# Methods

On This Page

*Methods* are functions that are associated with a particular type. Classes, structures, and enumerations can all define instance methods, which encapsulate specific tasks and functionality for working with an instance of a given type. Classes, structures, and enumerations can also define type methods, which are associated with the type itself. Type methods are similar to class methods in Objective-C.

The fact that structures and enumerations can define methods in Swift is a major difference from C and Objective-C. In Objective-C, classes are the only types that can define methods. In Swift, you can choose whether to define a class, structure, or enumeration, and still have the flexibility to define methods on the type you create.

## Instance Methods

*Instance methods* are functions that belong to instances of a particular class, structure, or enumeration. They support the functionality of those instances, either by providing ways to access and modify instance properties, or by providing functionality related to the instance's purpose. Instance methods have exactly the same syntax as functions, as described in Functions.

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

Here's an example that defines a simple `Counter` class, which can be used to count the number of times an action occurs:

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
    func increment(by amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
```

The `Counter` class defines three instance methods:

- `increment()` increments the counter by 1.
- `increment(by: Int)` increments the counter by a specified integer amount.
- `reset()` resets the counter to zero.

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

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

```
let counter = Counter()
// the initial counter value is 0
counter.increment()
// the counter's value is now 1
counter.increment(by: 5)
// the counter's value is now 6
```

```
counter.reset()
// the counter's value is now 0
```

Function parameters can have both a name (for use within the function's body) and an argument label (for use when calling the function), as described in Function Argument Labels and Parameter Names. The same is true for method parameters, because methods are just functions that are associated with a type.

## The self Property

Every instance of a type has an implicit property called `self`, which is exactly equivalent to the instance itself. You use the `self` property to refer to the current instance within its own instance methods.

The `increment()` method in the example above could have been written like this:

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

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

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

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

```
struct Point {
    var x = 0.0, y = 0.0
    func isToTheRightOf(x: Double) -> Bool {
        return self.x > x
    }
}
let somePoint = Point(x: 4.0, y: 5.0)
if somePoint.isToTheRightOf(x: 1.0) {
    print("This point is to the right of the line where x == 1.0")
}
// Prints "This point is to the right of the line where x == 1.0"
```

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

## Modifying Value Types from Within Instance Methods

Structures and enumerations are *value types*. By default, the properties of a value type cannot be modified from within its instance methods.

However, if you need to modify the properties of your structure or enumeration within a particular method, you can opt in to *mutating* behavior for that method. The method can then mutate (that is, change) its properties from within the method, and any changes that it makes are written back to the original structure when the method ends. The method can also assign a completely new instance to its implicit `self` property, and this new instance will replace the existing one when the method ends.

You can opt in to this behavior by placing the `mutating` keyword before the `func` keyword for that method:

```
struct Point {
```

```
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveBy(x: 2.0, y: 3.0)
print("The point is now at (\(somePoint.x), \(somePoint.y))")
// Prints "The point is now at (3.0, 4.0)"
```

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

Note that you cannot call a mutating method on a constant of structure type, because its properties cannot be changed, even if they are variable properties, as described in Stored Properties of Constant Structure Instances:

```
let fixedPoint = Point(x: 3.0, y: 3.0)
fixedPoint.moveBy(x: 2.0, y: 3.0)
// this will report an error
```

## Assigning to self Within a Mutating Method

Mutating methods can assign an entirely new instance to the implicit `self` property. The `Point` example shown above could have been written in the following way instead:

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
```

This version of the mutating `moveBy(x:y:)` method creates a brand new structure whose `x` and `y` values are set to the target location. The end result of calling this alternative version of the method will be exactly the same as for calling the earlier version.

Mutating methods for enumerations can set the implicit `self` parameter to be a different case from the same enumeration:

```
enum TriStateSwitch {
    case off, low, high
    mutating func next() {
        switch self {
        case .off:
            self = .low
        case .low:
            self = .high
        case .high:
            self = .off
        }
    }
}
var ovenLight = TriStateSwitch.low
ovenLight.next()
```

```
// ovenLight is now equal to .high
ovenLight.next()
// ovenLight is now equal to .off
```

This example defines an enumeration for a three-state switch. The switch cycles between three different power states (`off`, `low` and `high`) every time its `next()` method is called.

## Type Methods

Instance methods, as described above, are methods that are called on an instance of a particular type. You can also define methods that are called on the type itself. These kinds of methods are called *type methods*. You indicate type methods by writing the `static` keyword before the method's `func` keyword. Classes may also use the `class` keyword to allow subclasses to override the superclass's implementation of that method.

