方法

*方法*是与特定类型相关的函数。类，结构和枚举都可以定义实例方法，它们封装了与给定类型的实例一起工作的特定任务和功能。类，结构和枚举也可以定义类型方法，它们与类型本身相关联。类型方法与Objective-C中的类方法类似。

结构和枚举可以在Swift中定义方法这一事实与C和Objective-C的主要区别在于。在Objective-C中，类是唯一可以定义方法的类型。在Swift中，您可以选择是定义类，结构还是枚举，还可以灵活地根据所创建的类型定义方法。

实例方法

*实例方法*是属于特定类，结构或枚举实例的函数。它们通过提供访问和修改实例属性的方法，或者通过提供与实例目的相关的功能来支持这些实例的功能。实例方法具有完全相同的语法功能，如描述的功能。

你在它所属的类型的开始和结束花括号内编写一个实例方法。一个实例方法具有对该类型的所有其他实例方法和属性的隐式访问。实例方法只能在其所属类型的特定实例上调用。不能在没有现有实例的情况下独立调用它。

以下是一个定义简单`Counter`类的示例，可用于计算操作发生的次数：

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

在`Counter`类定义了三个实例方法：

- `increment()`通过增加计数器1。
- `increment(by: Int)` 将计数器递增指定的整数量。
- `reset()` 将计数器重置为零。

本`Counter`类还声明一个变量属性`count`，以跟踪当前计数器值。

您可以使用与属性相同的点语法来调用实例方法：

```
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
```

函数参数可以同时具有一个名称（用于在函数体内使用）和一个参数标签（用于调用该函数时），如函数参数标签和参数名称中所述。方法参数也是如此，因为方法只是与类型相关的函数。

自己的财产

每个类型的实例都有一个隐含的属性`self`，这个属性完全等同于实例本身。您可以使用该`self`属性在其自己的实例方法中引用当前实例。

上例中的`increment()`方法可能是这样写的：

```
func increment() {
    sel
}
```

在实践中，你不需要`self`经常写代码。如果您没有明确写入`self`，则Swift假定您在方法中使用已知属性或方法名称时引用当前实例的属性或方法。这个假设通过在三个实例方法中使用`count`（而不是`self.count`）来证明`Counter`。

此规则的主要例外情况发生在实例方法的参数名称与该实例的属性名称相同时。在这种情况下，参数名称优先，并且有必要以更合适的方式引用该属性。您可以使用该`self`属性来区分参数名称和属性名称。

在此，`self`调用方法参数`x`和也称为实例属性之间的歧义`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"
```

如果没有`self`前缀，Swift会假定两个使用`x`的方法参数都被称为`x`。

从实例方法中修改值类型

结构和枚举是*值类型*。默认情况下，值类型的属性不能从其实例方法中修改。

但是，如果您需要在特定方法中修改结构或枚举的属性，则可以选择对该方法进行*变异*。然后该方法可以从方法内部改变（也就是改变）它的属性，并且当方法结束时，它所做的任何改变都会被写回原始结构。该方法还可以为其隐式`self`属性指定一个全新的实例，并且该方法结束时，此新实例将替换现有的实例。

您可以通过将`mutating`关键字放在该方法的关键字之前加入此行为`func`：

```
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)"
```

`Point`上面 的结构定义了一个变异`moveBy(x:y:)`方法，它将一个`Point`实例移动一定数量。这种方法不是返回一个新点，而是实际上修改了它被调用的点。该`mutating`关键字被添加到其定义中以使其能够修改其属性。

请注意，您不能在结构类型的常量上调用变异方法，因为它的属性无法更改，即使它们是可变属性，如常量结构实例的存储属性中所述：

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

在变异方法中赋值给自己

突变方法可以为隐式`self`属性分配一个全新的实例。`Point`上面显示的例子可以用下面的方式写成：

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

此版本的变异`moveBy(x:y:)`方法会创建一个全新的结构，其值`x`和`y`值设置为目标位置。调用该方法的替代版本的最终结果与调用早期版本的结果完全相同。

枚举的变异方法可以将隐式`self`参数设置为与相同枚举不同的情况：

