // Prints "Hello, Seattle!"

The beginConcert(in:) function takes a parameter of type Location & Named, which means “any type that’s a subclass of Location and that conforms to the Named protocol.” In this case, City satisfies both requirements.

Passing birthdayPerson to the beginConcert(in:) function is invalid because Person isn’t a subclass of Location. Likewise, if you made a subclass of Location that didn’t conform to the Named protocol, calling beginConcert(in:) with an instance of that type is also invalid.

## Checking for Protocol Conformance

You can use the is and as operators described in [Type Casting](https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html) to check for protocol conformance, and to cast to a specific protocol. Checking for and casting to a protocol follows exactly the same syntax as checking for and casting to a type:

* The is operator returns true if an instance conforms to a protocol and returns false if it doesn’t.
* The as? version of the downcast operator returns an optional value of the protocol’s type, and this value is nil if the instance doesn’t conform to that protocol.
* The as! version of the downcast operator forces the downcast to the protocol type and triggers a runtime error if the downcast doesn’t succeed.

This example defines a protocol called HasArea, with a single property requirement of a gettable Double property called area:

1. protocol HasArea {
2. var area: Double { get }
3. }

Here are two classes, Circle and Country, both of which conform to the HasArea protocol:

1. class Circle: HasArea {
2. let pi = 3.1415927
3. var radius: Double
4. var area: Double { return pi \* radius \* radius }
5. init(radius: Double) { self.radius = radius }
6. }
7. class Country: HasArea {
8. var area: Double
9. init(area: Double) { self.area = area }
10. }

The Circle class implements the area property requirement as a computed property, based on a stored radius property. The Country class implements the area requirement directly as a stored property. Both classes correctly conform to the HasArea protocol.

Here’s a class called Animal, which doesn’t conform to the HasArea protocol:

1. class Animal {
2. var legs: Int
3. init(legs: Int) { self.legs = legs }
4. }

The Circle, Country and Animal classes don’t have a shared base class. Nonetheless, they’re all classes, and so instances of all three types can be used to initialize an array that stores values of type AnyObject:

1. let objects: [AnyObject] = [
2. Circle(radius: 2.0),
3. Country(area: 243\_610),
4. Animal(legs: 4)
5. ]

The objects array is initialized with an array literal containing a Circle instance with a radius of 2 units; a Country instance initialized with the surface area of the United Kingdom in square kilometers; and an Animal instance with four legs.

The objects array can now be iterated, and each object in the array can be checked to see if it conforms to the HasArea protocol:

1. for object in objects {
2. if let objectWithArea = object as? HasArea {
3. print("Area is \(objectWithArea.area)")
4. } else {
5. print("Something that doesn't have an area")
6. }
7. }
8. // Area is 12.5663708
9. // Area is 243610.0
10. // Something that doesn't have an area

Whenever an object in the array conforms to the HasArea protocol, the optional value returned by the as? operator is unwrapped with optional binding into a constant called objectWithArea. The objectWithArea constant is known to be of type HasArea, and so its area property can be accessed and printed in a type-safe way.

Note that the underlying objects aren’t changed by the casting process. They continue to be a Circle, a Country and an Animal. However, at the point that they’re stored in the objectWithArea constant, they’re only known to be of type HasArea, and so only their area property can be accessed.

## Optional Protocol Requirements

You can define optional requirements for protocols. These requirements don’t have to be implemented by types that conform to the protocol. Optional requirements are prefixed by the optional modifier as part of the protocol’s definition. Optional requirements are available so that you can write code that interoperates with Objective-C. Both the protocol and the optional requirement must be marked with the @objc attribute. Note that @objc protocols can be adopted only by classes that inherit from Objective-C classes or other @objc classes. They can’t be adopted by structures or enumerations.

When you use a method or property in an optional requirement, its type automatically becomes an optional. For example, a method of type (Int) -> String becomes ((Int) -> String)?. Note that the entire function type is wrapped in the optional, not the method’s return value.

An optional protocol requirement can be called with optional chaining, to account for the possibility that the requirement was not implemented by a type that conforms to the protocol. You check for an implementation of an optional method by writing a question mark after the name of the method when it’s called, such as someOptionalMethod?(someArgument). For information on optional chaining, see [Optional Chaining](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html).

The following example defines an integer-counting class called Counter, which uses an external data source to provide its increment amount. This data source is defined by the CounterDataSource protocol, which has two optional requirements:

1. @objc protocol CounterDataSource {
2. @objc optional func increment(forCount count: Int) -> Int
3. @objc optional var fixedIncrement: Int { get }
4. }

The CounterDataSource protocol defines an optional method requirement called increment(forCount:) and an optional property requirement called fixedIncrement. These requirements define two different ways for data sources to provide an appropriate increment amount for a Counter instance.

NOTE

Strictly speaking, you can write a custom class that conforms to CounterDataSource without implementing either protocol requirement. They’re both optional, after all. Although technically allowed, this wouldn’t make for a very good data source.

The Counter class, defined below, has an optional dataSource property of type CounterDataSource?:

1. class Counter {
2. var count = 0
3. var dataSource: CounterDataSource?
4. func increment() {
5. if let amount = dataSource?.increment?(forCount: count) {
6. count += amount
7. } else if let amount = dataSource?.fixedIncrement {
8. count += amount
9. }
10. }
11. }

The Counter class stores its current value in a variable property called count. The Counter class also defines a method called increment, which increments the count property every time the method is called.

The increment() method first tries to retrieve an increment amount by looking for an implementation of the increment(forCount:) method on its data source. The increment() method uses optional chaining to try to call increment(forCount:), and passes the current count value as the method’s single argument.

Note that two levels of optional chaining are at play here. First, it’s possible that dataSource may be nil, and so dataSource has a question mark after its name to indicate that increment(forCount:) should be called only if dataSource isn’t nil. Second, even if dataSource does exist, there’s no guarantee that it implements increment(forCount:), because it’s an optional requirement. Here, the possibility that increment(forCount:) might not be implemented is also handled by optional chaining. The call to increment(forCount:) happens only if increment(forCount:) exists—that is, if it isn’t nil. This is why increment(forCount:) is also written with a question mark after its name.

Because the call to increment(forCount:) can fail for either of these two reasons, the call returns an optional Int value. This is true even though increment(forCount:) is defined as returning a non-optional Int value in the definition of CounterDataSource. Even though there are two optional chaining operations, one after another, the result is still wrapped in a single optional. For more information about using multiple optional chaining operations, see [Linking Multiple Levels of Chaining](https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html#ID252).

After calling increment(forCount:), the optional Int that it returns is unwrapped into a constant called amount, using optional binding. If the optional Int does contain a value—that is, if the delegate and method both exist, and the method returned a value—the unwrapped amount is added onto the stored count property, and incrementation is complete.

If it’s not possible to retrieve a value from the increment(forCount:) method—either because dataSource is nil, or because the data source doesn’t implement increment(forCount:)—then the increment() method tries to retrieve a value from the data source’s fixedIncrement property instead. The fixedIncrement property is also an optional requirement, so its value is an optional Int value, even though fixedIncrement is defined as a non-optional Int property as part of the CounterDataSource protocol definition.

Here’s a simple CounterDataSource implementation where the data source returns a constant value of 3 every time it’s queried. It does this by implementing the optional fixedIncrement property requirement:

1. class ThreeSource: NSObject, CounterDataSource {
2. let fixedIncrement = 3
3. }

You can use an instance of ThreeSource as the data source for a new Counter instance:

1. var counter = Counter()
2. counter.dataSource = ThreeSource()
3. for \_ in 1...4 {
4. counter.increment()
5. print(counter.count)
6. }
7. // 3
8. // 6
9. // 9
10. // 12

The code above creates a new Counter instance; sets its data source to be a new ThreeSource instance; and calls the counter’s increment() method four times. As expected, the counter’s count property increases by three each time increment() is called.

Here’s a more complex data source called TowardsZeroSource, which makes a Counter instance count up or down towards zero from its current count value:

1. class TowardsZeroSource: NSObject, CounterDataSource {
2. func increment(forCount count: Int) -> Int {
3. if count == 0 {
4. return 0
5. } else if count < 0 {
6. return 1
7. } else {
8. return -1
9. }
10. }
11. }

The TowardsZeroSource class implements the optional increment(forCount:) method from the CounterDataSource protocol and uses the count argument value to work out which direction to count in. If count is already zero, the method returns 0 to indicate that no further counting should take place.

You can use an instance of TowardsZeroSource with the existing Counter instance to count from -4 to zero. Once the counter reaches zero, no more counting takes place:

1. counter.count = -4
2. counter.dataSource = TowardsZeroSource()
3. for \_ in 1...5 {
4. counter.increment()
5. print(counter.count)
6. }
7. // -3
8. // -2
9. // -1
10. // 0
11. // 0

## Protocol Extensions

Protocols can be extended to provide method, initializer, subscript, and computed property implementations to conforming types. This allows you to define behavior on protocols themselves, rather than in each type’s individual conformance or in a global function.

For example, the RandomNumberGenerator protocol can be extended to provide a randomBool() method, which uses the result of the required random() method to return a random Bool value:

1. extension RandomNumberGenerator {
2. func randomBool() -> Bool {
3. return random() > 0.5
4. }
5. }

By creating an extension on the protocol, all conforming types automatically gain this method implementation without any additional modification.

1. let generator = LinearCongruentialGenerator()
2. print("Here's a random number: \(generator.random())")
3. // Prints "Here's a random number: 0.3746499199817101"
4. print("And here's a random Boolean: \(generator.randomBool())")
5. // Prints "And here's a random Boolean: true"

Protocol extensions can add implementations to conforming types but can’t make a protocol extend or inherit from another protocol. Protocol inheritance is always specified in the protocol declaration itself.

### Providing Default Implementations

You can use protocol extensions to provide a default implementation to any method or computed property requirement of that protocol. If a conforming type provides its own implementation of a required method or property, that implementation will be used instead of the one provided by the extension.

NOTE

Protocol requirements with default implementations provided by extensions are distinct from optional protocol requirements. Although conforming types don’t have to provide their own implementation of either, requirements with default implementations can be called without optional chaining.

For example, the PrettyTextRepresentable protocol, which inherits the TextRepresentable protocol can provide a default implementation of its required prettyTextualDescription property to simply return the result of accessing the textualDescription property:

1. extension PrettyTextRepresentable {
2. var prettyTextualDescription: String {
3. return textualDescription
4. }
5. }

### Adding Constraints to Protocol Extensions

When you define a protocol extension, you can specify constraints that conforming types must satisfy before the methods and properties of the extension are available. You write these constraints after the name of the protocol you’re extending by writing a generic where clause. For more about generic where clauses, see [Generic Where Clauses](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID192).

For example, you can define an extension to the Collection protocol that applies to any collection whose elements conform to the Equatable protocol. By constraining a collection’s elements to the Equatable protocol, a part of the standard library, you can use the == and != operators to check for equality and inequality between two elements.

1. extension Collection where Element: Equatable {
2. func allEqual() -> Bool {
3. for element in self {
4. if element != self.first {
5. return false
6. }
7. }
8. return true
9. }
10. }

The allEqual() method returns true only if all the elements in the collection are equal.

Consider two arrays of integers, one where all the elements are the same, and one where they aren’t:

1. let equalNumbers = [100, 100, 100, 100, 100]
2. let differentNumbers = [100, 100, 200, 100, 200]

Because arrays conform to Collection and integers conform to Equatable, equalNumbers and differentNumbers can use the allEqual() method:

1. print(equalNumbers.allEqual())
2. // Prints "true"
3. print(differentNumbers.allEqual())
4. // Prints "false"

NOTE

If a conforming type satisfies the requirements for multiple constrained extensions that provide implementations for the same method or property, Swift uses the implementation corresponding to the most specialized constraints.

PostgreSQL is the world’s most advanced open source database and the fourth most popular database. In development for more than 20 years, PostgreSQL is managed by a well-organized and highly principled and experienced open source community. It is an object-oriented database that is fully ACID compliant and highly extensible, enabling the community to add new features and capabilities as workload demands evolved.

Because of its extensibility, PostgreSQL offers a wide variety of built-in [PostgreSQL data types](https://www.enterprisedb.com/docs/en/8.4/pg/datatype.html) including JSON, XML, HSTORE (key-value), Geo-spatial (PostGIS), IPv6; flexible indexing, featuring composite indexes, GiST, SP- GiST, GIN; full Text Search, online index reorganization; background workers such as a managed process known as Mongress, which accepts MongoDB queries to interface with Postgres data; a contrib module interface: pgcrypto (data encryption), pg\_trgm (find 'similar' data), HSTORE (schema-less data); and extensive SQL support.

PostgreSQL runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, macOS, Solaris), and Windows. It offers the following programming languages: PL/pgSQL, PL/SQL, Java, Python, Ruby, C/C+, PHP, Perl, Tcl, Scheme. Postgres also offers the following Library interfaces: OCI, libpq, JDBC, ODBC, .NET, Perl, Python, Ruby, C/C+, PHP, Lisp, Scheme, and Qt.

PostgreSQL databases provide enterprise-class database solutions and are used by a wide variety of enterprises across many industries, including financial services, information technology, government and media & communications.

Download PostgreSQL today to enjoy the benefits of open source databases.

## What is IBM Cloud Databases for PostgreSQL?

PostgreSQL is an object-relational SQL database, complemented by powerful enhancements like indexable JSON, publish/subscribe functions and drivers.

#### IBM Cloudant

Cloudant is a scalable JSON document database optimized for web, mobile, IoT and serverless applications. The service is compatible with an open source ecosystem that includes CouchDB, PouchDB and libraries for the most popular web and mobile development stacks.

#### IBM Cloud Databases for MongoDB

IBM Cloud Databases for MongoDB provides a JSON document store that can be scaled vertically and horizontally. It has a rich query and aggregation framework and is accessible through a wide range of application frameworks and languages.

#### IBM Cloud Databases for Elasticsearch

#### IBM CIBM Cloud Databases for Elasticsearch

IBM Cloud Databases for Elasticsearch combines the flexibility of a full-text search engine with the power of a JSON document database’s indexing.

#### IBM Cloud Databases for etcd

IBM Cloud Databases for etcd is a key and value store that developers can use to hold the always-correct data they need to coordinate and manage the server cluster for distributed server configuration management.

# Generics

Generic code enables you to write flexible, reusable functions and types that can work with any type, subject to requirements that you define. You can write code that avoids duplication and expresses its intent in a clear, abstracted manner.

Generics are one of the most powerful features of Swift, and much of the Swift standard library is built with generic code. In fact, you’ve been using generics throughout the Language Guide, even if you didn’t realize it. For example, Swift’s Array and Dictionary types are both generic collections. You can create an array that holds Int values, or an array that holds String values, or indeed an array for any other type that can be created in Swift. Similarly, you can create a dictionary to store values of any specified type, and there are no limitations on what that type can be.

## The Problem That Generics Solve

Here’s a standard, nongeneric function called swapTwoInts(\_:\_:), which swaps two Int values:

1. func swapTwoInts(\_ a: inout Int, \_ b: inout Int) {
2. let temporaryA = a
3. a = b
4. b = temporaryA
5. }

This function makes use of in-out parameters to swap the values of a and b, as described in [In-Out Parameters](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID173).

The swapTwoInts(\_:\_:) function swaps the original value of b into a, and the original value of a into b. You can call this function to swap the values in two Int variables:

1. var someInt = 3
2. var anotherInt = 107
3. swapTwoInts(&someInt, &anotherInt)
4. print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
5. // Prints "someInt is now 107, and anotherInt is now 3"

The swapTwoInts(\_:\_:) function is useful, but it can only be used with Int values. If you want to swap two String values, or two Double values, you have to write more functions, such as the swapTwoStrings(\_:\_:) and swapTwoDoubles(\_:\_:) functions shown below:

1. func swapTwoStrings(\_ a: inout String, \_ b: inout String) {
2. let temporaryA = a
3. a = b
4. b = temporaryA
5. }
6. func swapTwoDoubles(\_ a: inout Double, \_ b: inout Double) {
7. let temporaryA = a
8. a = b
9. b = temporaryA
10. }

You may have noticed that the bodies of the swapTwoInts(\_:\_:), swapTwoStrings(\_:\_:), and swapTwoDoubles(\_:\_:) functions are identical. The only difference is the type of the values that they accept (Int, String, and Double).

It’s more useful, and considerably more flexible, to write a single function that swaps two values of any type. Generic code enables you to write such a function. (A generic version of these functions is defined below.)

NOTE

In all three functions, the types of a and b must be the same. If a and b aren’t of the same type, it isn’t possible to swap their values. Swift is a type-safe language, and doesn’t allow (for example) a variable of type String and a variable of type Double to swap values with each other. Attempting to do so results in a compile-time error.

## The Problem That Generics Solve

Here’s a standard, nongeneric function called swapTwoInts(\_:\_:), which swaps two Int values:

1. func swapTwoInts(\_ a: inout Int, \_ b: inout Int) {
2. let temporaryA = a
3. a = b
4. b = temporaryA
5. }

This function makes use of in-out parameters to swap the values of a and b, as described in [In-Out Parameters](https://docs.swift.org/swift-book/LanguageGuide/Functions.html#ID173).

The swapTwoInts(\_:\_:) function swaps the original value of b into a, and the original value of a into b. You can call this function to swap the values in two Int variables:

1. var someInt = 3
2. var anotherInt = 107
3. swapTwoInts(&someInt, &anotherInt)
4. print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
5. // Prints "someInt is now 107, and anotherInt is now 3"

The swapTwoInts(\_:\_:) function is useful, but it can only be used with Int values. If you want to swap two String values, or two Double values, you have to write more functions, such as the swapTwoStrings(\_:\_:) and swapTwoDoubles(\_:\_:) functions shown below:

1. func swapTwoStrings(\_ a: inout String, \_ b: inout String) {
2. let temporaryA = a
3. a = b
4. b = temporaryA
5. }
6. func swapTwoDoubles(\_ a: inout Double, \_ b: inout Double) {
7. let temporaryA = a
8. a = b
9. b = temporaryA
10. }

You may have noticed that the bodies of the swapTwoInts(\_:\_:), swapTwoStrings(\_:\_:), and swapTwoDoubles(\_:\_:) functions are identical. The only difference is the type of the values that they accept (Int, String, and Double).