> NOTE
>
> In Objective-C, you can define type-level methods only for Objective-C classes. In Swift, you can define type-level methods for all classes, structures, and enumerations. Each type method is explicitly scoped to the type it supports.

Type methods are called with dot syntax, like instance methods. However, you call type methods on the type, not on an instance of that type. Here's how you call a type method on a class called `SomeClass`:

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
SomeClass.someTypeMethod()
```

Within the body of a type method, the implicit `self` property refers to the type itself, rather than an instance of that type. This means that you can use `self` to disambiguate between type properties and type method parameters, just as you do for instance properties and instance method parameters.

More generally, any unqualified method and property names that you use within the body of a type method will refer to other type-level methods and properties. A type method can call another type method with the other method's name, without needing to prefix it with the type name. Similarly, type methods on structures and enumerations can access type properties by using the type property's name without a type name prefix.

The example below defines a structure called `LevelTracker`, which tracks a player's progress through the different levels or stages of a game. It is a single-player game, but can store information for multiple players on a single device.

All of the game's levels (apart from level one) are locked when the game is first played. Every time a player finishes a level, that level is unlocked for all players on the device. The `LevelTracker` structure uses type properties and methods to keep track of which levels of the game have been unlocked. It also tracks the current level for an individual player.

```
struct LevelTracker {
    static var highestUnlockedLevel = 1
    var currentLevel = 1

    static func unlock(_ level: Int) {
        if level > highestUnlockedLevel { highestUnlockedLevel = level }
    }

    static func isUnlocked(_ level: Int) -> Bool {
        return level <= highestUnlockedLevel
```

```
    }

    @discardableResult
    mutating func advance(to level: Int) -> Bool {
        if LevelTracker.isUnlocked(level) {
            currentLevel = level
            return true
        } else {
            return false
        }
    }
}
```

The `LevelTracker` structure keeps track of the highest level that any player has unlocked. This value is stored in a type property called `highestUnlockedLevel`.

`LevelTracker` also defines two type functions to work with the `highestUnlockedLevel` property. The first is a type function called `unlock(_:)`, which updates the value of `highestUnlockedLevel` whenever a new level is unlocked. The second is a convenience type function called `isUnlocked(_:)`, which returns `true` if a particular level number is already unlocked. (Note that these type methods can access the `highestUnlockedLevel` type property without your needing to write it as `LevelTracker.highestUnlockedLevel`.)

In addition to its type property and type methods, `LevelTracker` tracks an individual player's progress through the game. It uses an instance property called `currentLevel` to track the level that a player is currently playing.

To help manage the `currentLevel` property, `LevelTracker` defines an instance method called `advance(to:)`. Before updating `currentLevel`, this method checks whether the requested new level is already unlocked. The `advance(to:)` method returns a Boolean value to indicate whether or not it was actually able to set `currentLevel`. Because it's not necessarily a mistake for code that calls the `advance(to:)` method to ignore the return value, this function is marked with the `@discardableResult` attribute. For more information about this attribute, see Attributes.

The `LevelTracker` structure is used with the `Player` class, shown below, to track and update the progress of an individual player:

```
class Player {
    var tracker = LevelTracker()
    let playerName: String
    func complete(level: Int) {
        LevelTracker.unlock(level + 1)
        tracker.advance(to: level + 1)
    }
    init(name: String) {
        playerName = name
    }
}
```

The `Player` class creates a new instance of `LevelTracker` to track that player's progress. It also provides a method called `complete(level:)`, which is called whenever a player completes a particular level. This method unlocks the next level for all players and updates the player's progress to move them to the next level. (The Boolean return value of `advance(to:)` is ignored, because the level is known to have been unlocked by the call to `LevelTracker.unlock(_:)` on the previous line.)

You can create an instance of the `Player` class for a new player, and see what happens when the player completes level one:

```
var player = Player(name: "Argyrios")
player.complete(level: 1)
print("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")
```

```
// Prints "highest unlocked level is now 2"
```

If you create a second player, whom you try to move to a level that is not yet unlocked by any player in the game, the attempt to set the player's current level fails:

```
player = Player(name: "Beto")
if player.tracker.advance(to: 6) {
    print("player is now on level 6")
} else {
    print("level 6 has not yet been unlocked")
}
// Prints "level 6 has not yet been unlocked"
```