Основы языка Swift

Урок 6

Продвинутое ООП

Техники настоящих гуру. Дженерики. Замыкания. Subscripting. Функции высшего порядка.

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(elements: Int ...) -> 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 elements where index < self.elements.count {
            weight += self.elements[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.

Замыкания

Замыкания очень похожи на функции, они могут принимать аргументы и возвращать результат. Но это только внешнее сходство. На самом деле замыкание – это анонимный блок кода. Оно не имеет имени, не привязано к чему-либо, его можно хранить в переменных, передавать аргументом в методы. Замыкание имеет свою область памяти и передается по ссылкам, как объекты. Замыкание

может захватывать и удерживать внутри себя любые переменные, и они будут существовать в памяти, даже если все остальные ссылки на них будут уничтожены. Замыкание имеет свой тип: (тип_аргумента) -> тип_возвращаемого_значения.

Напишем простое замыкание, которое принимает два аргумента и возвращает результат сравнения.

```
// замыкание имеет тип, его можно присвоить переменной
let closure: (Int, Int) -> Bool = { (itemOne: Int, itemTwo: Int) -> Bool in
    return itemOne == itemTwo
}
closure(2, 2) // true
```

Этот пример мало чем отличается от обычной функции. Чтобы оценить всю прелесть замыканий, напишем функцию высшего порядка. Несмотря на громкое название, функция высшего порядка – это обычная функция, которая в качестве аргументов принимает замыкания.

Функция будет принимать два аргумента: массив целых чисел и замыкание. Замыкание будет принимать целое число и возвращать логическое значение. Внутри функции будет перебираться массив. Каждый его элемент будет отправлен в замыкание, и если оно вернет истину, элемент будет скопирован во временный массив. В конце функция вернет временный массив. Таким образом мы фильтруем исходный массив. Также напишем два замыкания: одно будет проверять элемент на четность, второе – на нечетность. Объявим массив целых чисел, а затем вызовем функцию высшего порядка и передадим ей массив и замыкание.

```
// определяет четное число
let odd: (Int) -> Bool = { (element: Int) -> Bool in
    return element % 2 == 0
}
// определяет нечетное число
let even: (Int) -> Bool = { (element: Int) -> Bool in
    return element % 2 != 0
}
var array = [1,2,3,4,5,6,7,8,9,10]
// принимает два аргумента - массив и замыкание
func filter(array: [Int], predicate: (Int) -> Bool ) -> [Int] {
    var tmpArray = [Int]()               // создает временный массив
    for element in array {
        if predicate(element) {          // вызываем замыкание, чтобы проверить
элемент
            tmpArray.append(element)
        }
    }
    return tmpArray                      // возвращаем отфильтрованный массив
}
filter(array: array, predicate: odd)    // [2, 4, 6, 8, 10]
filter(array: array, predicate: even) // [1, 3, 5, 7, 9]
```

Мы только что написали функцию, результат которой зависит от переданного в нее замыкания. Этот прием очень часто используется в программировании.

В примере выше мы сначала присваивали замыкание переменной, а потом отправляли в функцию, но можно объявить замыкание в момент вызова функции высшего порядка. Такой вариант более удобен.

```
filter(array: array, predicate: { (element: Int) -> Bool in
    return element % 2 == 0
}) // [2, 4, 6, 8, 10]
```

Есть такой термин, как **закрывающее замыкание**. Если замыкание передается последним параметром, его можно вынести за круглые скобки, что сделает синтаксис более привлекательным.

```
filter(array: array) { (element: Int) -> Bool in
    return element % 2 == 0
} // [2, 4, 6, 8, 10]
```

Сами замыкания тоже можно очень сильно сокращать, используя механизм вывода типов, встроенный в язык. Например, функция «filter» определяет тип для замыкания, и компилятор может автоматически выводить его у переданного замыкания.