It’s more useful, and considerably more flexible, to write a single function that swaps two values of any type. Generic code enables you to write such a function. (A generic version of these functions is defined below.)

NOTE

In all three functions, the types of a and b must be the same. If a and b aren’t of the same type, it isn’t possible to swap their values. Swift is a type-safe language, and doesn’t allow (for example) a variable of type String and a variable of type Double to swap values with each other. Attempting to do so results in a compile-time error.

## Generic Functions

Generic functions can work with any type. Here’s a generic version of the swapTwoInts(\_:\_:) function from above, called swapTwoValues(\_:\_:):

1. func swapTwoValues<T>(\_ a: inout T, \_ b: inout T) {
2. let temporaryA = a
3. a = b
4. b = temporaryA
5. }

The body of the swapTwoValues(\_:\_:) function is identical to the body of the swapTwoInts(\_:\_:) function. However, the first line of swapTwoValues(\_:\_:) is slightly different from swapTwoInts(\_:\_:). Here’s how the first lines compare:

1. func swapTwoInts(\_ a: inout Int, \_ b: inout Int)
2. func swapTwoValues<T>(\_ a: inout T, \_ b: inout T)

The generic version of the function uses a placeholder type name (called T, in this case) instead of an actual type name (such as Int, String, or Double). The placeholder type name doesn’t say anything about what T must be, but it does say that both a and b must be of the same type T, whatever T represents. The actual type to use in place of T is determined each time the swapTwoValues(\_:\_:) function is called.

The other difference between a generic function and a nongeneric function is that the generic function’s name (swapTwoValues(\_:\_:)) is followed by the placeholder type name (T) inside angle brackets (<T>). The brackets tell Swift that T is a placeholder type name within the swapTwoValues(\_:\_:) function definition. Because T is a placeholder, Swift doesn’t look for an actual type called T.

The swapTwoValues(\_:\_:) function can now be called in the same way as swapTwoInts, except that it can be passed two values of any type, as long as both of those values are of the same type as each other. Each time swapTwoValues(\_:\_:) is called, the type to use for T is inferred from the types of values passed to the function.

In the two examples below, T is inferred to be Int and String respectively:

1. var someInt = 3
2. var anotherInt = 107
3. swapTwoValues(&someInt, &anotherInt)
4. // someInt is now 107, and anotherInt is now 3
5. var someString = "hello"
6. var anotherString = "world"
7. swapTwoValues(&someString, &anotherString)
8. // someString is now "world", and anotherString is now "hello"

NOTE

The swapTwoValues(\_:\_:) function defined above is inspired by a generic function called swap, which is part of the Swift standard library, and is automatically made available for you to use in your apps. If you need the behavior of the swapTwoValues(\_:\_:) function in your own code, you can use Swift’s existing swap(\_:\_:) function rather than providing your own implementation.

abase’s indexing.

## Type Parameters

In the swapTwoValues(\_:\_:) example above, the placeholder type T is an example of a type parameter. Type parameters specify and name a placeholder type, and are written immediately after the function’s name, between a pair of matching angle brackets (such as <T>).

Once you specify a type parameter, you can use it to define the type of a function’s parameters (such as the a and b parameters of the swapTwoValues(\_:\_:) function), or as the function’s return type, or as a type annotation within the body of the function. In each case, the type parameter is replaced with an actual type whenever the function is called. (In the swapTwoValues(\_:\_:) example above, T was replaced with Int the first time the function was called, and was replaced with String the second time it was called.)

You can provide more than one type parameter by writing multiple type parameter names within the angle brackets, separated by commas.

## Naming Type Parameters

In most cases, type parameters have descriptive names, such as Key and Value in Dictionary<Key, Value> and Element in Array<Element>, which tells the reader about the relationship between the type parameter and the generic type or function it’s used in. However, when there isn’t a meaningful relationship between them, it’s traditional to name them using single letters such as T, U, and V, such as T in the swapTwoValues(\_:\_:) function above.

NOTE

Always give type parameters upper camel case names (such as T and MyTypeParameter) to indicate that they’re a placeholder for a type, not a value.

## Generic Types

In addition to generic functions, Swift enables you to define your own generic types. These are custom classes, structures, and enumerations that can work with any type, in a similar way to Array and Dictionary.

This section shows you how to write a generic collection type called Stack. A stack is an ordered set of values, similar to an array, but with a more restricted set of operations than Swift’s Array type. An array allows new items to be inserted and removed at any location in the array. A stack, however, allows new items to be appended only to the end of the collection (known as pushing a new value on to the stack). Similarly, a stack allows items to be removed only from the end of the collection (known as popping a value off the stack).

NOTE

The concept of a stack is used by the UINavigationController class to model the view controllers in its navigation hierarchy. You call the UINavigationController class pushViewController(\_:animated:) method to add (or push) a view controller on to the navigation stack, and its popViewControllerAnimated(\_:) method to remove (or pop) a view controller from the navigation stack. A stack is a useful collection model whenever you need a strict “last in, first out” approach to managing a collection.

Here’s how to write a nongeneric version of a stack, in this case for a stack of Int values:

1. struct IntStack {
2. var items = [Int]()
3. mutating func push(\_ item: Int) {
4. items.append(item)
5. }
6. mutating func pop() -> Int {
7. return items.removeLast()
8. }
9. }

This structure uses an Array property called items to store the values in the stack. Stack provides two methods, push and pop, to push and pop values on and off the stack. These methods are marked as mutating, because they need to modify (or *mutate*) the structure’s items array.

## Extending a Generic Type

When you extend a generic type, you don’t provide a type parameter list as part of the extension’s definition. Instead, the type parameter list from the original type definition is available within the body of the extension, and the original type parameter names are used to refer to the type parameters from the original definition.

The following example extends the generic Stack type to add a read-only computed property called topItem, which returns the top item on the stack without popping it from the stack:

1. extension Stack {
2. var topItem: Element? {
3. return items.isEmpty ? nil : items[items.count - 1]
4. }
5. }

The topItem property returns an optional value of type Element. If the stack is empty, topItem returns nil; if the stack isn’t empty, topItem returns the final item in the items array.

Note that this extension doesn’t define a type parameter list. Instead, the Stack type’s existing type parameter name, Element, is used within the extension to indicate the optional type of the topItem computed property.

The topItem computed property can now be used with any Stack instance to access and query its top item without removing it.

1. if let topItem = stackOfStrings.topItem {
2. print("The top item on the stack is \(topItem).")
3. }
4. // Prints "The top item on the stack is tres."

Extensions of a generic type can also include requirements that instances of the extended type must satisfy in order to gain the new functionality, as discussed in [Extensions with a Generic Where Clause](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID553) below.

## Type Constraints

The swapTwoValues(\_:\_:) function and the Stack type can work with any type. However, it’s sometimes useful to enforce certain type constraints on the types that can be used with generic functions and generic types. Type constraints specify that a type parameter must inherit from a specific class, or conform to a particular protocol or protocol composition.

For example, Swift’s Dictionary type places a limitation on the types that can be used as keys for a dictionary. As described in [Dictionaries](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html#ID113), the type of a dictionary’s keys must be hashable. That is, it must provide a way to make itself uniquely representable. Dictionary needs its keys to be hashable so that it can check whether it already contains a value for a particular key. Without this requirement, Dictionary could not tell whether it should insert or replace a value for a particular key, nor would it be able to find a value for a given key that is already in the dictionary.

This requirement is enforced by a type constraint on the key type for Dictionary, which specifies that the key type must conform to the Hashable protocol, a special protocol defined in the Swift standard library. All of Swift’s basic types (such as String, Int, Double, and Bool) are hashable by default.

You can define your own type constraints when creating custom generic types, and these constraints provide much of the power of generic programming. Abstract concepts like Hashable characterize types in terms of their conceptual characteristics, rather than their concrete type.

### Type Constraint Syntax

You write type constraints by placing a single class or protocol constraint after a type parameter’s name, separated by a colon, as part of the type parameter list. The basic syntax for type constraints on a generic function is shown below (although the syntax is the same for generic types):

1. func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {
2. // function body goes here
3. }

The hypothetical function above has two type parameters. The first type parameter, T, has a type constraint that requires T to be a subclass of SomeClass. The second type parameter, U, has a type constraint that requires U to conform to the protocol SomeProtocol.

### Type Constraints in Action

Here’s a nongeneric function called findIndex(ofString:in:), which is given a String value to find and an array of String values within which to find it. The findIndex(ofString:in:) function returns an optional Int value, which will be the index of the first matching string in the array if it’s found, or nil if the string can’t be found:

1. func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {
2. for (index, value) in array.enumerated() {
3. if value == valueToFind {
4. return index
5. }
6. }
7. return nil
8. }

The findIndex(ofString:in:) function can be used to find a string value in an array of strings:

1. let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
2. if let foundIndex = findIndex(ofString: "llama", in: strings) {
3. print("The index of llama is \(foundIndex)")
4. }
5. // Prints "The index of llama is 2"

The principle of finding the index of a value in an array isn’t useful only for strings, however. You can write the same functionality as a generic function by replacing any mention of strings with values of some type T instead.

Here’s how you might expect a generic version of findIndex(ofString:in:), called findIndex(of:in:), to be written. Note that the return type of this function is still Int?, because the function returns an optional index number, not an optional value from the array. Be warned, though—this function doesn’t compile, for reasons explained after the example:

1. func findIndex<T>(of valueToFind: T, in array:[T]) -> Int? {
2. for (index, value) in array.enumerated() {
3. if value == valueToFind {
4. return index
5. }
6. }
7. return nil
8. }

This function doesn’t compile as written above. The problem lies with the equality check, “if value == valueToFind”. Not every type in Swift can be compared with the equal to operator (==). If you create your own class or structure to represent a complex data model, for example, then the meaning of “equal to” for that class or structure isn’t something that Swift can guess for you. Because of this, it isn’t possible to guarantee that this code will work for every possible type T, and an appropriate error is reported when you try to compile the code.

All is not lost, however. The Swift standard library defines a protocol called Equatable, which requires any conforming type to implement the equal to operator (==) and the not equal to operator (!=) to compare any two values of that type. All of Swift’s standard types automatically support the Equatable protocol.

Any type that is Equatable can be used safely with the findIndex(of:in:) function, because it’s guaranteed to support the equal to operator. To express this fact, you write a type constraint of Equatable as part of the type parameter’s definition when you define the function:

1. func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -> Int? {
2. for (index, value) in array.enumerated() {
3. if value == valueToFind {
4. return index
5. }
6. }
7. return nil
8. }

The single type parameter for findIndex(of:in:) is written as T: Equatable, which means “any type T that conforms to the Equatable protocol.”

The findIndex(of:in:) function now compiles successfully and can be used with any type that is Equatable, such as Double or String:

1. let doubleIndex = findIndex(of: 9.3, in: [3.14159, 0.1, 0.25])
2. // doubleIndex is an optional Int with no value, because 9.3 isn't in the array
3. let stringIndex = findIndex(of: "Andrea", in: ["Mike", "Malcolm", "Andrea"])
4. // stringIndex is an optional Int containing a value of 2

## Associated Types

When defining a protocol, it’s sometimes useful to declare one or more associated types as part of the protocol’s definition. An associated type gives a placeholder name to a type that is used as part of the protocol. The actual type to use for that associated type isn’t specified until the protocol is adopted. Associated types are specified with the associatedtype keyword.

### Associated Types in Action

Here’s an example of a protocol called Container, which declares an associated type called Item:

1. protocol Container {
2. associatedtype Item
3. mutating func append(\_ item: Item)
4. var count: Int { get }
5. subscript(i: Int) -> Item { get }
6. }

The Container protocol defines three required capabilities that any container must provide:

* It must be possible to add a new item to the container with an append(\_:) method.
* It must be possible to access a count of the items in the container through a count property that returns an Int value.
* It must be possible to retrieve each item in the container with a subscript that takes an Int index value.

This protocol doesn’t specify how the items in the container should be stored or what type they’re allowed to be. The protocol only specifies the three bits of functionality that any type must provide in order to be considered a Container. A conforming type can provide additional functionality, as long as it satisfies these three requirements.

Any type that conforms to the Container protocol must be able to specify the type of values it stores. Specifically, it must ensure that only items of the right type are added to the container, and it must be clear about the type of the items returned by its subscript.

To define these requirements, the Container protocol needs a way to refer to the type of the elements that a container will hold, without knowing what that type is for a specific container. The Container protocol needs to specify that any value passed to the append(\_:) method must have the same type as the container’s element type, and that the value returned by the container’s subscript will be of the same type as the container’s element type.

To achieve this, the Container protocol declares an associated type called Item, written as associatedtype Item. The protocol doesn’t define what Item is—that information is left for any conforming type to provide. Nonetheless, the Item alias provides a way to refer to the type of the items in a Container, and to define a type for use with the append(\_:) method and subscript, to ensure that the expected behavior of any Container is enforced.

Here’s a version of the nongeneric IntStack type from [Generic Types](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID184) above, adapted to conform to the Container protocol:

1. struct IntStack: Container {
2. // original IntStack implementation
3. var items = [Int]()
4. mutating func push(\_ item: Int) {
5. items.append(item)
6. }
7. mutating func pop() -> Int {
8. return items.removeLast()
9. }
10. // conformance to the Container protocol
11. typealias Item = Int
12. mutating func append(\_ item: Int) {
13. self.push(item)
14. }
15. var count: Int {
16. return items.count
17. }
18. subscript(i: Int) -> Int {
19. return items[i]
20. }
21. }

The IntStack type implements all three of the Container protocol’s requirements, and in each case wraps part of the IntStack type’s existing functionality to satisfy these requirements.

Moreover, IntStack specifies that for this implementation of Container, the appropriate Item to use is a type of Int. The definition of typealias Item = Int turns the abstract type of Item into a concrete type of Int for this implementation of the Container protocol.

Thanks to Swift’s type inference, you don’t actually need to declare a concrete Item of Int as part of the definition of IntStack. Because IntStack conforms to all of the requirements of the Container protocol, Swift can infer the appropriate Item to use, simply by looking at the type of the append(\_:) method’s item parameter and the return type of the subscript. Indeed, if you delete the typealias Item = Int line from the code above, everything still works, because it’s clear what type should be used for Item.

You can also make the generic Stack type conform to the Container protocol:

1. struct Stack<Element>: Container {
2. // original Stack<Element> implementation
3. var items = [Element]()
4. mutating func push(\_ item: Element) {
5. items.append(item)
6. }
7. mutating func pop() -> Element {
8. return items.removeLast()
9. }
10. // conformance to the Container protocol
11. mutating func append(\_ item: Element) {
12. self.push(item)
13. }
14. var count: Int {
15. return items.count
16. }
17. subscript(i: Int) -> Element {
18. return items[i]
19. }
20. }

This time, the type parameter Element is used as the type of the append(\_:) method’s item parameter and the return type of the subscript. Swift can therefore infer that Element is the appropriate type to use as the Item for this particular container.

### Extending an Existing Type to Specify an Associated Type

You can extend an existing type to add conformance to a protocol, as described in [Adding Protocol Conformance with an Extension](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID277). This includes a protocol with an associated type.

Swift’s Array type already provides an append(\_:) method, a count property, and a subscript with an Int index to retrieve its elements. These three capabilities match the requirements of the Container protocol. This means that you can extend Array to conform to the Container protocol simply by declaring that Array adopts the protocol. You do this with an empty extension, as described in [Declaring Protocol Adoption with an Extension](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID278):

1. extension Array: Container {}

Array’s existing append(\_:) method and subscript enable Swift to infer the appropriate type to use for Item, just as for the generic Stack type above. After defining this extension, you can use any Array as a Container.

### Adding Constraints to an Associated Type

You can add type constraints to an associated type in a protocol to require that conforming types satisfy those constraints. For example, the following code defines a version of Container that requires the items in the container to be equatable.

1. protocol Container {
2. associatedtype Item: Equatable
3. mutating func append(\_ item: Item)
4. var count: Int { get }
5. subscript(i: Int) -> Item { get }
6. }

To conform to this version of Container, the container’s Item type has to conform to the Equatable protocol.

### Using a Protocol in Its Associated Type’s Constraints

A protocol can appear as part of its own requirements. For example, here’s a protocol that refines the Container protocol, adding the requirement of a suffix(\_:) method. The suffix(\_:) method returns a given number of elements from the end of the container, storing them in an instance of the Suffix type.

1. protocol SuffixableContainer: Container {
2. associatedtype Suffix: SuffixableContainer where Suffix.Item == Item
3. func suffix(\_ size: Int) -> Suffix
4. }

In this protocol, Suffix is an associated type, like the Item type in the Container example above. Suffix has two constraints: It must conform to the SuffixableContainer protocol (the protocol currently being defined), and its Item type must be the same as the container’s Item type. The constraint on Item is a generic where clause, which is discussed in [Associated Types with a Generic Where Clause](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID557) below.

Here’s an extension of the Stack type from [Generic Types](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID184) above that adds conformance to the SuffixableContainer protocol:

1. extension Stack: SuffixableContainer {
2. func suffix(\_ size: Int) -> Stack {
3. var result = Stack()
4. for index in (count-size)..<count {
5. result.append(self[index])
6. }
7. return result
8. }
9. // Inferred that Suffix is Stack.
10. }
11. var stackOfInts = Stack<Int>()
12. stackOfInts.append(10)
13. stackOfInts.append(20)
14. stackOfInts.append(30)
15. let suffix = stackOfInts.suffix(2)
16. // suffix contains 20 and 30

In the example above, the Suffix associated type for Stack is also Stack, so the suffix operation on Stack returns another Stack. Alternatively, a type that conforms to SuffixableContainer can have a Suffix type that’s different from itself—meaning the suffix operation can return a different type. For example, here’s an extension to the nongeneric IntStack type that adds SuffixableContainer conformance, using Stack<Int> as its suffix type instead of IntStack:

1. extension IntStack: SuffixableContainer {
2. func suffix(\_ size: Int) -> Stack<Int> {
3. var result = Stack<Int>()
4. for index in (count-size)..<count {
5. result.append(self[index])
6. }
7. return result
8. }
9. // Inferred that Suffix is Stack<Int>.
10. }

## Generic Where Clauses

Type constraints, as described in [Type Constraints](https://docs.swift.org/swift-book/LanguageGuide/Generics.html#ID186), enable you to define requirements on the type parameters associated with a generic function, subscript, or type.

