Урок 5

ООП

Описываем требования к классам. Протоколы. Расширения. Полиморфизм. Композиция.

Протоколы

Мы уже отлично поработали с автомобилями. Но теперь у нас другая задача. На заводе выпускаются детали различной формы, и необходимо сделать конвейер, который бы оценивал периметр каждой из деталей. Фактически надо создать массив, в который мы положим фигуры и в цикле рассчитаем периметр для каждой из них.

Первая проблема, которая возникает на этом этапе, состоит в том, что периметр для разных фигур рассчитывается по-разному. И реализовывать в цикле логику расчета для каждого типа очень неудобно. Тем более, если мы начнем выпускать детали новой формы, нам придется добавлять новый алгоритм в цикл. Решается эта задача просто: мы создадим базовый класс фигуры и все фигуры будем наследовать от него. Определим в нем пустой метод подсчета периметра, а в наследниках просто переопределим его для расчета необходимых данных.

Вторая проблема состоит в том, что в массив нельзя класть разные типы объектов. Но наш план с базовым классом решает и ее. Если в качестве типа массива указать базовый класс, в него можно добавлять все объекты классов, унаследованных от него.

```
class Figure {                                  // базовый класс
    func calculatePerimiter() -> Double {     // расчет периметра
        return 0
    }
}
class Rectangle: Figure {
    var sideA: Double
    var sideB: Double
    override func calculatePerimiter() -> Double {
        return sideA + sideB
    }
    init(sideA: Double, sideB: Double) {
        self.sideA = sideA
        self.sideB = sideB
    }
}
class Circle: Figure {
    var radius: Double
    override func calculatePerimiter() -> Double {
        return 2.0 * M_PI * radius
    }
    init(radius: Double) {
        self.radius = radius
    }
}
var figures: [Figure] = [               // Можно создать массив базового класса
    Rectangle(sideA: 10.0, sideB: 12.0),  // и добавлять туда наследников
    Circle(radius: 18)                      // разных наследников
]
for figure in figures {
                                        // считаем периметр и выводим в консоль
    let perimiter = figure.calculatePerimiter()
    print(perimiter)
}
```

Все отлично, задача выполнена. Но у нас в приложении появился бесполезный класс «Figure». Его нигде нельзя использовать, объекты этого класса ничего не делают. Давайте избавимся от него, а на помощь нам придут протоколы.

```
protocol Figure {                                // Протокол
    func calculatePerimiter() -> Double      // просто описание метода
}
class Rectangle: Figure {                    // имплементируем протокол
    var sideA: Double
    var sideB: Double
    func calculatePerimiter() -> Double { // реализуем свойство, override не
нужен
        return sideA + sideB
    }
    init(sideA: Double, sideB: Double) {
        self.sideA = sideA
        self.sideB = sideB
    }
}
class Circle: Figure {
    var radius: Double

    func calculatePerimiter() -> Double {
        return 2.0 * M_PI * radius
    }
    init(radius: Double) {
        self.radius = radius
    }
}
var figures: [Figure] = [                // Можно создать массив типа протокола
    Rectangle(sideA: 10.0, sideB: 12.0),// и добавлять туда классы,
имплементирующие протокол
    Circle(radius: 18)
]
for figure in figures {
                                        // считаем периметр и выводим в консоль
    let perimiter = figure.calculatePerimiter()
    print(perimiter)
}
```

Протокол, в отличие от класса, не содержит каких-либо действий методов, он не может совершать никакой работы. Нельзя создать объект типа протокола. Это просто описание свойств и методов без их реализации.

Сила протоколов в том, что они объявляют стандарт. Любой ваш класс, структура или перечисление может принять этот стандарт и реализовать все, что описано в протоколе. Кроме того, вы можете использовать протокол для указания типа переменных или массивов. Это позволяет присваивать им значения любых типов, лишь бы эти типы имплементировали протокол.

