Урок 6

Продвинутое ООП

Техники настоящих гуру. Дженерики. Subscripting.

Дженерики

На прошлом уроке мы реализовали для завода конвейер, определяющий периметр различных изделий. Но в конце конвейера наши детали падают в кучу и создают беспорядок. Можно было бы сложить их в линию, но она получится очень длинной, будет занимать много места, и искать нужный тип детали в ней будет слишком долго. Мы, как инженеры, решили сделать специальную конструкцию для хранения изделий. Идея следующая: устанавливаем трубу, уходящую глубоко в пол, и помещаем детали в нее. Внутри будет механизм, регулирующий глубину трубы, второе дно. Чем больше деталей в трубе, тем глубже опускается второе дно, и наоборот. Глубина всегда будет такой, чтобы верхняя деталь выступала из трубы и ее можно было легко взять. Это очень удобно: рабочий может подойти и в любой момент взять деталь. Более того, установим столько труб, сколько типов деталей у нас есть, ведь рабочим нужна деталь конкретного типа.

В программировании есть похожая структура данных, она называется **стек**. В отличие от массива, элементы в нее можно добавлять только в конец и с конца же забирать. Элемент, помещенный в стек последним, извлекается первым. К сожалению, в Swift нет такой структуры. Давайте ее напишем.

Создадим структуру с одним свойством (массивом для хранения элементов) и двумя методами (добавления и извлечения элементов).

```
class Rectangle {
    var sideA: Double
    var sideB: Double
    func calculatePerimiter() -> Double {
        return sideA + sideB
    }
    init(sideA: Double, sideB: Double) {
        self.sideA = sideA
        self.sideB = sideB
    }
}
struct Stack {                                      // коллекция типа стек
    private var elements: [Rectangle] = []      // массив, где мы будем хранить
элементы
    mutating func push(_ element: Rectangle) { // добавляем элемент в конец
массива
        elements.append(element)
    }
    mutating func pop() -> Rectangle? {      // извлекаем элемент из массива
        return elements.removeLast()
    }
}
var stack = Stack()                             // создаем пустой стек
// добавляем элементы
stack.push(Rectangle(sideA: 10, sideB: 20))
stack.push(Rectangle(sideA: 2, sideB: 2))
stack.push(Rectangle(sideA: 17, sideB: 90))
stack.push(Rectangle(sideA: 10, sideB: 3))
//извлекаем элементы
stack.pop()                                    // Rectangle(10,3)
stack.pop()                                   // Rectangle(17,90)
```

Отлично, наш стек работает. Но в него можно класть только прямоугольные детали, а нам нужно несколько стеков для разных деталей. Можно, конечно, объявить еще «CircleStack». Но это не очень удобно. Есть один способ создавать структуры, классы и методы заранее неизвестного типа. Такие структуры называются **дженериками**.

Дженерик – это такое описание данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание.

При объявлении нашего стека рядом с именем структуры мы в треугольных скобках объявим новый тип данных – «Stack<T>». На самом деле никакого типа «T» не существует, это описание заранее неизвестного типа данных, который будет указан в момент создания экземпляра структуры. Не обязательно называть наш тип «T», это лишь общепринятый пример, вы можете назвать его как угодно. Далее внутри нашей структуры мы можем использовать этот тип, объявлять массив типа «T», методы будут принимать и возвращать тип «T». Когда затем мы создадим экземпляр нашего стека, мы укажем, какой на самом деле тип надо использовать, – «Stack<Rectangle>». После этого все места, где было указано «T», будут заменены на «Rectangle».

```swift
class Circle {
    var radius: Double
    func calculatePerimiter() -> Double {
        return 2.0 * Double.pi * radius
    }
    init(radius: Double) {
        self.radius = radius
    }
}
class Rectangle {
    var sideA: Double
    var sideB: Double
    func calculatePerimiter() -> Double {
        return sideA + sideB
    }
    init(sideA: Double, sideB: Double) {
        self.sideA = sideA
        self.sideB = sideB
    }
}
struct Stack<T> {                    // T - это какой-то пока неизвестный тип
    private var elements: [T] = []           // массив типа T
    mutating func push(_ element: T) {      // добавляем элемент типа T
        elements.append(element)
    }
    mutating func pop() -> T? {            // извлекаем элемент типа T
        return elements.removeLast()
    }
}
var stackRectangle = Stack<Rectangle>()  // создаем стек типа Rectangle
var stackCircle = Stack<Circle>()       // создаем стек типа Circle
// добавляем элементы
stackRectangle.push(Rectangle(sideA: 10, sideB: 20))
stackRectangle.push(Rectangle(sideA: 2, sideB: 2))
stackCircle.push(Circle(radius: 5))
stackCircle.push(Circle(radius: 5))
```