It can also be useful to define requirements for associated types. You do this by defining a generic where clause. A generic where clause enables you to require that an associated type must conform to a certain protocol, or that certain type parameters and associated types must be the same. A generic where clause starts with the where keyword, followed by constraints for associated types or equality relationships between types and associated types. You write a generic where clause right before the opening curly brace of a type or function’s body.

The example below defines a generic function called allItemsMatch, which checks to see if two Container instances contain the same items in the same order. The function returns a Boolean value of true if all items match and a value of false if they don’t.

The two containers to be checked don’t have to be the same type of container (although they can be), but they do have to hold the same type of items. This requirement is expressed through a combination of type constraints and a generic where clause:

1. func allItemsMatch<C1: Container, C2: Container>
2. (\_ someContainer: C1, \_ anotherContainer: C2) -> Bool
3. where C1.Item == C2.Item, C1.Item: Equatable {
4. // Check that both containers contain the same number of items.
5. if someContainer.count != anotherContainer.count {
6. return false
7. }
8. // Check each pair of items to see if they're equivalent.
9. for i in 0..<someContainer.count {
10. if someContainer[i] != anotherContainer[i] {
11. return false
12. }
13. }
14. // All items match, so return true.
15. return true
16. }

This function takes two arguments called someContainer and anotherContainer. The someContainer argument is of type C1, and the anotherContainer argument is of type C2. Both C1 and C2 are type parameters for two container types to be determined when the function is called.

The following requirements are placed on the function’s two type parameters:

* C1 must conform to the Container protocol (written as C1: Container).
* C2 must also conform to the Container protocol (written as C2: Container).
* The Item for C1 must be the same as the Item for C2 (written as C1.Item == C2.Item).
* The Item for C1 must conform to the Equatable protocol (written as C1.Item: Equatable).

The first and second requirements are defined in the function’s type parameter list, and the third and fourth requirements are defined in the function’s generic where clause.

These requirements mean:

* someContainer is a container of type C1.
* anotherContainer is a container of type C2.
* someContainer and anotherContainer contain the same type of items.
* The items in someContainer can be checked with the not equal operator (!=) to see if they’re different from each other.

The third and fourth requirements combine to mean that the items in anotherContainer can also be checked with the != operator, because they’re exactly the same type as the items in someContainer.

These requirements enable the allItemsMatch(\_:\_:) function to compare the two containers, even if they’re of a different container type.

The allItemsMatch(\_:\_:) function starts by checking that both containers contain the same number of items. If they contain a different number of items, there’s no way that they can match, and the function returns false.

After making this check, the function iterates over all of the items in someContainer with a for-in loop and the half-open range operator (..<). For each item, the function checks whether the item from someContainer isn’t equal to the corresponding item in anotherContainer. If the two items aren’t equal, then the two containers don’t match, and the function returns false.

If the loop finishes without finding a mismatch, the two containers match, and the function returns true.

Here’s how the allItemsMatch(\_:\_:) function looks in action:

1. var stackOfStrings = Stack<String>()
2. stackOfStrings.push("uno")
3. stackOfStrings.push("dos")
4. stackOfStrings.push("tres")
5. var arrayOfStrings = ["uno", "dos", "tres"]
6. if allItemsMatch(stackOfStrings, arrayOfStrings) {
7. print("All items match.")
8. } else {
9. print("Not all items match.")
10. }
11. // Prints "All items match."

The example above creates a Stack instance to store String values, and pushes three strings onto the stack. The example also creates an Array instance initialized with an array literal containing the same three strings as the stack. Even though the stack and the array are of a different type, they both conform to the Container protocol, and both contain the same type of values. You can therefore call the allItemsMatch(\_:\_:) function with these two containers as its arguments. In the example above, the allItemsMatch(\_:\_:) function correctly reports that all of the items in the two containers match.

## Extensions with a Generic Where Clause

You can also use a generic where clause as part of an extension. The example below extends the generic Stack structure from the previous examples to add an isTop(\_:) method.

1. extension Stack where Element: Equatable {
2. func isTop(\_ item: Element) -> Bool {
3. guard let topItem = items.last else {
4. return false
5. }
6. return topItem == item
7. }
8. }

This new isTop(\_:) method first checks that the stack isn’t empty, and then compares the given item against the stack’s topmost item. If you tried to do this without a generic where clause, you would have a problem: The implementation of isTop(\_:) uses the == operator, but the definition of Stack doesn’t require its items to be equatable, so using the == operator results in a compile-time error. Using a generic where clause lets you add a new requirement to the extension, so that the extension adds the isTop(\_:) method only when the items in the stack are equatable.

Here’s how the isTop(\_:) method looks in action:

1. if stackOfStrings.isTop("tres") {
2. print("Top element is tres.")
3. } else {
4. print("Top element is something else.")
5. }
6. // Prints "Top element is tres."

If you try to call the isTop(\_:) method on a stack whose elements aren’t equatable, you’ll get a compile-time error.

1. struct NotEquatable { }
2. var notEquatableStack = Stack<NotEquatable>()
3. let notEquatableValue = NotEquatable()
4. notEquatableStack.push(notEquatableValue)
5. notEquatableStack.isTop(notEquatableValue) // Error

You can use a generic where clause with extensions to a protocol. The example below extends the Container protocol from the previous examples to add a startsWith(\_:) method.

1. extension Container where Item: Equatable {
2. func startsWith(\_ item: Item) -> Bool {
3. return count >= 1 && self[0] == item
4. }
5. }

The startsWith(\_:) method first makes sure that the container has at least one item, and then it checks whether the first item in the container matches the given item. This new startsWith(\_:) method can be used with any type that conforms to the Container protocol, including the stacks and arrays used above, as long as the container’s items are equatable.

1. if [9, 9, 9].startsWith(42) {
2. print("Starts with 42.")
3. } else {
4. print("Starts with something else.")
5. }
6. // Prints "Starts with something else."

The generic where clause in the example above requires Item to conform to a protocol, but you can also write a generic where clauses that require Item to be a specific type. For example:

1. extension Container where Item == Double {
2. func average() -> Double {
3. var sum = 0.0
4. for index in 0..<count {
5. sum += self[index]
6. }
7. return sum / Double(count)
8. }
9. }
10. print([1260.0, 1200.0, 98.6, 37.0].average())
11. // Prints "648.9"

This example adds an average() method to containers whose Item type is Double. It iterates over the items in the container to add them up, and divides by the container’s count to compute the average. It explicitly converts the count from Int to Double to be able to do floating-point division.

You can include multiple requirements in a generic where clause that is part of an extension, just like you can for a generic where clause that you write elsewhere. Separate each requirement in the list with a comma.

## Associated Types with a Generic Where Clause

You can include a generic where clause on an associated type. For example, suppose you want to make a version of Container that includes an iterator, like what the Sequence protocol uses in the standard library. Here’s how you write that:

1. protocol Container {
2. associatedtype Item
3. mutating func append(\_ item: Item)
4. var count: Int { get }
5. subscript(i: Int) -> Item { get }
6. associatedtype Iterator: IteratorProtocol where Iterator.Element == Item
7. func makeIterator() -> Iterator
8. }

The generic where clause on Iterator requires that the iterator must traverse over elements of the same item type as the container’s items, regardless of the iterator’s type. The makeIterator() function provides access to a container’s iterator.

For a protocol that inherits from another protocol, you add a constraint to an inherited associated type by including the generic where clause in the protocol declaration. For example, the following code declares a ComparableContainer protocol that requires Item to conform to Comparable:

1. protocol ComparableContainer: Container where Item: Comparable { }

## Generic Subscripts

Subscripts can be generic, and they can include generic where clauses. You write the placeholder type name inside angle brackets after subscript, and you write a generic where clause right before the opening curly brace of the subscript’s body. For example:

1. extension Container {
2. subscript<Indices: Sequence>(indices: Indices) -> [Item]
3. where Indices.Iterator.Element == Int {
4. var result = [Item]()
5. for index in indices {
6. result.append(self[index])
7. }
8. return result
9. }
10. }

This extension to the Container protocol adds a subscript that takes a sequence of indices and returns an array containing the items at each given index. This generic subscript is constrained as follows:

* The generic parameter Indices in angle brackets has to be a type that conforms to the Sequence protocol from the standard library.
* The subscript takes a single parameter, indices, which is an instance of that Indices type.
* The generic where clause requires that the iterator for the sequence must traverse over elements of type Int. This ensures that the indices in the sequence are the same type as the indices used for a container.

Taken together, these constraints mean that the value passed for the indices parameter is a sequence of integers.

# Opaque Types

A function or method with an opaque return type hides its return value’s type information. Instead of providing a concrete type as the function’s return type, the return value is described in terms of the protocols it supports. Hiding type information is useful at boundaries between a module and code that calls into the module, because the underlying type of the return value can remain private. Unlike returning a value whose type is a protocol type, opaque types preserve type identity—the compiler has access to the type information, but clients of the module don’t.

## The Problem That Opaque Types Solve

For example, suppose you’re writing a module that draws ASCII art shapes. The basic characteristic of an ASCII art shape is a draw() function that returns the string representation of that shape, which you can use as the requirement for the Shape protocol:

1. protocol Shape {
2. func draw() -> String
3. }
4. struct Triangle: Shape {
5. var size: Int
6. func draw() -> String {
7. var result = [String]()
8. for length in 1...size {
9. result.append(String(repeating: "\*", count: length))
10. }
11. return result.joined(separator: "\n")
12. }
13. }
14. let smallTriangle = Triangle(size: 3)
15. print(smallTriangle.draw())
16. // \*
17. // \*\*
18. // \*\*\*

You could use generics to implement operations like flipping a shape vertically, as shown in the code below. However, there’s an important limitation to this approach: The flipped result exposes the exact generic types that were used to create it.

1. struct FlippedShape<T: Shape>: Shape {
2. var shape: T
3. func draw() -> String {
4. let lines = shape.draw().split(separator: "\n")
5. return lines.reversed().joined(separator: "\n")
6. }
7. }
8. let flippedTriangle = FlippedShape(shape: smallTriangle)
9. print(flippedTriangle.draw())
10. // \*\*\*
11. // \*\*
12. // \*

This approach to defining a JoinedShape<T: Shape, U: Shape> structure that joins two shapes together vertically, like the code below shows, results in types like JoinedShape<FlippedShape<Triangle>, Triangle> from joining a flipped triangle with another triangle.

1. struct JoinedShape<T: Shape, U: Shape>: Shape {
2. var top: T
3. var bottom: U
4. func draw() -> String {
5. return top.draw() + "\n" + bottom.draw()
6. }
7. }
8. let joinedTriangles = JoinedShape(top: smallTriangle, bottom: flippedTriangle)
9. print(joinedTriangles.draw())
10. // \*
11. // \*\*
12. // \*\*\*
13. // \*\*\*
14. // \*\*
15. // \*

Exposing detailed information about the creation of a shape allows types that aren’t meant to be part of the ASCII art module’s public interface to leak out because of the need to state the full return type. The code inside the module could build up the same shape in a variety of ways, and other code outside the module that uses the shape shouldn’t have to account for the implementation details about the list of transformations. Wrapper types like JoinedShape and FlippedShape don’t matter to the module’s users, and they shouldn’t be visible. The module’s public interface consists of operations like joining and flipping a shape, and those operations return another Shape value.

## Returning an Opaque Type

You can think of an opaque type like being the reverse of a generic type. Generic types let the code that calls a function pick the type for that function’s parameters and return value in a way that’s abstracted away from the function implementation. For example, the function in the following code returns a type that depends on its caller:

1. func max<T>(\_ x: T, \_ y: T) -> T where T: Comparable { ... }

The code that calls max(\_:\_:) chooses the values for x and y, and the type of those values determines the concrete type of T. The calling code can use any type that conforms to the Comparable protocol. The code inside the function is written in a general way so it can handle whatever type the caller provides. The implementation of max(\_:\_:) uses only functionality that all Comparable types share.

Those roles are reversed for a function with an opaque return type. An opaque type lets the function implementation pick the type for the value it returns in a way that’s abstracted away from the code that calls the function. For example, the function in the following example returns a trapezoid without exposing the underlying type of that shape.

1. struct Square: Shape {
2. var size: Int
3. func draw() -> String {
4. let line = String(repeating: "\*", count: size)
5. let result = Array<String>(repeating: line, count: size)
6. return result.joined(separator: "\n")
7. }
8. }
9. func makeTrapezoid() -> some Shape {
10. let top = Triangle(size: 2)
11. let middle = Square(size: 2)
12. let bottom = FlippedShape(shape: top)
13. let trapezoid = JoinedShape(
14. top: top,
15. bottom: JoinedShape(top: middle, bottom: bottom)
16. )
17. return trapezoid
18. }
19. let trapezoid = makeTrapezoid()
20. print(trapezoid.draw())
21. // \*
22. // \*\*
23. // \*\*
24. // \*\*
25. // \*\*
26. // \*

The makeTrapezoid() function in this example declares its return type as some Shape; as a result, the function returns a value of some given type that conforms to the Shape protocol, without specifying any particular concrete type. Writing makeTrapezoid() this way lets it express the fundamental aspect of its public interface—the value it returns is a shape—without making the specific types that the shape is made from a part of its public interface. This implementation uses two triangles and a square, but the function could be rewritten to draw a trapezoid in a variety of other ways without changing its return type.

This example highlights the way that an opaque return type is like the reverse of a generic type. The code inside makeTrapezoid() can return any type it needs to, as long as that type conforms to the Shape protocol, like the calling code does for a generic function. The code that calls the function needs to be written in a general way, like the implementation of a generic function, so that it can work with any Shape value that’s returned by makeTrapezoid().

You can also combine opaque return types with generics. The functions in the following code both return a value of some type that conforms to the Shape protocol.

1. func flip<T: Shape>(\_ shape: T) -> some Shape {
2. return FlippedShape(shape: shape)
3. }
4. func join<T: Shape, U: Shape>(\_ top: T, \_ bottom: U) -> some Shape {
5. JoinedShape(top: top, bottom: bottom)
6. }
7. let opaqueJoinedTriangles = join(smallTriangle, flip(smallTriangle))
8. print(opaqueJoinedTriangles.draw())
9. // \*
10. // \*\*
11. // \*\*\*
12. // \*\*\*
13. // \*\*
14. // \*

The value of opaqueJoinedTriangles in this example is the same as joinedTriangles in the generics example in the [The Problem That Opaque Types Solve](https://docs.swift.org/swift-book/LanguageGuide/OpaqueTypes.html#ID613) section earlier in this chapter. However, unlike the value in that example, flip(\_:) and join(\_:\_:) wrap the underlying types that the generic shape operations return in an opaque return type, which prevents those types from being visible. Both functions are generic because the types they rely on are generic, and the type parameters to the function pass along the type information needed by FlippedShape and JoinedShape.

If a function with an opaque return type returns from multiple places, all of the possible return values must have the same type. For a generic function, that return type can use the function’s generic type parameters, but it must still be a single type. For example, here’s an invalid version of the shape-flipping function that includes a special case for squares:

1. func invalidFlip<T: Shape>(\_ shape: T) -> some Shape {
2. if shape is Square {
3. return shape // Error: return types don't match
4. }
5. return FlippedShape(shape: shape) // Error: return types don't match
6. }

If you call this function with a Square, it returns a Square; otherwise, it returns a FlippedShape. This violates the requirement to return values of only one type and makes invalidFlip(\_:) invalid code. One way to fix invalidFlip(\_:) is to move the special case for squares into the implementation of FlippedShape, which lets this function always return a FlippedShape value:

1. struct FlippedShape<T: Shape>: Shape {
2. var shape: T
3. func draw() -> String {
4. if shape is Square {
5. return shape.draw()
6. }
7. let lines = shape.draw().split(separator: "\n")
8. return lines.reversed().joined(separator: "\n")
9. }
10. }

The requirement to always return a single type doesn’t prevent you from using generics in an opaque return type. Here’s an example of a function that incorporates its type parameter into the underlying type of the value it returns:

1. func `repeat`<T: Shape>(shape: T, count: Int) -> some Collection {
2. return Array<T>(repeating: shape, count: count)
3. }

In this case, the underlying type of the return value varies depending on T: Whatever shape is passed it, repeat(shape:count:) creates and returns an array of that shape. Nevertheless, the return value always has the same underlying type of [T], so it follows the requirement that functions with opaque return types must return values of only a single type.

## Differences Between Opaque Types and Protocol Types

Returning an opaque type looks very similar to using a protocol type as the return type of a function, but these two kinds of return type differ in whether they preserve type identity. An opaque type refers to one specific type, although the caller of the function isn’t able to see which type; a protocol type can refer to any type that conforms to the protocol. Generally speaking, protocol types give you more flexibility about the underlying types of the values they store, and opaque types let you make stronger guarantees about those underlying types.

For example, here’s a version of flip(\_:) that returns a value of protocol type instead of using an opaque return type:

1. func protoFlip<T: Shape>(\_ shape: T) -> Shape {
2. return FlippedShape(shape: shape)
3. }

This version of protoFlip(\_:) has the same body as flip(\_:), and it always returns a value of the same type. Unlike flip(\_:), the value that protoFlip(\_:) returns isn’t required to always have the same type—it just has to conform to the Shape protocol. Put another way, protoFlip(\_:) makes a much looser API contract with its caller than flip(\_:) makes. It reserves the flexibility to return values of multiple types:

1. func protoFlip<T: Shape>(\_ shape: T) -> Shape {
2. if shape is Square {
3. return shape
4. }
5. return FlippedShape(shape: shape)
6. }

The revised version of the code returns an instance of Square or an instance of FlippedShape, depending on what shape is passed in. Two flipped shapes returned by this function might have completely different types. Other valid versions of this function could return values of different types when flipping multiple instances of the same shape. The less specific return type information from protoFlip(\_:) means that many operations that depend on type information aren’t available on the returned value. For example, it’s not possible to write an == operator comparing results returned by this function.

