Вычисляемые свойства в расширениях

Расширения могут добавлять вычисляемые свойства экземпляра и вычисляемые свойства типа к существующим типам. В примере мы добавляем пять вычисляемых свойств экземпляра во встроенный тип `Double` языка Swift, для обеспечения работы данного типа с единицами длины:

```
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("Один дюйм - это \(oneInch) метра")
// Выведет "Один дюйм- это 0.0254 метра"
let threeFeet = 3.ft
print("Три фута - это \(threeFeet) метра")
// Выведет "Три фута - это 0.914399970739201 метра"
```

Эти вычисляемые свойства объясняют, что тип `Double` должен считаться как конкретная единица измерения длины. Хотя они реализованы как вычисляемые свойства, имена этих свойств могут быть добавлены к литералу чисел с плавающей точкой через точечный синтаксис, как способ использовать значения литерала для проведения преобразований длины.

В этом примере, значение *1.0* типа `Double` отображает “один метр”. Это причина, по которой `m` возвращает `self`, что равно *1.m*, то есть посчитать `Double` от числа *1.0*.

Другие единицы требуют некоторых преобразований, чтобы выражать свое значение через метры. Один километр то же самое что и *1000* метров, так что `km` - вычисляемое свойство, которое умножает значение на *1_000.00*,

чтобы отобразить величину в метрах. По аналогии поступаем и с остальными свойствами, как например, с футом, футов в одном метре насчитывается *3.28024*, так что вычисляемое свойство `ft` делит подчеркнутое значение `Double` на *3.28024* для того, чтобы перевести футы в метры.

Эти свойства являются вычисляемыми свойствами только для чтения, так что они могут быть выражены без ключевого слова `get`. Их возвращаемое значение является типом `Double` и может быть использовано в математических вычислениях, где поддерживается тип `Double`:

```
let aMarathon = 42.km + 195.m
print("Марафон имеет длину \(aMarathon) метров")
// Выведет "Марафон имеет длину 42195.0 метров"
```

Заметка

Расширения могут добавлять новые вычисляемые свойства, но они не могут добавить свойства хранения или наблюдателей свойства к уже существующим свойствам.

Инициализаторы в расширениях

Расширения могут добавить новые инициализаторы существующему типу. Это позволяет вам расширить другие типы для принятия ваших собственных типов в качестве параметров инициализатора, или для обеспечения дополнительных опций инициализации, которые не были включены как часть первоначальной реализации типа.

Расширения могут добавлять вспомогательные инициализаторы классу, но они не могут добавить новый назначенный инициализатор или деинициализатор классу. Назначенные инициализаторы и деинициализаторы должны всегда предоставляться реализацией исходного класса.

Если вы используете расширения для того, чтобы добавить инициализатор к типу значений, который обеспечивает значения по умолчанию для всех своих хранимых свойств и не определяет какого-либо пользовательского инициализатора, то вы можете вызвать дефолтный инициализатор и почленный инициализатор для того типа значений изнутри инициализатора вашего расширения. Это не будет работать, если вы уже написали инициализатор как часть исходной реализации значения типа, подробнее см. Делегирование инициализатора для типов значения.

Если вы используете расширение для добавления инициализатора в структуру, которая была объявлена в другом модуле, новый инициализатор не может получить доступ к себе до тех пор, пока он не вызовет инициализатор из модуля определения.

Пример ниже определяет структуру `Rect` для отображения геометрического прямоугольника. Пример так же определяет две вспомогательные структуры `Size` и `Point`, обе из которых предоставляют значения по умолчанию 0.0 для всех своих свойств:

```
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()
}
```

Из-за того, что структура `Rect` предоставляет значения по умолчанию для всех своих свойств, она автоматически получает инициализатор по умолчанию и почленный инициализатор, что описано в главе Дефолтные инициализаторы.

Эти инициализаторы могут быть использованы для создания экземпляров `Rect`:

```
let defaultRect = Rect()
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
                          size: Size(width: 5.0, height: 5.0))
```

Вы можете расширить структуру *Rect* для предоставления дополнительного инициализатора, который принимает определенную точку и размер:

```
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)
    }
}
```

Этот новый инициализатор начинается с вычисления исходной точки, основываясь на значениях свойств `center` и `size`. Потом инициализатор вызывает почленный инициализатор структуры `init(origin:size:)`, который хранит новую исходную точку и размеры в соответствующих свойствах:

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
                      size: Size(width: 3.0, height: 3.0))
// исходная точка centerRect (2.5, 2.5) и его размер (3.0, 3.0)
```

Заметка

Если вы предоставляете новый инициализатор вместе с расширением, вы все еще ответственены за то, что каждый экземпляр должен быть полностью инициализирован, к моменту, когда инициализатор заканчивает свою работу.