Во-первых, мы можем убрать возвращаемый тип, он будет выведен из контекста.

```
filter(array: array) { (element: Int) in
    return element % 2 == 0
} // [2, 4, 6, 8, 10]
```

Во-вторых, мы можем убрать тип у аргументов.

```
filter(array: array) { (element) in
    return element % 2 == 0
} // [2, 4, 6, 8, 10]
```

Если мы не пишем типы у аргументов и возвращаемого значения, можно убрать и круглые скобки.

```
filter(array: array) { element in
    return element % 2 == 0
} // [2, 4, 6, 8, 10]
```

Так как из контекста понятно, что в замыкание будет передан один аргумент, можно убрать и его, а в теле замыкания он будет доступен через переменную «$0». Если бы переменных передавалось несколько, например три, они были бы доступны по именам «$0», «$1», «$2». Так как мы убираем имя аргумента, мы обязаны убрать и ключевое слово «in».

```
filter(array: array) {
    return $0 % 2 == 0
} // [2, 4, 6, 8, 10]
```

Так как у нас осталась одна строка в замыкании, компилятор поймет, что ее результат и есть возвращаемое значение, и слово «return» мы тоже можем не писать.

```
filter(array: array) {
    $0 % 2 == 0
}// [2, 4, 6, 8, 10]
```

Это самый сокращенный вариант замыкания. Не всегда получается настолько его сократить, и не всегда стоит это делать. Если тело замыкания большое, слово «return» и имена переменных лучше оставить, чтобы было легче читать. Давайте перепишем последний пример в одну строку, чтобы получилось максимально красиво.

```
filter(array: array) { $0 % 2 == 0 }// [2, 4, 6, 8, 10]
```

Посмотрим, что у нас получилось. С помощью такой компактной записи мы можем отфильтровать любой массив целых чисел по любым условиям. А если мы сделаем функцию высшего порядка дженериком, мы сможем фильтровать любые массивы.

На самом деле это настолько удобно и часто используется, что в стандартной библиотеке есть такая функция высшего порядка.

```
var array = [1,2,3,4,5,6,7,8,9,10]
array.filter{ $0 % 2 == 0 }  // [2, 4, 6, 8, 10]
array.filter{ $0 % 2 != 0 } // [1, 3, 5, 7, 9]
```

Кроме того, есть стандартные функции:

- «sort» – сортировки массива.
- «map» – преобразования массива.
- «reduce» – вычисляет результат агрегирования элементов.
- «forEach» – итерирование массива по аналогии с for.

```
var array = [1,2,3,4,5,6,7,8,9,10]
array.sort { $0 > $1 }         // [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
array.map { $0 * $0 }          // [100, 81, 64, 49, 36, 25, 16, 9, 4, 1]
array.reduce(0){ $0 + $1 }     // 55
array.reduce(1){ $0 * $1 }     // 3628800
array.forEach{
    print($0)
}
```

Последнее, что нужно знать про замыкания, – они захватывают все ссылки как сильные («strong»). Это может сыграть с вами очень злую шутку: вы можете запросто создать цикл удержания.

