Extensions

On This Page

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. For more details, see Protocol Extensions.

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

Extension Syntax

Declare extensions with the `extension` keyword:

```
extension SomeType {
    // new functionality to add to SomeType goes here
}
```

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:

```
extension SomeType: SomeProtocol, AnotherProtocol {
    // implementation of protocol requirements goes here
}
```

Adding protocol conformance in this way is described in Adding Protocol Conformance with an Extension.

An extension can be used to extend an existing generic type, as described in Extending a Generic Type. You can also extend a generic type to conditionally add functionality, as described in Extensions with a Generic Where Clause.

> NOTE
> 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:

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}
let oneInch = 25.4.mm
print("One inch is \(oneInch) meters")
// Prints "One inch is 0.0254 meters"
let threeFeet = 3.ft
print("Three feet is \(threeFeet) meters")
// 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:

```
let aMarathon = 42.km + 195.m
print("A marathon is \(aMarathon) meters long")
// 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.

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:

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
}
```

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. These initializers can be used to create new `Rect` instances:

```
let defaultRect = Rect()
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
                          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:

```
extension Rect {
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

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:

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
                      size: Size(width: 3.0, height: 3.0))
// 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:

```
extension Int {
```

```
    func repetitions(task: () -> Void) {
        for _ in 0..<self {
            task()
        }
    }
}
```

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:

```
3.repetitions {
    print("Hello!")
}
// Hello!
// Hello!
// 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:

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
// 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:

```
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
        for _ in 0..<digitIndex {
            decimalBase *= 10
        }
        return (self / decimalBase) % 10
```

```
    }
}
746381295[0]
// returns 5
746381295[1]
// returns 9
746381295[2]
// returns 2
746381295[8]
// 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:

```
746381295[9]
// returns 0, as if you had requested:
0746381295[9]
```

Nested Types

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

```
extension Int {
    enum Kind {
        case negative, zero, positive
    }
    var kind: Kind {
        switch self {
        case 0:
            return .zero
        case let x where x > 0:
            return .positive
        default:
            return .negative
        }
    }
}
```

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:

```
func printIntegerKinds(_ numbers: [Int]) {
    for number in numbers {
        switch number.kind {
        case .negative:
            print("- ", terminator: "")
        case .zero:
            print("0 ", terminator: "")
        case .positive:
```

```
            print("+ ", terminator: "")
        }
    }
    print("")
}
printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
// 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`.