1. let protoFlippedTriangle = protoFlip(smallTriangle)
2. let sameThing = protoFlip(smallTriangle)
3. protoFlippedTriangle == sameThing // Error

The error on the last line of the example occurs for several reasons. The immediate issue is that the Shape doesn’t include an == operator as part of its protocol requirements. If you try adding one, the next issue you’ll encounter is that the == operator needs to know the types of its left-hand and right-hand arguments. This sort of operator usually takes arguments of type Self, matching whatever concrete type adopts the protocol, but adding a Self requirement to the protocol doesn’t allow for the type erasure that happens when you use the protocol as a type.

Using a protocol type as the return type for a function gives you the flexibility to return any type that conforms to the protocol. However, the cost of that flexibility is that some operations aren’t possible on the returned values. The example shows how the == operator isn’t available—it depends on specific type information that isn’t preserved by using a protocol type.

Another problem with this approach is that the shape transformations don’t nest. The result of flipping a triangle is a value of type Shape, and the protoFlip(\_:) function takes an argument of some type that conforms to the Shape protocol. However, a value of a protocol type doesn’t conform to that protocol; the value returned by protoFlip(\_:) doesn’t conform to Shape. This means code like protoFlip(protoFlip(smallTriange)) that applies multiple transformations is invalid because the flipped shape isn’t a valid argument to protoFlip(\_:).

In contrast, opaque types preserve the identity of the underlying type. Swift can infer associated types, which lets you use an opaque return value in places where a protocol type can’t be used as a return value. For example, here’s a version of the Container protocol from [Generics](https://docs.swift.org/swift-book/LanguageGuide/Generics.html):

1. protocol Container {
2. associatedtype Item
3. var count: Int { get }
4. subscript(i: Int) -> Item { get }
5. }
6. extension Array: Container { }

You can’t use Container as the return type of a function because that protocol has an associated type. You also can’t use it as constraint a generic return type because there isn’t enough information outside the function body to infer what the generic type needs to be.

1. // Error: Protocol with associated types can't be used as a return type.
2. func makeProtocolContainer<T>(item: T) -> Container {
3. return [item]
4. }
5. // Error: Not enough information to infer C.
6. func makeProtocolContainer<T, C: Container>(item: T) -> C {
7. return [item]
8. }

Using the opaque type some Container as a return type expresses the desired API contract—the function returns a container, but declines to specify the container’s type:

1. func makeOpaqueContainer<T>(item: T) -> some Container {
2. return [item]
3. }
4. let opaqueContainer = makeOpaqueContainer(item: 12)
5. let twelve = opaqueContainer[0]
6. print(type(of: twelve))
7. // Prints "Int"

The type of twelve is inferred to be Int, which illustrates the fact that type inference works with opaque types. In the implementation of makeOpaqueContainer(item:), the underlying type of the opaque container is [T]. In this case, T is Int, so the return value is an array of integers and the Item associated type is inferred to be Int. The subscript on Container returns Item, which means that the type of twelve is also inferred to be Int.

## How ARC Works

Every time you create a new instance of a class, ARC allocates a chunk of memory to store information about that instance. This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance.

Additionally, when an instance is no longer needed, ARC frees up the memory used by that instance so that the memory can be used for other purposes instead. This ensures that class instances do not take up space in memory when they are no longer needed.

However, if ARC were to deallocate an instance that was still in use, it would no longer be possible to access that instance’s properties, or call that instance’s methods. Indeed, if you tried to access the instance, your app would most likely crash.

To make sure that instances don’t disappear while they are still needed, ARC tracks how many properties, constants, and variables are currently referring to each class instance. ARC will not deallocate an instance as long as at least one active reference to that instance still exists.

To make this possible, whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a strong reference to the instance. The reference is called a “strong” reference because it keeps a firm hold on that instance, and does not allow it to be deallocated for as long as that strong reference remains.

## Strong Reference Cycles Between Class Instances

In the examples above, ARC is able to track the number of references to the new Person instance you create and to deallocate that Person instance when it’s no longer needed.

However, it’s possible to write code in which an instance of a class never gets to a point where it has zero strong references. This can happen if two class instances hold a strong reference to each other, such that each instance keeps the other alive. This is known as a strong reference cycle.

You resolve strong reference cycles by defining some of the relationships between classes as weak or unowned references instead of as strong references. This process is described in [Resolving Strong Reference Cycles Between Class Instances](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID52). However, before you learn how to resolve a strong reference cycle, it’s useful to understand how such a cycle is caused.

Here’s an example of how a strong reference cycle can be created by accident. This example defines two classes called Person and Apartment, which model a block of apartments and its residents:

1. class Person {
2. let name: String
3. init(name: String) { self.name = name }
4. var apartment: Apartment?
5. deinit { print("\(name) is being deinitialized") }
6. }
7. class Apartment {
8. let unit: String
9. init(unit: String) { self.unit = unit }
10. var tenant: Person?
11. deinit { print("Apartment \(unit) is being deinitialized") }
12. }

Every Person instance has a name property of type String and an optional apartment property that is initially nil. The apartment property is optional, because a person may not always have an apartment.

Similarly, every Apartment instance has a unit property of type String and has an optional tenant property that is initially nil. The tenant property is optional because an apartment may not always have a tenant.

Both of these classes also define a deinitializer, which prints the fact that an instance of that class is being deinitialized. This enables you to see whether instances of Person and Apartment are being deallocated as expected.

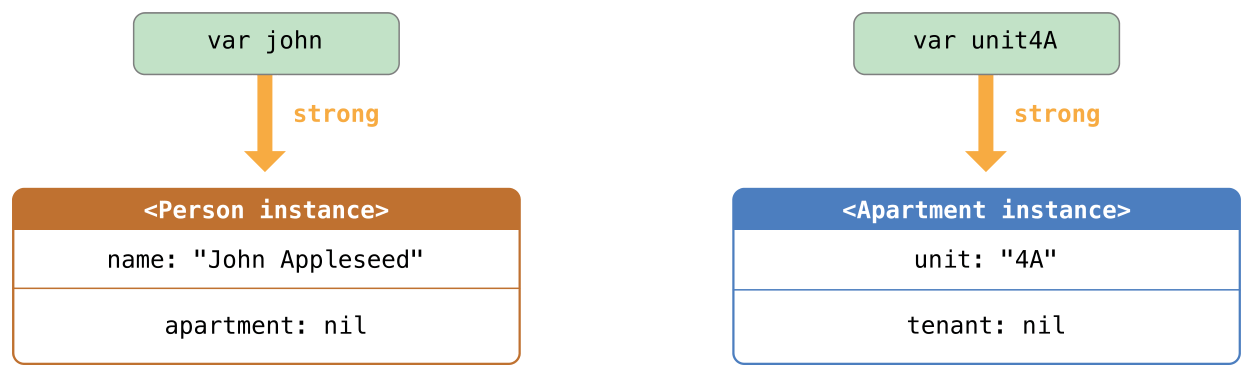
This next code snippet defines two variables of optional type called john and unit4A, which will be set to a specific Apartment and Person instance below. Both of these variables have an initial value of nil, by virtue of being optional:

1. var john: Person?
2. var unit4A: Apartment?

You can now create a specific Person instance and Apartment instance and assign these new instances to the john and unit4A variables:

1. john = Person(name: "John Appleseed")
2. unit4A = Apartment(unit: "4A")

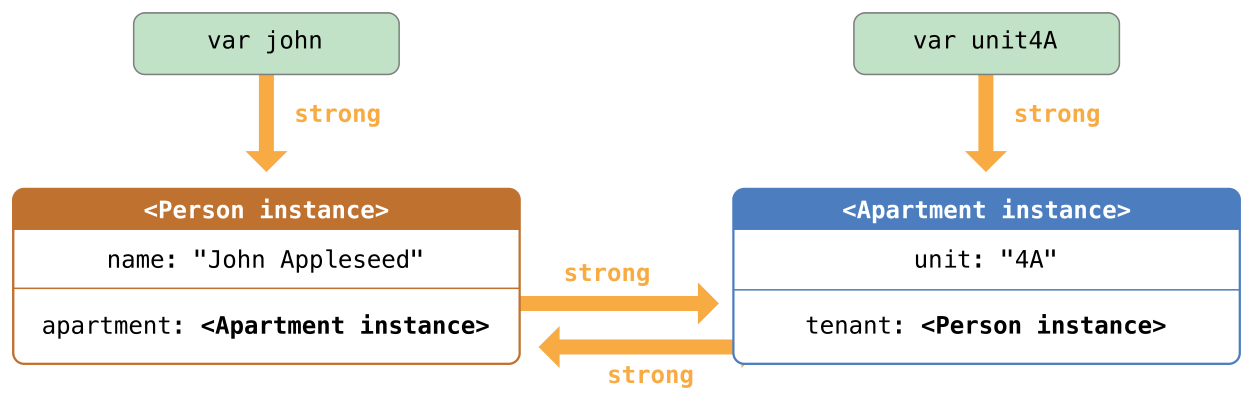
Here’s how the strong references look after creating and assigning these two instances. The john variable now has a strong reference to the new Person instance, and the unit4A variable has a strong reference to the new Apartment instance:



You can now link the two instances together so that the person has an apartment, and the apartment has a tenant. Note that an exclamation mark (!) is used to unwrap and access the instances stored inside the john and unit4A optional variables, so that the properties of those instances can be set:

1. john!.apartment = unit4A
2. unit4A!.tenant = john

Here’s how the strong references look after you link the two instances together:

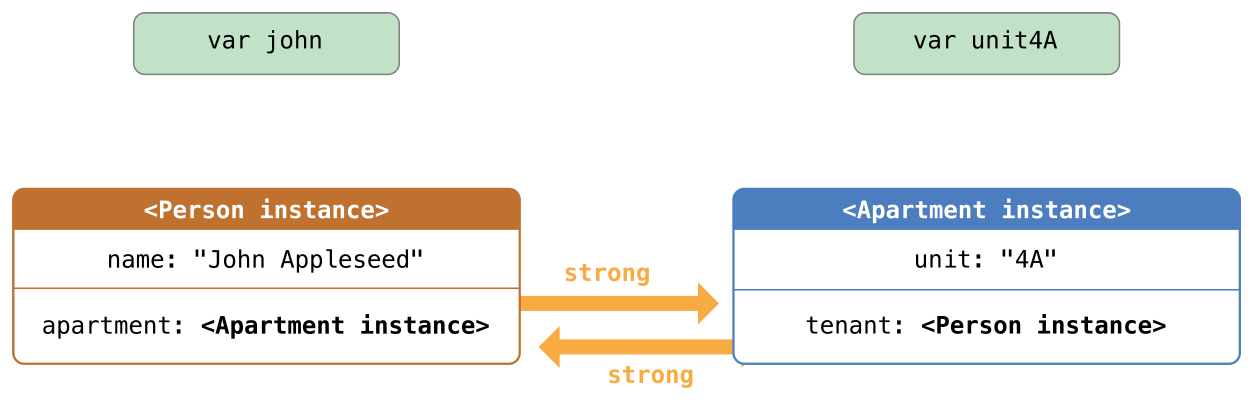


Unfortunately, linking these two instances creates a strong reference cycle between them. The Person instance now has a strong reference to the Apartment instance, and the Apartment instance has a strong reference to the Person instance. Therefore, when you break the strong references held by the john and unit4A variables, the reference counts do not drop to zero, and the instances are not deallocated by ARC:

1. john = nil
2. unit4A = nil

Note that neither deinitializer was called when you set these two variables to nil. The strong reference cycle prevents the Person and Apartment instances from ever being deallocated, causing a memory leak in your app.

Here’s how the strong references look after you set the john and unit4A variables to nil:



The strong references between the Person instance and the Apartment instance remain and cannot be broken.

## Resolving Strong Reference Cycles Between Class Instances

Swift provides two ways to resolve strong reference cycles when you work with properties of class type: weak references and unowned references.

Weak and unowned references enable one instance in a reference cycle to refer to the other instance without keeping a strong hold on it. The instances can then refer to each other without creating a strong reference cycle.

Use a weak reference when the other instance has a shorter lifetime—that is, when the other instance can be deallocated first. In the Apartment example above, it’s appropriate for an apartment to be able to have no tenant at some point in its lifetime, and so a weak reference is an appropriate way to break the reference cycle in this case. In contrast, use an unowned reference when the other instance has the same lifetime or a longer lifetime.

### Weak References

A weak reference is a reference that does not keep a strong hold on the instance it refers to, and so does not stop ARC from disposing of the referenced instance. This behavior prevents the reference from becoming part of a strong reference cycle. You indicate a weak reference by placing the weak keyword before a property or variable declaration.

Because a weak reference does not keep a strong hold on the instance it refers to, it’s possible for that instance to be deallocated while the weak reference is still referring to it. Therefore, ARC automatically sets a weak reference to nil when the instance that it refers to is deallocated. And, because weak references need to allow their value to be changed to nil at runtime, they are always declared as variables, rather than constants, of an optional type.

You can check for the existence of a value in the weak reference, just like any other optional value, and you will never end up with a reference to an invalid instance that no longer exists.

NOTE

Property observers aren’t called when ARC sets a weak reference to nil.

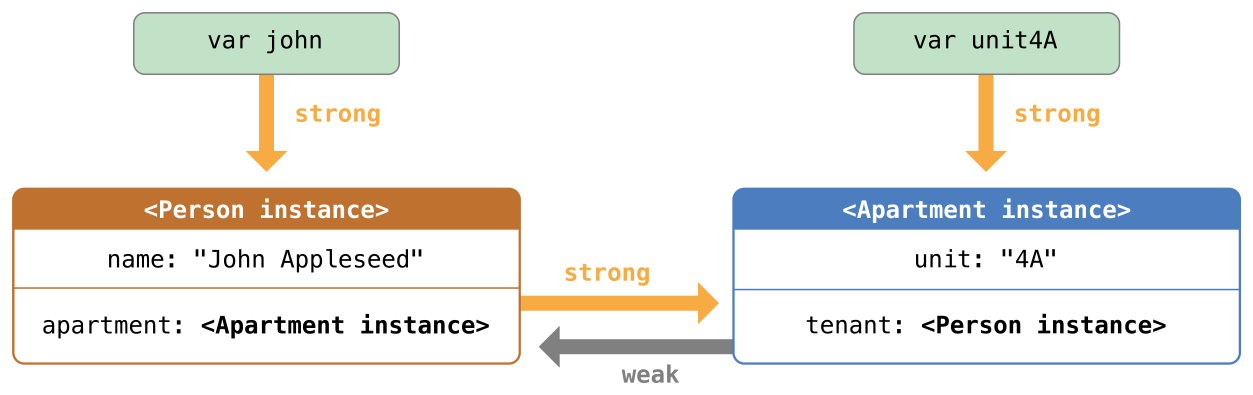
The example below is identical to the Person and Apartment example from above, with one important difference. This time around, the Apartment type’s tenant property is declared as a weak reference:

1. class Person {
2. let name: String
3. init(name: String) { self.name = name }
4. var apartment: Apartment?
5. deinit { print("\(name) is being deinitialized") }
6. }
7. class Apartment {
8. let unit: String
9. init(unit: String) { self.unit = unit }
10. weak var tenant: Person?
11. deinit { print("Apartment \(unit) is being deinitialized") }
12. }

The strong references from the two variables (john and unit4A) and the links between the two instances are created as before:

1. var john: Person?
2. var unit4A: Apartment?
3. john = Person(name: "John Appleseed")
4. unit4A = Apartment(unit: "4A")
5. john!.apartment = unit4A
6. unit4A!.tenant = john

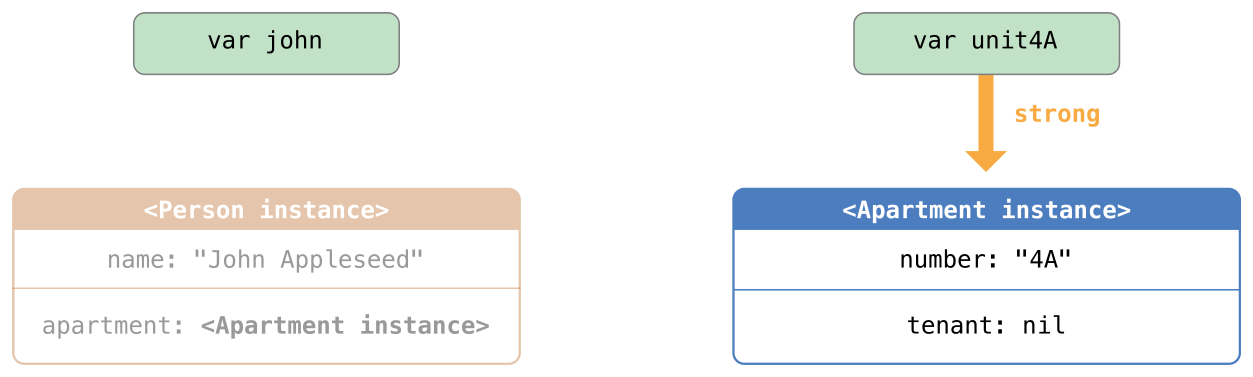
Here’s how the references look now that you’ve linked the two instances together:



The Person instance still has a strong reference to the Apartment instance, but the Apartment instance now has a weak reference to the Person instance. This means that when you break the strong reference held by the john variable by setting it to nil, there are no more strong references to the Person instance:

1. john = nil
2. // Prints "John Appleseed is being deinitialized"

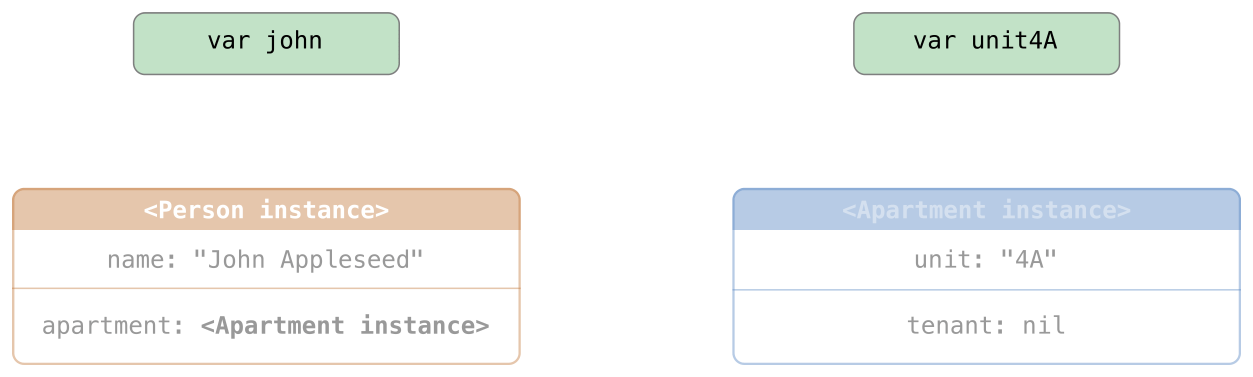
Because there are no more strong references to the Person instance, it’s deallocated and the tenant property is set to nil:



The only remaining strong reference to the Apartment instance is from the unit4A variable. If you break that strong reference, there are no more strong references to the Apartment instance:

1. unit4A = nil
2. // Prints "Apartment 4A is being deinitialized"

Because there are no more strong references to the Apartment instance, it too is deallocated:



NOTE

In systems that use garbage collection, weak pointers are sometimes used to implement a simple caching mechanism because objects with no strong references are deallocated only when memory pressure triggers garbage collection. However, with ARC, values are deallocated as soon as their last strong reference is removed, making weak references unsuitable for such a purpose.