```
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
```

这个例子定义了一个三态开关的枚举。每次调用其方法时`off`，开关在三种不同的功率状态（`low`和`high`）之间循环`next()`。

类型方法

如上所述，实例方法是在特定类型的实例上调用的方法。您也可以定义在类型本身上调用的方法。这些类型的方法称为*类型方法*。通过`static`在方法关键字前写入关键字来指示类型方法`func`。类也可以使用`class`关键字来允许子类重写超类的该方法的实现。

> 注意
>
> 在Objective-C中，只能为Objective-C类定义类型级方法。在Swift中，您可以为所有类，结构和枚举定义类型级别的方法。每种类型的方法都被显式限定为它所支持的类型。

类型方法使用点语法调用，如实例方法。但是，您在类型上调用类型方法，而不是在该类型的实例上调用。以下是在类中调用类型方法的方法`SomeClass`：

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

在类型方法的主体中，隐式`self`属性指的是类型本身，而不是该类型的实例。这意味着您可以使用`self`在类型属性和类型方法参数之间消除歧义，就像您对实例属性和实例方法参数所做的一样。

更一般地说，在类型方法体内使用的任何不合格的方法和属性名称都会引用其他类型级别的方法和属性。一个类型方法可以用另一个方法的名字调用另一个类型方法，而不需要用类型名称作为前缀。同样，结构和枚举上的类型方法可以通过使用type属性的名称而不使用类型名称前缀来访问类型属性。

下面的例子定义
人游戏，但可以在单个设备上存储多个玩家的信息。

当游戏第一次玩时，游戏的所有等级（除了等级1）都被锁定。每次玩家完成一个关卡时，该关卡都会在设备上的所有玩家上锁。该LevelTracker结构使用类型属性和方法来跟踪哪些级别的游戏已被解锁。它也跟踪单个玩家的当前水平。

```
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
        }
    }
}
```

该LevelTracker结构会跟踪任何玩家解锁的最高级别。该值存储在一个名为的类型属性中highestUnlockedLevel。

LevelTracker还定义了两种类型的函数来处理highestUnlockedLevel属性。第一种是被称为的类型函数unlock(_:)，它在highestUnlockedLevel每次解锁新级别时更新值。第二种是称为的便利型功能isUnlocked(_:)，true如果特定的等级号码已经解锁，则返回该功能。（请注意，这些类型的方法可以访问highestUnlockedLevel类型属性，而无需将其写入为LevelTracker.highestUnlockedLevel。）

除了其类型属性和类型方法之外，还可以LevelTracker跟踪单个玩家在游戏中的进度。它使用调用的实例属性currentLevel来跟踪玩家当前正在玩的级别。

为了帮助管理currentLevel属性，LevelTracker定义了一个名为的实例方法advance(to:)。在更新之前currentLevel，此方法检查所请求的新级别是否已解锁。该advance(to:)方法返回一个布尔值来指示它是否实际能够设置currentLevel。因为调用advance(to:)方法忽略返回值的代码不一定是错误的，所以此函数被标记为@discardableResult属性。有关此属性的更多信息，请参阅属性。

该LevelTracker结构与Player课程一起使用，如下所示，用于跟踪和更新单个玩家的进度：

```
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
    }
}
```

该Player课程创建一个新的实例LevelTracker来跟踪该玩家的进度。它还提供了一种称为的方法complete(level:)，只要玩家完成特定级别，该方法就会被调用。该方法为所有玩家释放下一个等级，并更新

玩家将其移动到下一个等级的进度。（布尔返回值将`advance(to:)`被忽略，因为已知该级别已被`LevelTracker.unlock(_:)`上一行的调用解锁。）

您可以`Player`为新玩家创建类的实例，并查看玩家完成第一级时会发生的情况：

```
var player = Player(name: "Argyrios")
player.complete(level: 1)
print("highest unlocked level is now \(LevelTracker.highestUnlockedLevel)")
// Prints "highest unlocked level is now 2"
```

如果您创建第二个玩家，您尝试移动到尚未被游戏中任何玩家解锁的级别，则设置玩家当前级别的尝试将失败：

```
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"
```