Methods

*Methods*是与实际类型关联的*Functions*。

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.

Instance Methods

Instance methods have exactly the same syntax as functions, as described in Functions.

实例方法可以隐式访问该类型的所有其他实例方法和属性。实例方法只能在其所属类型的特定实例上调用。没有现有实例，就不能孤立地调用它。

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

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

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.

当实例方法的参数名称与该实例的属性名称相同时，将发生此规则的主要例外。在这种情况下，参数名称优先，因此有必要以更限定的方式引用该属性。您可以使用`self`属性来区分参数名称和属性名称。

```
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"
```

如果没有`self`前缀，Swift会假设`x`的两种用法都引用了称为`x`的方法参数。

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)"
```

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 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 you call 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 can use the `class` keyword instead, to allow subclasses to override the superclass's implementation of that method.**

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.

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.