### Unowned References

Like a weak reference, an unowned reference does not keep a strong hold on the instance it refers to. Unlike a weak reference, however, an unowned reference is used when the other instance has the same lifetime or a longer lifetime. You indicate an unowned reference by placing the unowned keyword before a property or variable declaration.

An unowned reference is expected to always have a value. As a result, ARC never sets an unowned reference’s value to nil, which means that unowned references are defined using non-optional types.

IMPORTANT

Use an unowned reference only when you are sure that the reference always refers to an instance that has not been deallocated.

If you try to access the value of an unowned reference after that instance has been deallocated, you’ll get a runtime error.

The following example defines two classes, Customer and CreditCard, which model a bank customer and a possible credit card for that customer. These two classes each store an instance of the other class as a property. This relationship has the potential to create a strong reference cycle.

The relationship between Customer and CreditCard is slightly different from the relationship between Apartment and Person seen in the weak reference example above. In this data model, a customer may or may not have a credit card, but a credit card will always be associated with a customer. A CreditCard instance never outlives the Customer that it refers to. To represent this, the Customer class has an optional card property, but the CreditCard class has an unowned (and non-optional) customer property.

Furthermore, a new CreditCard instance can only be created by passing a number value and a customer instance to a custom CreditCard initializer. This ensures that a CreditCard instance always has a customer instance associated with it when the CreditCard instance is created.

Because a credit card will always have a customer, you define its customer property as an unowned reference, to avoid a strong reference cycle:

1. class Customer {
2. let name: String
3. var card: CreditCard?
4. init(name: String) {
5. self.name = name
6. }
7. deinit { print("\(name) is being deinitialized") }
8. }
9. class CreditCard {
10. let number: UInt64
11. unowned let customer: Customer
12. init(number: UInt64, customer: Customer) {
13. self.number = number
14. self.customer = customer
15. }
16. deinit { print("Card #\(number) is being deinitialized") }
17. }

NOTE

The number property of the CreditCard class is defined with a type of UInt64 rather than Int, to ensure that the number property’s capacity is large enough to store a 16-digit card number on both 32-bit and 64-bit systems.

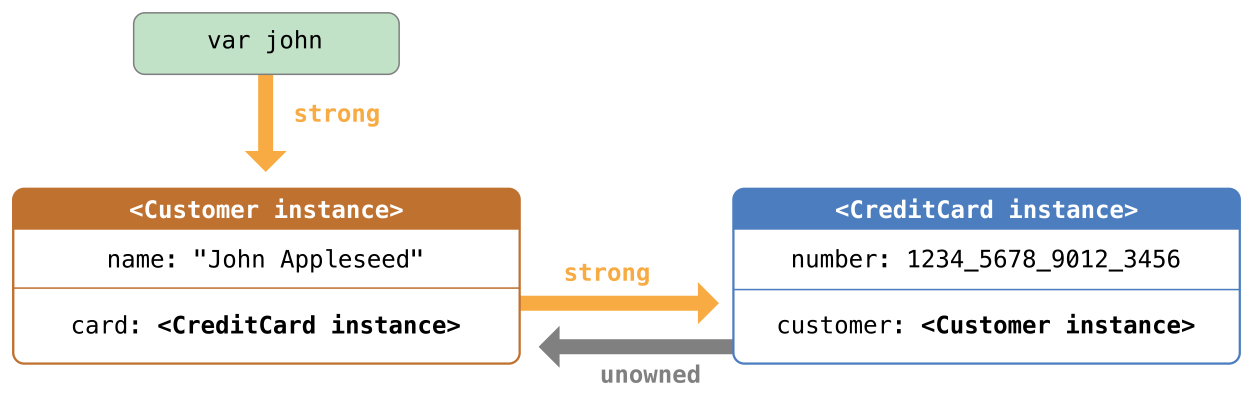
This next code snippet defines an optional Customer variable called john, which will be used to store a reference to a specific customer. This variable has an initial value of nil, by virtue of being optional:

1. var john: Customer?

You can now create a Customer instance, and use it to initialize and assign a new CreditCard instance as that customer’s card property:

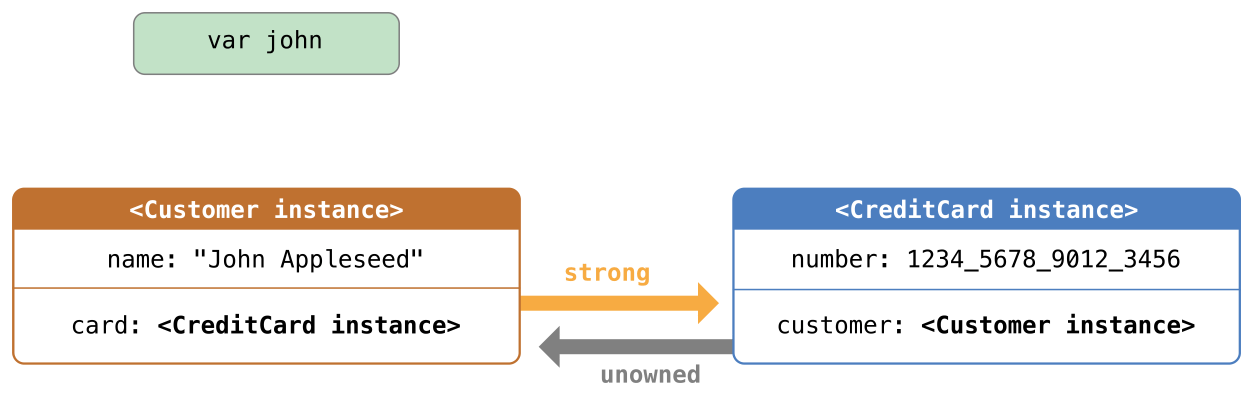
1. john = Customer(name: "John Appleseed")
2. john!.card = CreditCard(number: 1234\_5678\_9012\_3456, customer: john!)

Here’s how the references look, now that you’ve linked the two instances:



The Customer instance now has a strong reference to the CreditCard instance, and the CreditCard instance has an unowned reference to the Customer instance.

Because of the unowned customer reference, when you break the strong reference held by the john variable, there are no more strong references to the Customer instance:



Because there are no more strong references to the Customer instance, it’s deallocated. After this happens, there are no more strong references to the CreditCard instance, and it too is deallocated:

1. john = nil
2. // Prints "John Appleseed is being deinitialized"
3. // Prints "Card #1234567890123456 is being deinitialized"

The final code snippet above shows that the deinitializers for the Customer instance and CreditCard instance both print their “deinitialized” messages after the john variable is set to nil.

NOTE

The examples above show how to use safe unowned references. Swift also provides unsafe unowned references for cases where you need to disable runtime safety checks—for example, for performance reasons. As with all unsafe operations, you take on the responsibility for checking that code for safety.

You indicate an unsafe unowned reference by writing unowned(unsafe). If you try to access an unsafe unowned reference after the instance that it refers to is deallocated, your program will try to access the memory location where the instance used to be, which is an unsafe operation.

### Unowned References and Implicitly Unwrapped Optional Properties

The examples for weak and unowned references above cover two of the more common scenarios in which it’s necessary to break a strong reference cycle.

The Person and Apartment example shows a situation where two properties, both of which are allowed to be nil, have the potential to cause a strong reference cycle. This scenario is best resolved with a weak reference.

The Customer and CreditCard example shows a situation where one property that is allowed to be nil and another property that cannot be nil have the potential to cause a strong reference cycle. This scenario is best resolved with an unowned reference.

However, there is a third scenario, in which both properties should always have a value, and neither property should ever be nil once initialization is complete. In this scenario, it’s useful to combine an unowned property on one class with an implicitly unwrapped optional property on the other class.

This enables both properties to be accessed directly (without optional unwrapping) once initialization is complete, while still avoiding a reference cycle. This section shows you how to set up such a relationship.

The example below defines two classes, Country and City, each of which stores an instance of the other class as a property. In this data model, every country must always have a capital city, and every city must always belong to a country. To represent this, the Country class has a capitalCity property, and the City class has a country property:

1. class Country {
2. let name: String
3. var capitalCity: City!
4. init(name: String, capitalName: String) {
5. self.name = name
6. self.capitalCity = City(name: capitalName, country: self)
7. }
8. }
9. class City {
10. let name: String
11. unowned let country: Country
12. init(name: String, country: Country) {
13. self.name = name
14. self.country = country
15. }
16. }

To set up the interdependency between the two classes, the initializer for City takes a Country instance, and stores this instance in its country property.

The initializer for City is called from within the initializer for Country. However, the initializer for Country cannot pass self to the City initializer until a new Country instance is fully initialized, as described in [Two-Phase Initialization](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID220).

To cope with this requirement, you declare the capitalCity property of Country as an implicitly unwrapped optional property, indicated by the exclamation mark at the end of its type annotation (City!). This means that the capitalCity property has a default value of nil, like any other optional, but can be accessed without the need to unwrap its value as described in [Implicitly Unwrapped Optionals](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID334).

Because capitalCity has a default nil value, a new Country instance is considered fully initialized as soon as the Country instance sets its name property within its initializer. This means that the Country initializer can start to reference and pass around the implicit self property as soon as the name property is set. The Country initializer can therefore pass self as one of the parameters for the City initializer when the Country initializer is setting its own capitalCity property.

All of this means that you can create the Country and City instances in a single statement, without creating a strong reference cycle, and the capitalCity property can be accessed directly, without needing to use an exclamation mark to unwrap its optional value:

1. var country = Country(name: "Canada", capitalName: "Ottawa")
2. print("\(country.name)'s capital city is called \(country.capitalCity.name)")
3. // Prints "Canada's capital city is called Ottawa"

In the example above, the use of an implicitly unwrapped optional means that all of the two-phase class initializer requirements are satisfied. The capitalCity property can be used and accessed like a non-optional value once initialization is complete, while still avoiding a strong reference cycle.

## Strong Reference Cycles for Closures

You saw above how a strong reference cycle can be created when two class instance properties hold a strong reference to each other. You also saw how to use weak and unowned references to break these strong reference cycles.

A strong reference cycle can also occur if you assign a closure to a property of a class instance, and the body of that closure captures the instance. This capture might occur because the closure’s body accesses a property of the instance, such as self.someProperty, or because the closure calls a method on the instance, such as self.someMethod(). In either case, these accesses cause the closure to “capture” self, creating a strong reference cycle.

This strong reference cycle occurs because closures, like classes, are reference types. When you assign a closure to a property, you are assigning a reference to that closure. In essence, it’s the same problem as above—two strong references are keeping each other alive. However, rather than two class instances, this time it’s a class instance and a closure that are keeping each other alive.

Swift provides an elegant solution to this problem, known as a closure capture list. However, before you learn how to break a strong reference cycle with a closure capture list, it’s useful to understand how such a cycle can be caused.

The example below shows how you can create a strong reference cycle when using a closure that references self. This example defines a class called HTMLElement, which provides a simple model for an individual element within an HTML document:

1. class HTMLElement {
2. let name: String
3. let text: String?
4. lazy var asHTML: () -> String = {
5. if let text = self.text {
6. return "<\(self.name)>\(text)</\(self.name)>"
7. } else {
8. return "<\(self.name) />"
9. }
10. }
11. init(name: String, text: String? = nil) {
12. self.name = name
13. self.text = text
14. }
15. deinit {
16. print("\(name) is being deinitialized")
17. }
18. }

The HTMLElement class defines a name property, which indicates the name of the element, such as "h1" for a heading element, "p" for a paragraph element, or "br" for a line break element. HTMLElement also defines an optional text property, which you can set to a string that represents the text to be rendered within that HTML element.

In addition to these two simple properties, the HTMLElement class defines a lazy property called asHTML. This property references a closure that combines name and text into an HTML string fragment. The asHTML property is of type () -> String, or “a function that takes no parameters, and returns a String value”.

By default, the asHTML property is assigned a closure that returns a string representation of an HTML tag. This tag contains the optional text value if it exists, or no text content if text does not exist. For a paragraph element, the closure would return "<p>some text</p>" or "<p />", depending on whether the text property equals "some text" or nil.

The asHTML property is named and used somewhat like an instance method. However, because asHTML is a closure property rather than an instance method, you can replace the default value of the asHTML property with a custom closure, if you want to change the HTML rendering for a particular HTML element.

For example, the asHTML property could be set to a closure that defaults to some text if the text property is nil, in order to prevent the representation from returning an empty HTML tag:

1. let heading = HTMLElement(name: "h1")
2. let defaultText = "some default text"
3. heading.asHTML = {
4. return "<\(heading.name)>\(heading.text ?? defaultText)</\(heading.name)>"
5. }
6. print(heading.asHTML())
7. // Prints "<h1>some default text</h1>"

NOTE

The asHTML property is declared as a lazy property, because it’s only needed if and when the element actually needs to be rendered as a string value for some HTML output target. The fact that asHTML is a lazy property means that you can refer to self within the default closure, because the lazy property will not be accessed until after initialization has been completed and self is known to exist.

The HTMLElement class provides a single initializer, which takes a name argument and (if desired) a text argument to initialize a new element. The class also defines a deinitializer, which prints a message to show when an HTMLElement instance is deallocated.

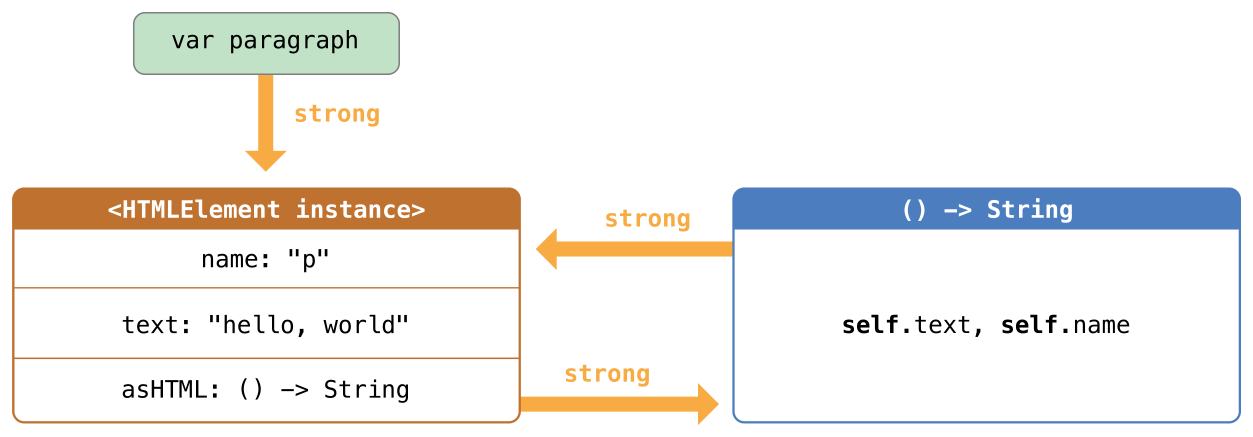
Here’s how you use the HTMLElement class to create and print a new instance:

1. var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
2. print(paragraph!.asHTML())
3. // Prints "<p>hello, world</p>"

NOTE

The paragraph variable above is defined as an optional HTMLElement, so that it can be set to nil below to demonstrate the presence of a strong reference cycle.

Unfortunately, the HTMLElement class, as written above, creates a strong reference cycle between an HTMLElement instance and the closure used for its default asHTML value. Here’s how the cycle looks:



The instance’s asHTML property holds a strong reference to its closure. However, because the closure refers to self within its body (as a way to reference self.name and self.text), the closure captures self, which means that it holds a strong reference back to the HTMLElement instance. A strong reference cycle is created between the two. (For more information about capturing values in a closure, see [Capturing Values](https://docs.swift.org/swift-book/LanguageGuide/Closures.html#ID103).)

NOTE

Even though the closure refers to self multiple times, it only captures one strong reference to the HTMLElement instance.

If you set the paragraph variable to nil and break its strong reference to the HTMLElement instance, neither the HTMLElement instance nor its closure are deallocated, because of the strong reference cycle:

1. paragraph = nil

Note that the message in the HTMLElement deinitializer is not printed, which shows that the HTMLElement instance is not deallocated.

## Resolving Strong Reference Cycles for Closures

You resolve a strong reference cycle between a closure and a class instance by defining a capture list as part of the closure’s definition. A capture list defines the rules to use when capturing one or more reference types within the closure’s body. As with strong reference cycles between two class instances, you declare each captured reference to be a weak or unowned reference rather than a strong reference. The appropriate choice of weak or unowned depends on the relationships between the different parts of your code.