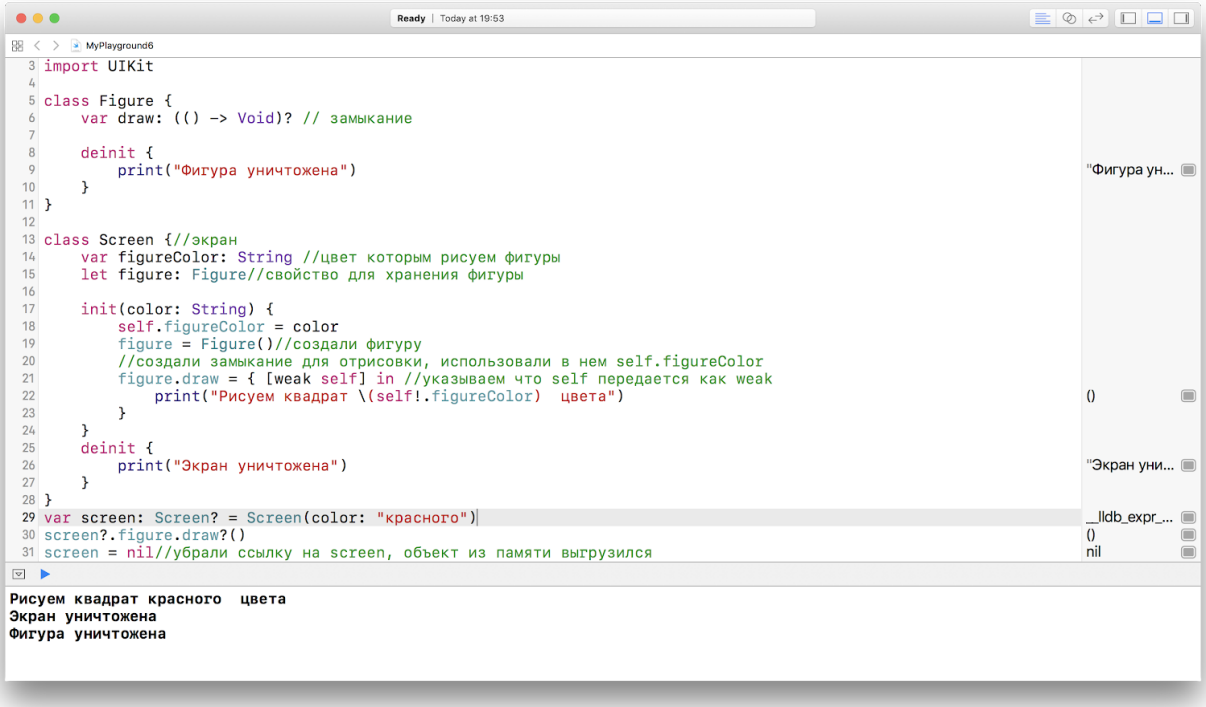
Напишем простой пример. Определим два класса – «экран» и «фигура». Экран будет содержать фигуру и цвет для отрисовки фигур. Фигура будет содержать замыкание для отрисовки себя на экране. При создании класса экрана мы создаем фигуру и замыкание для нее, в замыкании используем свойство класса «figureColor», и замыкание захватит его сильной ссылкой.

```
class Figure {
    var draw: (() -> Void)?     // замыкание
    deinit {
        print("Фигура уничтожена")
    }
}
class Screen { //экран
    var figureColor: String  // цвет, которым рисуем фигуры
```

```
    let figure: Figure      // свойство для хранения фигуры
    init(color: String) {
        self.figureColor = color
        figure = Figure()     // создали фигуру
// создали замыкание для отрисовки, использовали в нем self.figureColor
        figure.draw = { print("Рисуем квадрат \(self.figureColor)  цвета") }
    }
    deinit {
        print("Экран уничтожен")
    }
}
var screen: Screen? = Screen(color: "красного")
screen?.figure.draw?()
screen = nil  // убрали ссылку на screen, но объект из памяти не выгружается
```

В результате у нас получился цикл удержания: экран удерживает фигуру, фигура удерживает замыкание, замыкание удерживает экран. Это не совсем очевидно, и нужно быть готовым к таким ситуациям. Чтобы разорвать этот круг, надо передать «self» в замыкание по слабой ссылке.

```
class Figure {
    var draw: (() -> Void)?                    // замыкание
    deinit {
        print("Фигура уничтожена")
    }
}
class Screen {                                 // экран
    var figureColor: String                   // цвет, которым рисуем фигуры
    let figure: Figure                        // свойство для хранения фигуры
    init(color: String) {
        self.figureColor = color
        figure = Figure()                     // создали фигуру
// создали замыкание для отрисовки, использовали в нем self.figureColor
        figure.draw = { [weak self] in // указываем, что self передается как
weak
            print("Рисуем квадрат \(self!.figureColor)  цвета")
        }
    }
    deinit {
        print("Экран уничтожена")
    }
}
var screen: Screen? = Screen(color: "красного")
screen?.figure.draw?()
screen = nil                              // убрали ссылку на screen, объект из
памяти выгрузился
```



Домашнее задание

1. Реализовать свой тип коллекции «очередь» (queue) с использованием дженериков.
2. Добавить ему несколько методов высшего порядка, полезных для этой коллекции (пример: filter для массивов)
3. *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.