Хранилища установлены и используются, свалки деталей больше нет. Но руководство опять хочет нововведений. Требуется на каждом хранилище установить дисплей, который будет отображать общий вес деталей в нем. Казалось бы, все просто: добавим деталям свойство «вес», а в стек – вычисляемое свойство, суммирующее вес всех деталей. Но есть одна проблема. Так как тип данных «T» еще не определен на этапе описания стека, мы не можем быть уверенными, что он будет иметь какие-либо свойства. На помощь придут протоколы и возможность уточнять тип «T».

Если мы при объявлении неизвестного типа укажем, что он поддерживает протокол, мы больше не сможем создавать экземпляры дженерика для типов, не поддерживающих его. Так мы сократим область применения, но получим дополнительные возможности внутри дженерика.

```swift
protocol Weightabe {                        // создаем протокол поддержки веса
    var weight: Double { get set }
}
class Circle: Weightabe {                   // имплементируем протокол кругу
    var radius: Double
    var weight: Double
    func calculatePerimiter() -> Double {
        return 2.0 * Double.pi * radius
    }
    init(radius: Double, weight: Double) {
        self.radius = radius
        self.weight = weight
    }
}
class Rectangle: Weightabe {        // имплементируем протокол прямоугольнику
    var sideA: Double
    var sideB: Double
    var weight: Double
    func calculatePerimiter() -> Double {
        return sideA + sideB
    }
    init(sideA: Double, sideB: Double, weight: Double) {
        self.sideA = sideA
        self.sideB = sideB
        self.weight = weight
    }
}
// T - это какой-то пока неизвестный тип, но он поддерживает протокол
Weightabe
struct Stack<T: Weightabe> {
    private var elements: [T] = []            // массив типа T

    mutating func push(_ element: T) {      // добавляем элемент типа T
        elements.append(element)
    }
    mutating func pop() -> T? {             // извлекаем элемент типа T
        return elements.removeLast()
    }
    var totalWeght : Double {              // свойство, отражающее общий вес
деталей
        var weight = 0.0
        for element in elements {
            weight += element.weight    // мы можем использовать свойство
weight
        }
        return weight
    }
}
```

При уточнении типа вы можете указать один класс и несколько протоколов.

Subscripting

Подсчет веса всех деталей оказался таким полезным, что нас попросили добавить возможность подсчета веса конкретных деталей. Добавим и ее.

Удобнее всего указывать элементы в квадратных скобках, как в массиве. Этого можно добиться, описав у стека метод «subscripting».

```
struct Stack<T: Weightabe> {
    private var elements: [T] = []
    mutating func push(_ element: T) {
        elements.append(element)
    }
    mutating func pop() -> T? {
        return elements.removeLast()
    }
    var totalWeght : Double {
        var weight = 0.0
        for element in elements {
            weight += element.weight
        }
        return weight
    }
    subscript(indices: UInt ...) -> Double {  // доступ к стеку по индексу
        var weight = 0.0
        // перебираем все элементы по переданным индексам, пропускаем
       // те индексы, которые лежат за пределами массива
        for index in indices where index < self.elements.count {
            weight += self.elements[Int(index)].weight
        }
        return weight
    }
}
var stack = Stack<Circle>()
stack.push(Circle(radius: 4, weight: 12))
stack.push(Circle(radius: 4, weight: 12))
stack.push(Circle(radius: 4, weight: 12))
stack.push(Circle(radius: 4, weight: 12))
stack[0,2,3]                                    // 36
```

«Subscripting» очень похож на обычный метод, он также принимает и возвращает переменные. Можно объявить несколько «subscripting» с разными входными и возвращаемыми параметрами. Основных отличий два: во-первых, он вызывается не по имени, а через квадратные скобки; во-вторых, ему можно присваивать значение.

Домашнее задание

1. Реализовать свой тип коллекции «очередь» (queue) с использованием дженериков.

2. *Добавить свой subscript, который будет возвращать nil в случае обращения к несуществующему индексу.

Дополнительные материалы

1. https://developer.apple.com/library/prerelease/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309.