NOTE

Swift requires you to write self.someProperty or self.someMethod() (rather than just someProperty or someMethod()) whenever you refer to a member of self within a closure. This helps you remember that it’s possible to capture self by accident.

### Defining a Capture List

Each item in a capture list is a pairing of the weak or unowned keyword with a reference to a class instance (such as self) or a variable initialized with some value (such as delegate = self.delegate). These pairings are written within a pair of square braces, separated by commas.

Place the capture list before a closure’s parameter list and return type if they are provided:

1. lazy var someClosure = {
2. [unowned self, weak delegate = self.delegate]
3. (index: Int, stringToProcess: String) -> String in
4. // closure body goes here
5. }

If a closure does not specify a parameter list or return type because they can be inferred from context, place the capture list at the very start of the closure, followed by the in keyword:

1. lazy var someClosure = {
2. [unowned self, weak delegate = self.delegate] in
3. // closure body goes here
4. }

### Weak and Unowned References

Define a capture in a closure as an unowned reference when the closure and the instance it captures will always refer to each other, and will always be deallocated at the same time.

Conversely, define a capture as a weak reference when the captured reference may become nil at some point in the future. Weak references are always of an optional type, and automatically become nil when the instance they reference is deallocated. This enables you to check for their existence within the closure’s body.

NOTE

If the captured reference will never become nil, it should always be captured as an unowned reference, rather than a weak reference.

An unowned reference is the appropriate capture method to use to resolve the strong reference cycle in the HTMLElement example from [Strong Reference Cycles for Closures](https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID56) above. Here’s how you write the HTMLElement class to avoid the cycle:

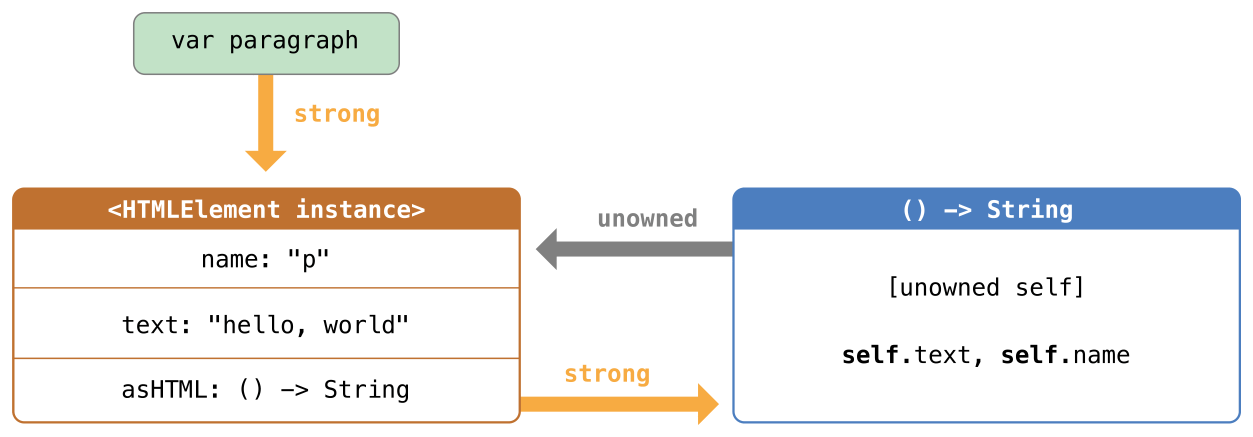
1. class HTMLElement {
2. let name: String
3. let text: String?
4. lazy var asHTML: () -> String = {
5. [unowned self] in
6. if let text = self.text {
7. return "<\(self.name)>\(text)</\(self.name)>"
8. } else {
9. return "<\(self.name) />"
10. }
11. }
12. init(name: String, text: String? = nil) {
13. self.name = name
14. self.text = text
15. }
16. deinit {
17. print("\(name) is being deinitialized")
18. }
19. }

This implementation of HTMLElement is identical to the previous implementation, apart from the addition of a capture list within the asHTML closure. In this case, the capture list is [unowned self], which means “capture self as an unowned reference rather than a strong reference”.

You can create and print an HTMLElement instance as before:

1. var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
2. print(paragraph!.asHTML())
3. // Prints "<p>hello, world</p>"

Here’s how the references look with the capture list in place:



This time, the capture of self by the closure is an unowned reference, and does not keep a strong hold on the HTMLElement instance it has captured. If you set the strong reference from the paragraph variable to nil, the HTMLElement instance is deallocated, as can be seen from the printing of its deinitializer message in the example below:

1. paragraph = nil
2. // Prints "p is being deinitialized"

If you’ve written concurrent or multithreaded code, conflicting access to memory might be a familiar problem. However, the conflicting access discussed here can happen on a single thread and doesn’t involve concurrent or multithreaded code.

If you have conflicting access to memory from within a single thread, Swift guarantees that you’ll get an error at either compile time or runtime. For multithreaded code, use [Thread Sanitizer](https://developer.apple.com/documentation/code_diagnostics/thread_sanitizer) to help detect conflicting access across threads.

### Characteristics of Memory Access

There are three characteristics of memory access to consider in the context of conflicting access: whether the access is a read or a write, the duration of the access, and the location in memory being accessed. Specifically, a conflict occurs if you have two accesses that meet all of the following conditions:

* At least one is a write access.
* They access the same location in memory.
* Their durations overlap.

The difference between a read and write access is usually obvious: a write access changes the location in memory, but a read access doesn’t. The location in memory refers to what is being accessed—for example, a variable, constant, or property. The duration of a memory access is either instantaneous or long-term.

An access is instantaneous if it’s not possible for other code to run after that access starts but before it ends. By their nature, two instantaneous accesses can’t happen at the same time. Most memory access is instantaneous. For example, all the read and write accesses in the code listing below are instantaneous:

1. func oneMore(than number: Int) -> Int {
2. return number + 1
3. }
4. var myNumber = 1
5. myNumber = oneMore(than: myNumber)
6. print(myNumber)
7. // Prints "2"

However, there are several ways to access memory, called long-term accesses, that span the execution of other code. The difference between instantaneous access and long-term access is that it’s possible for other code to run after a long-term access starts but before it ends, which is called overlap. A long-term access can overlap with other long-term accesses and instantaneous accesses.

Overlapping accesses appear primarily in code that uses in-out parameters in functions and methods or mutating methods of a structure. The specific kinds of Swift code that use long-term accesses are discussed in the sections below.

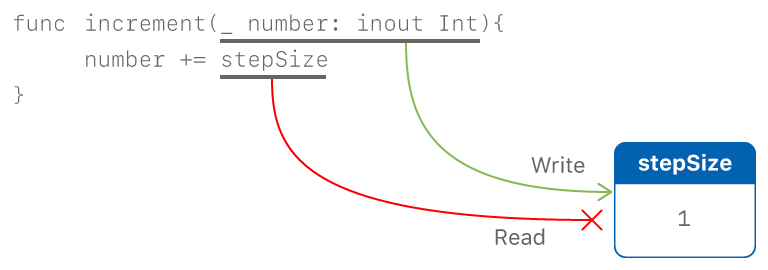
## Conflicting Access to In-Out Parameters

A function has long-term write access to all of its in-out parameters. The write access for an in-out parameter starts after all of the non-in-out parameters have been evaluated and lasts for the entire duration of that function call. If there are multiple in-out parameters, the write accesses start in the same order as the parameters appear.

One consequence of this long-term write access is that you can’t access the original variable that was passed as in-out, even if scoping rules and access control would otherwise permit it—any access to the original creates a conflict. For example:

1. var stepSize = 1
2. func increment(\_ number: inout Int) {
3. number += stepSize
4. }
5. increment(&stepSize)
6. // Error: conflicting accesses to stepSize

In the code above, stepSize is a global variable, and it is normally accessible from within increment(\_:). However, the read access to stepSize overlaps with the write access to number. As shown in the figure below, both number and stepSize refer to the same location in memory. The read and write accesses refer to the same memory and they overlap, producing a conflict.



One way to solve this conflict is to make an explicit copy of stepSize:

1. // Make an explicit copy.
2. var copyOfStepSize = stepSize
3. increment(&copyOfStepSize)
4. // Update the original.
5. stepSize = copyOfStepSize
6. // stepSize is now 2

When you make a copy of stepSize before calling increment(\_:), it’s clear that the value of copyOfStepSize is incremented by the current step size. The read access ends before the write access starts, so there isn’t a conflict.

Another consequence of long-term write access to in-out parameters is that passing a single variable as the argument for multiple in-out parameters of the same function produces a conflict. For example:

1. func balance(\_ x: inout Int, \_ y: inout Int) {
2. let sum = x + y
3. x = sum / 2
4. y = sum - x
5. }
6. var playerOneScore = 42
7. var playerTwoScore = 30
8. balance(&playerOneScore, &playerTwoScore) // OK
9. balance(&playerOneScore, &playerOneScore)
10. // Error: conflicting accesses to playerOneScore

The balance(\_:\_:) function above modifies its two parameters to divide the total value evenly between them. Calling it with playerOneScore and playerTwoScore as arguments doesn’t produce a conflict—there are two write accesses that overlap in time, but they access different locations in memory. In contrast, passing playerOneScore as the value for both parameters produces a conflict because it tries to perform two write accesses to the same location in memory at the same time.

NOTE

Because operators are functions, they can also have long-term accesses to their in-out parameters. For example, if balance(\_:\_:) was an operator function named <^>, writing playerOneScore <^> playerOneScore would result in the same conflict as balance(&playerOneScore, &playerOneScore).

## Conflicting Access to self in Methods

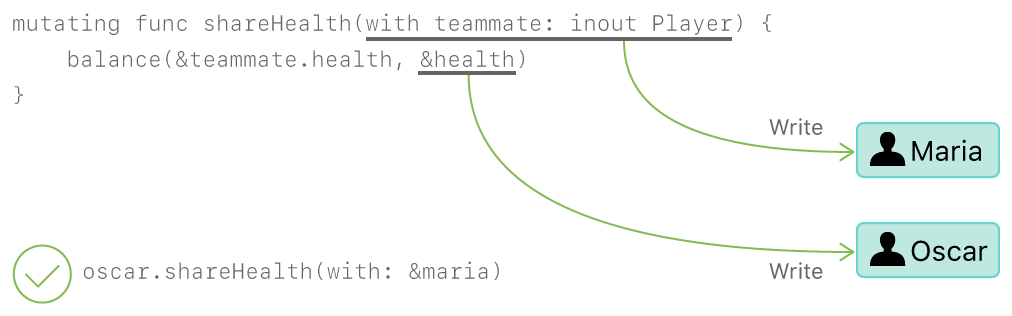
A mutating method on a structure has write access to self for the duration of the method call. For example, consider a game where each player has a health amount, which decreases when taking damage, and an energy amount, which decreases when using special abilities.

1. struct Player {
2. var name: String
3. var health: Int
4. var energy: Int
5. static let maxHealth = 10
6. mutating func restoreHealth() {
7. health = Player.maxHealth
8. }
9. }

In the restoreHealth() method above, a write access to self starts at the beginning of the method and lasts until the method returns. In this case, there’s no other code inside restoreHealth() that could have an overlapping access to the properties of a Player instance. The shareHealth(with:) method below takes another Player instance as an in-out parameter, creating the possibility of overlapping accesses.

1. extension Player {
2. mutating func shareHealth(with teammate: inout Player) {
3. balance(&teammate.health, &health)
4. }
5. }
6. var oscar = Player(name: "Oscar", health: 10, energy: 10)
7. var maria = Player(name: "Maria", health: 5, energy: 10)
8. oscar.shareHealth(with: &maria) // OK

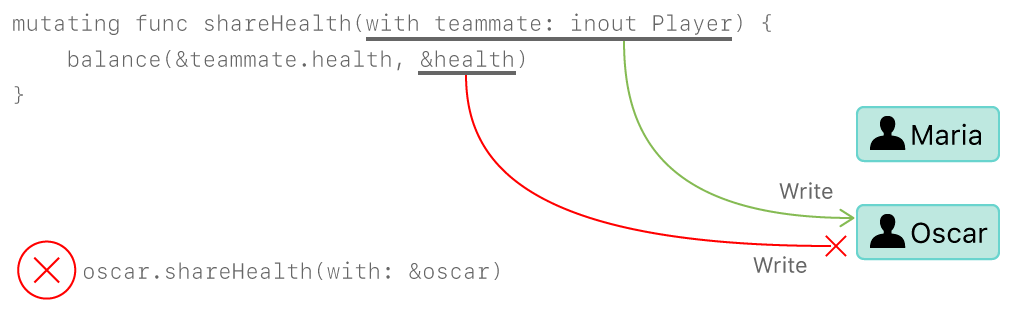
In the example above, calling the shareHealth(with:) method for Oscar’s player to share health with Maria’s player doesn’t cause a conflict. There’s a write access to oscar during the method call because oscar is the value of self in a mutating method, and there’s a write access to maria for the same duration because maria was passed as an in-out parameter. As shown in the figure below, they access different locations in memory. Even though the two write accesses overlap in time, they don’t conflict.



However, if you pass oscar as the argument to shareHealth(with:), there’s a conflict:

1. oscar.shareHealth(with: &oscar)
2. // Error: conflicting accesses to oscar

The mutating method needs write access to self for the duration of the method, and the in-out parameter needs write access to teammate for the same duration. Within the method, both self and teammate refer to the same location in memory—as shown in the figure below. The two write accesses refer to the same memory and they overlap, producing a conflict.



## Conflicting Access to Properties

Types like structures, tuples, and enumerations are made up of individual constituent values, such as the properties of a structure or the elements of a tuple. Because these are value types, mutating any piece of the value mutates the whole value, meaning read or write access to one of the properties requires read or write access to the whole value. For example, overlapping write accesses to the elements of a tuple produces a conflict:

1. var playerInformation = (health: 10, energy: 20)
2. balance(&playerInformation.health, &playerInformation.energy)
3. // Error: conflicting access to properties of playerInformation

In the example above, calling balance(\_:\_:) on the elements of a tuple produces a conflict because there are overlapping write accesses to playerInformation. Both playerInformation.health and playerInformation.energy are passed as in-out parameters, which means balance(\_:\_:) needs write access to them for the duration of the function call. In both cases, a write access to the tuple element requires a write access to the entire tuple. This means there are two write accesses to playerInformation with durations that overlap, causing a conflict.

The code below shows that the same error appears for overlapping write accesses to the properties of a structure that’s stored in a global variable.

1. var holly = Player(name: "Holly", health: 10, energy: 10)
2. balance(&holly.health, &holly.energy) // Error

In practice, most access to the properties of a structure can overlap safely. For example, if the variable holly in the example above is changed to a local variable instead of a global variable, the compiler can prove that overlapping access to stored properties of the structure is safe:

1. func someFunction() {
2. var oscar = Player(name: "Oscar", health: 10, energy: 10)
3. balance(&oscar.health, &oscar.energy) // OK
4. }

In the example above, Oscar’s health and energy are passed as the two in-out parameters to balance(\_:\_:). The compiler can prove that memory safety is preserved because the two stored properties don’t interact in any way.

The restriction against overlapping access to properties of a structure isn’t always necessary to preserve memory safety. Memory safety is the desired guarantee, but exclusive access is a stricter requirement than memory safety—which means some code preserves memory safety, even though it violates exclusive access to memory. Swift allows this memory-safe code if the compiler can prove that the nonexclusive access to memory is still safe. Specifically, it can prove that overlapping access to properties of a structure is safe if the following conditions apply:

* You’re accessing only stored properties of an instance, not computed properties or class properties.
* The structure is the value of a local variable, not a global variable.
* The structure is either not captured by any closures, or it’s captured only by nonescaping closures.

If the compiler can’t prove the access is safe, it doesn’t allow the access.

# Access Control

Access control restricts access to parts of your code from code in other source files and modules. This feature enables you to hide the implementation details of your code, and to specify a preferred interface through which that code can be accessed and used.

You can assign specific access levels to individual types (classes, structures, and enumerations), as well as to properties, methods, initializers, and subscripts belonging to those types. Protocols can be restricted to a certain context, as can global constants, variables, and functions.

In addition to offering various levels of access control, Swift reduces the need to specify explicit access control levels by providing default access levels for typical scenarios. Indeed, if you are writing a single-target app, you may not need to specify explicit access control levels at all.

NOTE

The various aspects of your code that can have access control applied to them (properties, types, functions, and so on) are referred to as “entities” in the sections below, for brevity.

## Modules and Source Files

Swift’s access control model is based on the concept of modules and source files.

A module is a single unit of code distribution—a framework or application that is built and shipped as a single unit and that can be imported by another module with Swift’s import keyword.

Each build target (such as an app bundle or framework) in Xcode is treated as a separate module in Swift. If you group together aspects of your app’s code as a stand-alone framework—perhaps to encapsulate and reuse that code across multiple applications—then everything you define within that framework will be part of a separate module when it’s imported and used within an app, or when it’s used within another framework.

A source file is a single Swift source code file within a module (in effect, a single file within an app or framework). Although it’s common to define individual types in separate source files, a single source file can contain definitions for multiple types, functions, and so on.

## Access Levels

Swift provides five different access levels for entities within your code. These access levels are relative to the source file in which an entity is defined, and also relative to the module that source file belongs to.

* Open access and public access enable entities to be used within any source file from their defining module, and also in a source file from another module that imports the defining module. You typically use open or public access when specifying the public interface to a framework. The difference between open and public access is described below.
* Internal access enables entities to be used within any source file from their defining module, but not in any source file outside of that module. You typically use internal access when defining an app’s or a framework’s internal structure.
* File-private access restricts the use of an entity to its own defining source file. Use file-private access to hide the implementation details of a specific piece of functionality when those details are used within an entire file.
* Private access restricts the use of an entity to the enclosing declaration, and to extensions of that declaration that are in the same file. Use private access to hide the implementation details of a specific piece of functionality when those details are used only within a single declaration.

Open access is the highest (least restrictive) access level and private access is the lowest (most restrictive) access level.

