Optional Chaining

On This Page

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 nonoptional 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:

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}
```

`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`:

```
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:

```
let roomCount = john.residence!.numberOfRooms
// 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:

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// 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 nonoptional value to the `roomCount` variable.

Note that this is true even though `numberOfRooms` is a nonoptional `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:

```
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:

```
class Person {
    var residence: Residence?
}
```

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]`:

```
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        get {
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    func printNumberOfRooms() {
        print("The number of rooms is \(numberOfRooms)")
    }
    var address: Address?
}
```

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:

```
class Room {
    let name: String
    init(name: String) { self.name = name }
}
```

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, 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:

```
let john = Person()
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// 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:

```
let someAddress = Address()
someAddress.buildingNumber = "29"
someAddress.street = "Acacia Road"
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:

```
func printNumberOfRooms() {
    print("The number of rooms is \(numberOfRooms)")
}
```

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. 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:

```
if john.residence?.printNumberOfRooms() != nil {
    print("It was possible to print the number of rooms.")
} else {
    print("It was not possible to print the number of rooms.")
}
// 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 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:

```
if let firstRoomName = john.residence?[0].name {
```

```
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// 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:

```
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:

```
let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "Living Room"))
johnsHouse.rooms.append(Room(name: "Kitchen"))
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// 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:

```
var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]
testScores["Dave"]?[0] = 91
testScores["Bev"]?[0] += 1
testScores["Brian"]?[0] = 72
// 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:

```
if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
// 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 `Address` 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?`:

```
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
    print("John's building identifier is \(buildingIdentifier).")
}
// 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:

```
if let beginsWithThe =
    john.residence?.address?.buildingIdentifier()?.hasPrefix("The") {
    if beginsWithThe {
        print("John's building identifier begins with \"The\".")
    } else {
        print("John's building identifier does not begin with \"The\".")
    }
}
// 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.