Open access applies only to classes and class members, and it differs from public access by allowing code outside the module to subclass and override, as discussed below in [Subclassing](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html#ID16). Marking a class as open explicitly indicates that you’ve considered the impact of code from other modules using that class as a superclass, and that you’ve designed your class’s code accordingly.

### Guiding Principle of Access Levels

Access levels in Swift follow an overall guiding principle: No entity can be defined in terms of another entity that has a lower (more restrictive) access level.

For example:

* A public variable can’t be defined as having an internal, file-private, or private type, because the type might not be available everywhere that the public variable is used.
* A function can’t have a higher access level than its parameter types and return type, because the function could be used in situations where its constituent types are unavailable to the surrounding code.

The specific implications of this guiding principle for different aspects of the language are covered in detail below.

### Default Access Levels

All entities in your code (with a few specific exceptions, as described later in this chapter) have a default access level of internal if you don’t specify an explicit access level yourself. As a result, in many cases you don’t need to specify an explicit access level in your code.

### Access Levels for Single-Target Apps

When you write a simple single-target app, the code in your app is typically self-contained within the app and doesn’t need to be made available outside of the app’s module. The default access level of internal already matches this requirement. Therefore, you don’t need to specify a custom access level. You may, however, want to mark some parts of your code as file private or private in order to hide their implementation details from other code within the app’s module.

### Access Levels for Frameworks

When you develop a framework, mark the public-facing interface to that framework as open or public so that it can be viewed and accessed by other modules, such as an app that imports the framework. This public-facing interface is the application programming interface (or API) for the framework.

NOTE

Any internal implementation details of your framework can still use the default access level of internal, or can be marked as private or file private if you want to hide them from other parts of the framework’s internal code. You need to mark an entity as open or public only if you want it to become part of your framework’s API.

### Access Levels for Unit Test Targets

When you write an app with a unit test target, the code in your app needs to be made available to that module in order to be tested. By default, only entities marked as open or public are accessible to other modules. However, a unit test target can access any internal entity, if you mark the import declaration for a product module with the @testable attribute and compile that product module with testing enabled.

## Access Control Syntax

Define the access level for an entity by placing one of the open, public, internal, fileprivate, or private modifiers at the beginning of the entity’s declaration.

1. public class SomePublicClass {}
2. internal class SomeInternalClass {}
3. fileprivate class SomeFilePrivateClass {}
4. private class SomePrivateClass {}
5. public var somePublicVariable = 0
6. internal let someInternalConstant = 0
7. fileprivate func someFilePrivateFunction() {}
8. private func somePrivateFunction() {}

Unless otherwise specified, the default access level is internal, as described in [Default Access Levels](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html#ID7). This means that SomeInternalClass and someInternalConstant can be written without an explicit access-level modifier, and will still have an access level of internal:

1. class SomeInternalClass {} // implicitly internal
2. let someInternalConstant = 0 // implicitly internal

## Custom Types

If you want to specify an explicit access level for a custom type, do so at the point that you define the type. The new type can then be used wherever its access level permits. For example, if you define a file-private class, that class can only be used as the type of a property, or as a function parameter or return type, in the source file in which the file-private class is defined.

The access control level of a type also affects the default access level of that type’s members (its properties, methods, initializers, and subscripts). If you define a type’s access level as private or file private, the default access level of its members will also be private or file private. If you define a type’s access level as internal or public (or use the default access level of internal without specifying an access level explicitly), the default access level of the type’s members will be internal.

IMPORTANT

A public type defaults to having internal members, not public members. If you want a type member to be public, you must explicitly mark it as such. This requirement ensures that the public-facing API for a type is something you opt in to publishing, and avoids presenting the internal workings of a type as public API by mistake.

1. public class SomePublicClass { // explicitly public class
2. public var somePublicProperty = 0 // explicitly public class member
3. var someInternalProperty = 0 // implicitly internal class member
4. fileprivate func someFilePrivateMethod() {} // explicitly file-private class member
5. private func somePrivateMethod() {} // explicitly private class member
6. }
7. class SomeInternalClass { // implicitly internal class
8. var someInternalProperty = 0 // implicitly internal class member
9. fileprivate func someFilePrivateMethod() {} // explicitly file-private class member
10. private func somePrivateMethod() {} // explicitly private class member
11. }
12. fileprivate class SomeFilePrivateClass { // explicitly file-private class
13. func someFilePrivateMethod() {} // implicitly file-private class member
14. private func somePrivateMethod() {} // explicitly private class member
15. }
16. private class SomePrivateClass { // explicitly private class
17. func somePrivateMethod() {} // implicitly private class member
18. }

### Tuple Types

The access level for a tuple type is the most restrictive access level of all types used in that tuple. For example, if you compose a tuple from two different types, one with internal access and one with private access, the access level for that compound tuple type will be private.

NOTE

Tuple types don’t have a standalone definition in the way that classes, structures, enumerations, and functions do. A tuple type’s access level is determined automatically from the types that make up the tuple type, and can’t be specified explicitly.

### Function Types

The access level for a function type is calculated as the most restrictive access level of the function’s parameter types and return type. You must specify the access level explicitly as part of the function’s definition if the function’s calculated access level doesn’t match the contextual default.

The example below defines a global function called someFunction(), without providing a specific access-level modifier for the function itself. You might expect this function to have the default access level of “internal”, but this isn’t the case. In fact, someFunction() won’t compile as written below:

1. func someFunction() -> (SomeInternalClass, SomePrivateClass) {
2. // function implementation goes here
3. }

The function’s return type is a tuple type composed from two of the custom classes defined above in [Custom Types](https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html#ID11). One of these classes is defined as internal, and the other is defined as private. Therefore, the overall access level of the compound tuple type is private (the minimum access level of the tuple’s constituent types).

Because the function’s return type is private, you must mark the function’s overall access level with the private modifier for the function declaration to be valid:

1. private func someFunction() -> (SomeInternalClass, SomePrivateClass) {
2. // function implementation goes here
3. }

It’s not valid to mark the definition of someFunction() with the public or internal modifiers, or to use the default setting of internal, because public or internal users of the function might not have appropriate access to the private class used in the function’s return type.

### Enumeration Types

The individual cases of an enumeration automatically receive the same access level as the enumeration they belong to. You can’t specify a different access level for individual enumeration cases.

In the example below, the CompassPoint enumeration has an explicit access level of public. The enumeration cases north, south, east, and west therefore also have an access level of public:

1. public enum CompassPoint {
2. case north
3. case south
4. case east
5. case west
6. }

#### Raw Values and Associated Values

The types used for any raw values or associated values in an enumeration definition must have an access level at least as high as the enumeration’s access level. For example, you can’t use a private type as the raw-value type of an enumeration with an internal access level.

### Nested Types

The access level of a nested type is the same as its containing type, unless the containing type is public. Nested types defined within a public type have an automatic access level of internal. If you want a nested type within a public type to be publicly available, you must explicitly declare the nested type as public.

## Subclassing

You can subclass any class that can be accessed in the current access context and that’s defined in the same module as the subclass. You can also subclass any open class that’s defined in a different module. A subclass can’t have a higher access level than its superclass—for example, you can’t write a public subclass of an internal superclass.

In addition, for classes that are defined in the same module, you can override any class member (method, property, initializer, or subscript) that’s visible in a certain access context. For classes that are defined in another module, you can override any open class member.

An override can make an inherited class member more accessible than its superclass version. In the example below, class A is a public class with a file-private method called someMethod(). Class B is a subclass of A, with a reduced access level of “internal”. Nonetheless, class B provides an override of someMethod() with an access level of “internal”, which is higher than the original implementation of someMethod():

1. public class A {
2. fileprivate func someMethod() {}
3. }
4. internal class B: A {
5. override internal func someMethod() {}
6. }

It’s even valid for a subclass member to call a superclass member that has lower access permissions than the subclass member, as long as the call to the superclass’s member takes place within an allowed access level context (that is, within the same source file as the superclass for a file-private member call, or within the same module as the superclass for an internal member call):

1. public class A {
2. fileprivate func someMethod() {}
3. }
4. internal class B: A {
5. override internal func someMethod() {
6. super.someMethod()
7. }
8. }

Because superclass A and subclass B are defined in the same source file, it’s valid for the B implementation of someMethod() to call super.someMethod().

## Constants, Variables, Properties, and Subscripts

A constant, variable, or property can’t be more public than its type. It’s not valid to write a public property with a private type, for example. Similarly, a subscript can’t be more public than either its index type or return type.

If a constant, variable, property, or subscript makes use of a private type, the constant, variable, property, or subscript must also be marked as private:

1. private var privateInstance = SomePrivateClass()

### Getters and Setters

Getters and setters for constants, variables, properties, and subscripts automatically receive the same access level as the constant, variable, property, or subscript they belong to.

You can give a setter a lower access level than its corresponding getter, to restrict the read-write scope of that variable, property, or subscript. You assign a lower access level by writing fileprivate(set), private(set), or internal(set) before the var or subscript introducer.

NOTE

This rule applies to stored properties as well as computed properties. Even though you don’t write an explicit getter and setter for a stored property, Swift still synthesizes an implicit getter and setter for you to provide access to the stored property’s backing storage. Use fileprivate(set), private(set), and internal(set) to change the access level of this synthesized setter in exactly the same way as for an explicit setter in a computed property.

The example below defines a structure called TrackedString, which keeps track of the number of times a string property is modified:

1. struct TrackedString {
2. private(set) var numberOfEdits = 0
3. var value: String = "" {
4. didSet {
5. numberOfEdits += 1
6. }
7. }
8. }

The TrackedString structure defines a stored string property called value, with an initial value of "" (an empty string). The structure also defines a stored integer property called numberOfEdits, which is used to track the number of times that value is modified. This modification tracking is implemented with a didSet property observer on the value property, which increments numberOfEdits every time the value property is set to a new value.

The TrackedString structure and the value property don’t provide an explicit access-level modifier, and so they both receive the default access level of internal. However, the access level for the numberOfEdits property is marked with a private(set) modifier to indicate that the property’s getter still has the default access level of internal, but the property is settable only from within code that’s part of the TrackedString structure. This enables TrackedString to modify the numberOfEdits property internally, but to present the property as a read-only property when it’s used outside the structure’s definition.

If you create a TrackedString instance and modify its string value a few times, you can see the numberOfEdits property value update to match the number of modifications:

1. var stringToEdit = TrackedString()
2. stringToEdit.value = "This string will be tracked."
3. stringToEdit.value += " This edit will increment numberOfEdits."
4. stringToEdit.value += " So will this one."
5. print("The number of edits is \(stringToEdit.numberOfEdits)")
6. // Prints "The number of edits is 3"

Although you can query the current value of the numberOfEdits property from within another source file, you can’t modify the property from another source file. This restriction protects the implementation details of the TrackedString edit-tracking functionality, while still providing convenient access to an aspect of that functionality.

Note that you can assign an explicit access level for both a getter and a setter if required. The example below shows a version of the TrackedString structure in which the structure is defined with an explicit access level of public. The structure’s members (including the numberOfEdits property) therefore have an internal access level by default. You can make the structure’s numberOfEdits property getter public, and its property setter private, by combining the public and private(set) access-level modifiers:

1. public struct TrackedString {
2. public private(set) var numberOfEdits = 0
3. public var value: String = "" {
4. didSet {
5. numberOfEdits += 1
6. }
7. }
8. public init() {}
9. }

## Initializers

Custom initializers can be assigned an access level less than or equal to the type that they initialize. The only exception is for required initializers (as defined in [Required Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID231)). A required initializer must have the same access level as the class it belongs to.

As with function and method parameters, the types of an initializer’s parameters can’t be more private than the initializer’s own access level.

### Default Initializers

As described in [Default Initializers](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html#ID213), Swift automatically provides a default initializer without any arguments for any structure or base class that provides default values for all of its properties and doesn’t provide at least one initializer itself.

A default initializer has the same access level as the type it initializes, unless that type is defined as public. For a type that is defined as public, the default initializer is considered internal. If you want a public type to be initializable with a no-argument initializer when used in another module, you must explicitly provide a public no-argument initializer yourself as part of the type’s definition.

### Default Memberwise Initializers for Structure Types

The default memberwise initializer for a structure type is considered private if any of the structure’s stored properties are private. Likewise, if any of the structure’s stored properties are file private, the initializer is file private. Otherwise, the initializer has an access level of internal.

As with the default initializer above, if you want a public structure type to be initializable with a memberwise initializer when used in another module, you must provide a public memberwise initializer yourself as part of the type’s definition.

## Protocols

If you want to assign an explicit access level to a protocol type, do so at the point that you define the protocol. This enables you to create protocols that can only be adopted within a certain access context.

The access level of each requirement within a protocol definition is automatically set to the same access level as the protocol. You can’t set a protocol requirement to a different access level than the protocol it supports. This ensures that all of the protocol’s requirements will be visible on any type that adopts the protocol.

NOTE

If you define a public protocol, the protocol’s requirements require a public access level for those requirements when they’re implemented. This behavior is different from other types, where a public type definition implies an access level of internal for the type’s members.

### Protocol Inheritance

If you define a new protocol that inherits from an existing protocol, the new protocol can have at most the same access level as the protocol it inherits from. For example, you can’t write a public protocol that inherits from an internal protocol.

### Protocol Conformance

A type can conform to a protocol with a lower access level than the type itself. For example, you can define a public type that can be used in other modules, but whose conformance to an internal protocol can only be used within the internal protocol’s defining module.

The context in which a type conforms to a particular protocol is the minimum of the type’s access level and the protocol’s access level. For example, if a type is public, but a protocol it conforms to is internal, the type’s conformance to that protocol is also internal.

When you write or extend a type to conform to a protocol, you must ensure that the type’s implementation of each protocol requirement has at least the same access level as the type’s conformance to that protocol. For example, if a public type conforms to an internal protocol, the type’s implementation of each protocol requirement must be at least internal.

NOTE

In Swift, as in Objective-C, protocol conformance is global—it isn’t possible for a type to conform to a protocol in two different ways within the same program.

## Extensions

You can extend a class, structure, or enumeration in any access context in which the class, structure, or enumeration is available. Any type members added in an extension have the same default access level as type members declared in the original type being extended. If you extend a public or internal type, any new type members you add have a default access level of internal. If you extend a file-private type, any new type members you add have a default access level of file private. If you extend a private type, any new type members you add have a default access level of private.

Alternatively, you can mark an extension with an explicit access-level modifier (for example, private) to set a new default access level for all members defined within the extension. This new default can still be overridden within the extension for individual type members.

You can’t provide an explicit access-level modifier for an extension if you’re using that extension to add protocol conformance. Instead, the protocol’s own access level is used to provide the default access level for each protocol requirement implementation within the extension.

### Private Members in Extensions

Extensions that are in the same file as the class, structure, or enumeration that they extend behave as if the code in the extension had been written as part of the original type’s declaration. As a result, you can:

* Declare a private member in the original declaration, and access that member from extensions in the same file.
* Declare a private member in one extension, and access that member from another extension in the same file.
* Declare a private member in an extension, and access that member from the original declaration in the same file.

This behavior means you can use extensions in the same way to organize your code, whether or not your types have private entities. For example, given the following simple protocol:

1. protocol SomeProtocol {
2. func doSomething()
3. }

You can use an extension to add protocol conformance, like this:

1. struct SomeStruct {
2. private var privateVariable = 12
3. }
4. extension SomeStruct: SomeProtocol {
5. func doSomething() {
6. print(privateVariable)
7. }
8. }

## Generics

The access level for a generic type or generic function is the minimum of the access level of the generic type or function itself and the access level of any type constraints on its type parameters.

## Type Aliases

Any type aliases you define are treated as distinct types for the purposes of access control. A type alias can have an access level less than or equal to the access level of the type it aliases. For example, a private type alias can alias a private, file-private, internal, public, or open type, but a public type alias can’t alias an internal, file-private, or private type.

NOTE

This rule also applies to type aliases for associated types used to satisfy protocol conformances.

### Bitwise NOT Operator

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

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

1. let initialBits: UInt8 = 0b00001111
2. let invertedBits = ~initialBits // equals 11110000

### Bitwise AND Operator

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

1. let firstSixBits: UInt8 = 0b11111100
2. let lastSixBits: UInt8 = 0b00111111
3. let middleFourBits = firstSixBits & lastSixBits // equals 00111100

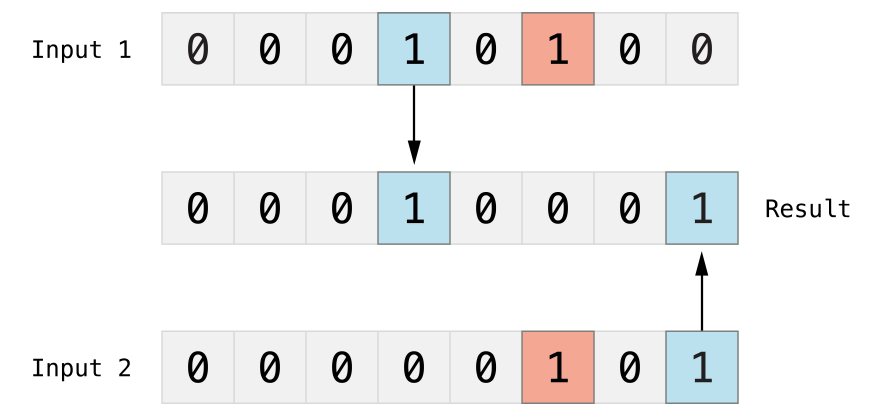
### Bitwise OR Operator

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

1. let someBits: UInt8 = 0b10110010
2. let moreBits: UInt8 = 0b01011110
3. let combinedbits = someBits | moreBits // equals 11111110

### Bitwise XOR Operator

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



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

1. let firstBits: UInt8 = 0b00010100
2. let otherBits: UInt8 = 0b00000101
3. let outputBits = firstBits ^ otherBits // equals 00010001

### Bitwise Left and Right Shift Operators

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

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

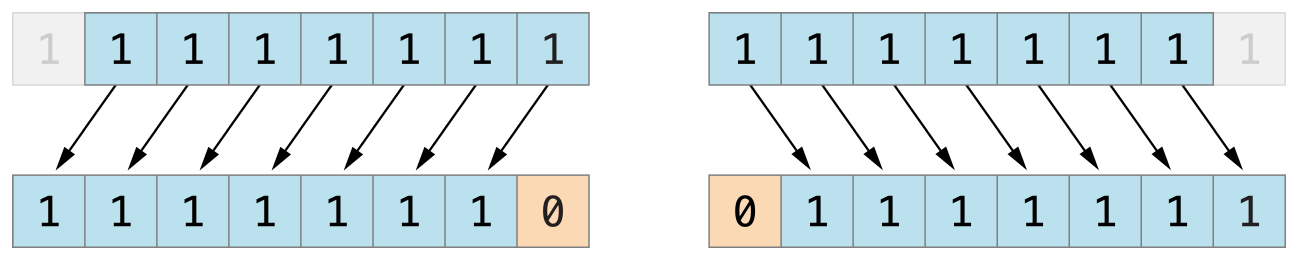
#### Shifting Behavior for Unsigned Integers

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

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

This approach is known as a logical shift.

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



Here’s how bit shifting looks in Swift code:

1. let shiftBits: UInt8 = 4 // 00000100 in binary
2. shiftBits << 1 // 00001000
3. shiftBits << 2 // 00010000
4. shiftBits << 5 // 10000000
5. shiftBits << 6 // 00000000
6. shiftBits >> 2 // 00000001

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

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

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

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

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

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

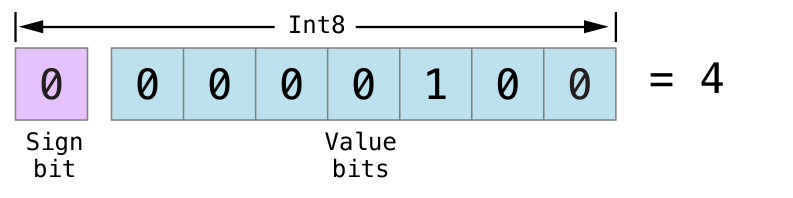
Finally, the blue component is obtained by performing a bitwise AND between the numbers 0xCC6699 and 0x0000FF, which gives an output value of 0x000099. There’s no need to shift this to the right, as 0x000099 already equals 0x99, which has a decimal value of 153.

#### Shifting Behavior for Signed Integers

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

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

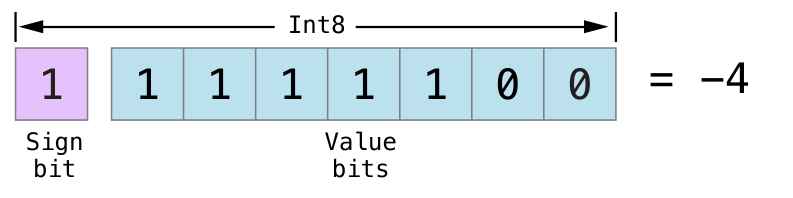
The remaining bits (known as the value bits) store the actual value. Positive numbers are stored in exactly the same way as for unsigned integers, counting upwards from 0. Here’s how the bits inside an Int8 look for the number 4:



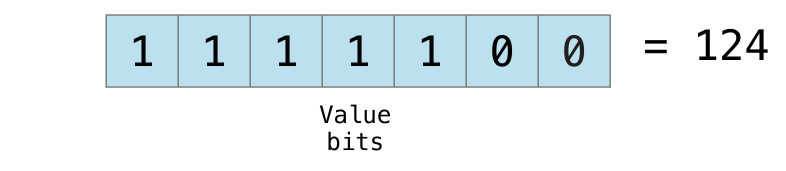
The sign bit is 0 (meaning “positive”), and the seven value bits are just the number 4, written in binary notation.

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

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

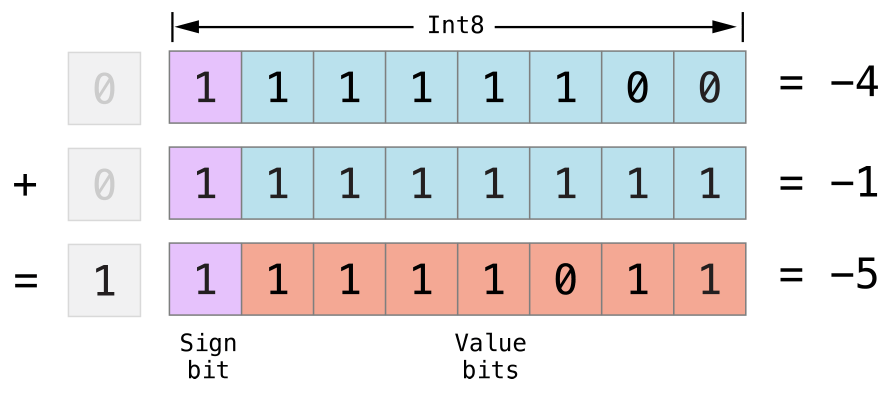


This time, the sign bit is 1 (meaning “negative”), and the seven value bits have a binary value of 124 (which is 128 - 4):

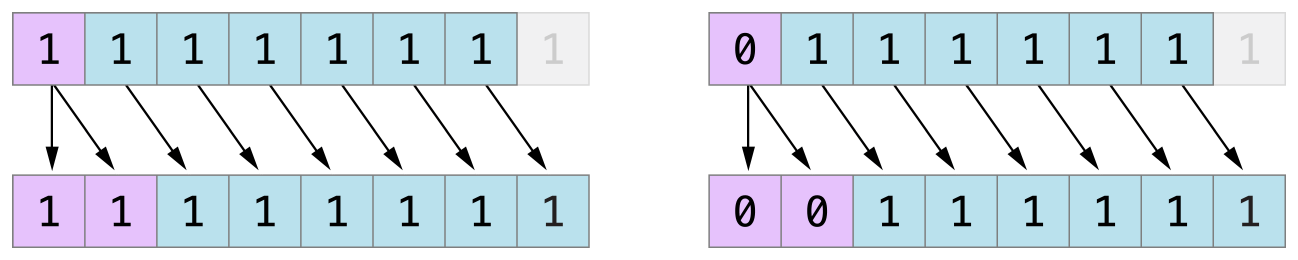


This encoding for negative numbers is known as a two’s complement representation. It may seem an unusual way to represent negative numbers, but it has several advantages.

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



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



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

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

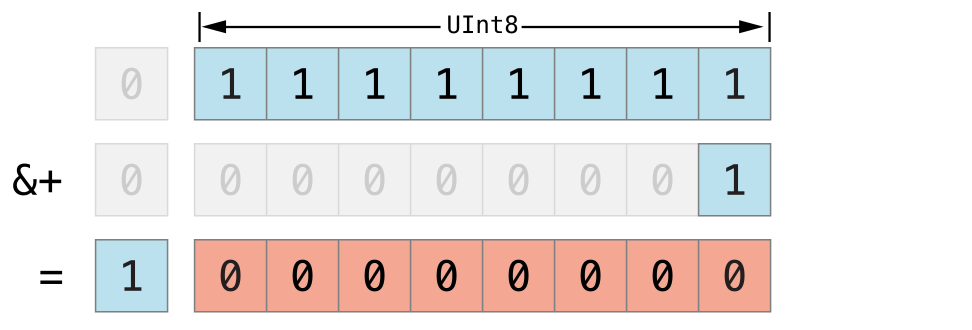
### Value Overflow

Numbers can overflow in both the positive and negative direction.

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

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

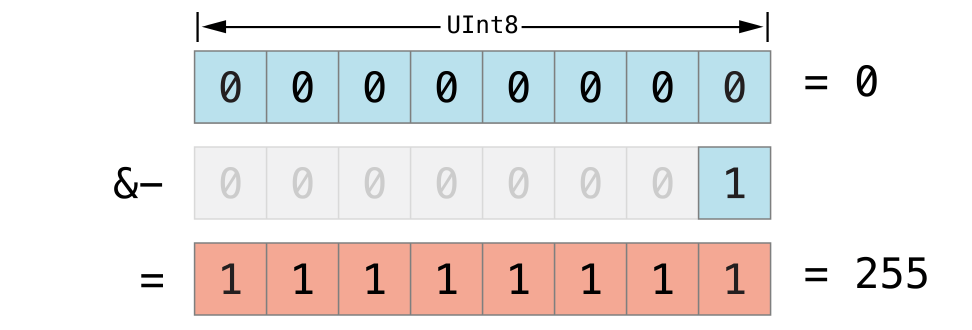
The variable unsignedOverflow is initialized with the maximum value a UInt8 can hold (255, or 11111111 in binary). It is then incremented by 1 using the overflow addition operator (&+). This pushes its binary representation just over the size that a UInt8 can hold, causing it to overflow beyond its bounds,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



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

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

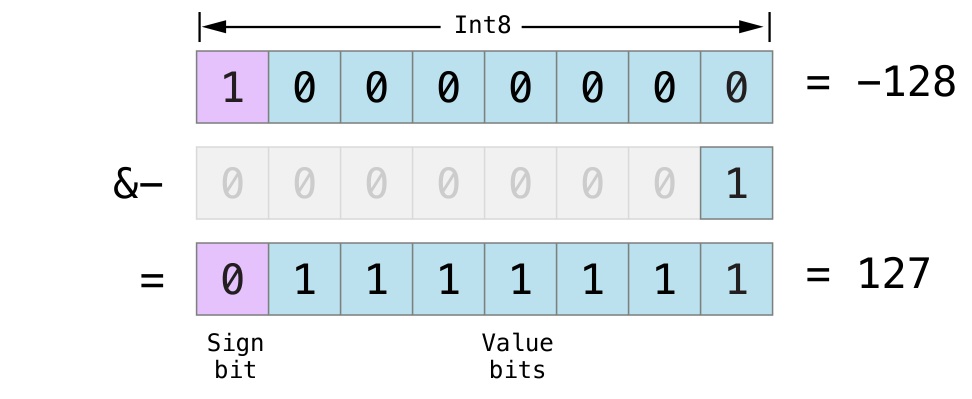
The minimum value that a UInt8 can hold is zero, or 00000000 in binary. If you subtract 1 from 00000000 using the overflow subtraction operator (&-), the number will overflow and wrap around to 11111111, or 255 in decimal.



Overflow also occurs for signed integers. All addition and subtraction for signed integers is performed in bitwise fashion, with the sign bit included as part of the numbers being added or subtracted, as described in [Bitwise Left and Right Shift Operators](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID34).

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

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



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

## Precedence and Associativity

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

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

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

1. 2 + 3 % 4 \* 5
2. // this equals 17

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

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

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

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

1. 2 + ((3 % 4) \* 5)

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

1. 2 + (3 \* 5)

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

1. 2 + 15

This calculation yields the final answer of 17.

For information about the operators provided by the Swift standard library, including a complete list of the operator precedence groups and associativity settings, see [Operator Declarations](https://developer.apple.com/documentation/swift/operator_declarations).

NOTE

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

## Operator Methods

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

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

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

1. struct Vector2D {
2. var x = 0.0, y = 0.0
3. }
4. extension Vector2D {
5. static func + (left: Vector2D, right: Vector2D) -> Vector2D {
6. return Vector2D(x: left.x + right.x, y: left.y + right.y)
7. }
8. }

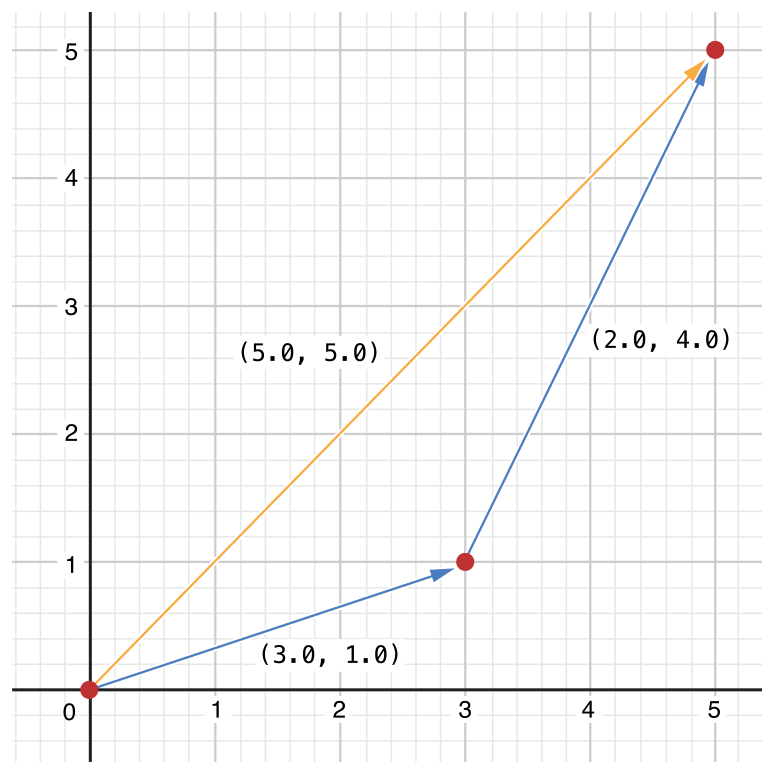
The operator method is defined as a type method on Vector2D, with a method name that matches the operator to be overloaded (+). Because addition isn’t part of the essential behavior for a vector, the type method is defined in an extension of Vector2D rather than in the main structure declaration of Vector2D.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

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

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

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

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



### Prefix and Postfix Operators

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

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

1. extension Vector2D {
2. static prefix func - (vector: Vector2D) -> Vector2D {
3. return Vector2D(x: -vector.x, y: -vector.y)
4. }
5. }

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

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

1. let positive = Vector2D(x: 3.0, y: 4.0)
2. let negative = -positive
3. // negative is a Vector2D instance with values of (-3.0, -4.0)
4. let alsoPositive = -negative
5. // alsoPositive is a Vector2D instance with values of (3.0, 4.0)

### Compound Assignment Operators

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

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

1. extension Vector2D {
2. static func += (left: inout Vector2D, right: Vector2D) {
3. left = left + right
4. }
5. }

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

1. var original = Vector2D(x: 1.0, y: 2.0)
2. let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
3. original += vectorToAdd
4. // original now has values of (4.0, 6.0)

NOTE

It isn’t possible to overload the default assignment operator (=). Only the compound assignment operators can be overloaded. Similarly, the ternary conditional operator (a ? b : c) can’t be overloaded.

### Equivalence Operators

By default, custom classes and structures don’t have an implementation of the equivalence operators, known as the equal to operator (==) and not equal to operator (!=). You usually implement the == operator, and use the standard library’s default implementation of the != operator that negates the result of the == operator. There are two ways to implement the == operator: You can implement it yourself, or for many types, you can ask Swift to synthesize an implementation for you. In both cases, you add conformance to the standard library’s Equatable protocol.

You provide an implementation of the == operator in the same way as you implement other infix operators:

1. extension Vector2D: Equatable {
2. static func == (left: Vector2D, right: Vector2D) -> Bool {
3. return (left.x == right.x) && (left.y == right.y)
4. }
5. }

The example above implements an == operator to check whether two Vector2D instances have equivalent values. In the context of Vector2D, it makes sense to consider “equal” as meaning “both instances have the same x values and y values”, and so this is the logic used by the operator implementation.

You can now use this operator to check whether two Vector2D instances are equivalent:

1. let twoThree = Vector2D(x: 2.0, y: 3.0)
2. let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
3. if twoThree == anotherTwoThree {
4. print("These two vectors are equivalent.")
5. }
6. // Prints "These two vectors are equivalent."

In many simple cases, you can ask Swift to provide synthesized implementations of the equivalence operators for you. Swift provides synthesized implementations for the following kinds of custom types:

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

To receive a synthesized implementation of ==, declare Equatable conformance in the file that contains the original declaration, without implementing an == operator yourself.

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

1. struct Vector3D: Equatable {
2. var x = 0.0, y = 0.0, z = 0.0
3. }
4. let twoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
5. let anotherTwoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
6. if twoThreeFour == anotherTwoThreeFour {
7. print("These two vectors are also equivalent.")
8. }
9. // Prints "These two vectors are also equivalent."

## Custom Operators

You can declare and implement your own custom operators in addition to the standard operators provided by Swift. For a list of characters that can be used to define custom operators, see [Operators](https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html#ID418).

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

1. prefix operator +++

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

1. extension Vector2D {
2. static prefix func +++ (vector: inout Vector2D) -> Vector2D {
3. vector += vector
4. return vector
5. }
6. }
7. var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
8. let afterDoubling = +++toBeDoubled
9. // toBeDoubled now has values of (2.0, 8.0)
10. // afterDoubling also has values of (2.0, 8.0)

### Precedence for Custom Infix Operators

Custom infix operators each belong to a precedence group. A precedence group specifies an operator’s precedence relative to other infix operators, as well as the operator’s associativity. See [Precedence and Associativity](https://docs.swift.org/swift-book/LanguageGuide/AdvancedOperators.html#ID41) for an explanation of how these characteristics affect an infix operator’s interaction with other infix operators.

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

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

1. infix operator +-: AdditionPrecedence
2. extension Vector2D {
3. static func +- (left: Vector2D, right: Vector2D) -> Vector2D {
4. return Vector2D(x: left.x + right.x, y: left.y - right.y)
5. }
6. }
7. let firstVector = Vector2D(x: 1.0, y: 2.0)
8. let secondVector = Vector2D(x: 3.0, y: 4.0)
9. let plusMinusVector = firstVector +- secondVector
10. // plusMinusVector is a Vector2D instance with values of (4.0, -2.0)

This operator adds together the x values of two vectors, and subtracts the y value of the second vector from the first. Because it is in essence an “additive” operator, it has been given the same precedence group as additive infix operators such as + and -. For information about the operators provided by the Swift standard library, including a complete list of the operator precedence groups and associativity settings, see [Operator Declarations](https://developer.apple.com/documentation/swift/operator_declarations). For more information about precedence groups and to see the syntax for defining your own operators and precedence groups, see [Operator Declaration](https://docs.swift.org/swift-book/ReferenceManual/Declarations.html#ID380).

NOTE

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