Важно понимать, что протоколы не надо использовать для описания целых классов, в них следует описывать лишь какое-то конкретное поведение. Например, протокол расчета периметра никак не

привязан к фигурам, и если у нас появится некий класс «House», мы можем имплементировать ему наш протокол для расчета его периметра. Так как протокол не имеет отношения к фигуре, давайте его переименуем. Он определяет поведение расчета периметра, и подходящим именем будет «perimterCalculatable». Теперь, если если какой-то класс имплементирует этот протокол, будет сразу понятно, что у него можно рассчитать периметр.

```
protocol PerimterCalculatable {           // Протокол
    func calculatePerimiter() -> Double  // просто описание метода
}
class House: PerimterCalculatable {     // у дома можно посчитать периметр
    var sideA: Double?
    var sideB: Double?
    func calculatePerimiter() -> Double {
        return sideA! + sideB!
    }
}
class Circle: PerimterCalculatable {   // у круга можно посчитать периметр
    var radius: Double!

    func calculatePerimiter() -> Double {
        return 2.0 * Double.pi * radius
    }
}
```

Расширения

В отличие от классов и протоколов, расширение не является отдельной сущностью. Его нельзя создать, а потом применить. Но тем не менее, расширения очень полезны: они позволяют расширять функционал уже имеющихся классов без их изменения. Это означает, что мы можем взять абсолютно любой класс в приложении и добавить ему любые методы. Это работает даже с классами, которые определены в библиотеках и изменить их определение нет возможности.

Например, мы бы хотели добавить расчет диаметра кругу, не меняя сам класс. Нам достаточно написать для него расширение.

```
extension Circle {                    // используем ключевое слово extension
    func perimeter() -> Double {     // и можем добавлять новые методы
        return radius * 2.0
    }
}
let circle = Circle(radius: 12)
circle.perimeter()                   // 24
```

Расширения могут добавлять только новое поведение. Проще говоря, они могут добавлять методы, конструкторы (только вспомогательные), но не свойства. Правда, вычисляемые свойства все же являются поведением, так как не хранят значений, а только проводят вычисления. Так как метод расчета периметра больше похож на свойство, давайте его перепишем.

```
extension Circle {             // используем ключевое слово extension
    var perimeter: Double {    // теперь периметр - вычисляемое свойство
        return radius * 2.0
```

```
    }
}
let circle = Circle(radius: 12)
circle.perimeter             // 24
```

На самом деле в примере выше не было особого смысла выносить расчет периметра в расширение, мы могли поменять и класс. Но представьте, что нам необходим метод у типа «String», который убирал бы из строки все четные символы, но класс объявлен в стандартной библиотеке и изменить его мы не можем, а вот расширить – легко.

```
extension String {
    func oddChars() -> String {
        var oddChars: [Character] = []
        for char in self.characters.enumerated() {
            if char.offset % 2 == 0 {
                oddChars.append(char.element)
            }
        }
        return String(oddChars)
    }
}
let newString = "qwertyy".oddChars()
print(newString) // qety
```

Есть еще одно необязательное, но желательное применение расширений. Если вы имплементируете классу протокол, то лучше сделать это в расширении. Так все методы протокола будут сгруппированы в одном месте. Давайте перепишем класс фигуры этим способом.

```
class Circle {
    var radius: Double!
}
extension Circle: PerimterCalculatable {    // теперь вся информация о
имплементации протокола сгруппирована в одном месте
    func calculatePerimiter() -> Double {
        return 2.0 * Double.pi * radius
    }
}
```

И есть еще одно невероятно полезное свойство расширений. Вы можете расширить протокол, добавив в него реализацию по умолчанию.

К нашему методу расчета периметра сложно написать реализацию по умолчанию, так как она отличается в зависимости от фигуры. Но, например, мы можем добавить метод вывода в консоль описания объекта. На самом деле есть стандартный протокол, который делает почти то, что нам нужно, – «CustomStringConvertible». Он определяет вычисляемое свойство description. Именно его значение выводится в консоль, когда мы делаем «print(object)». Давайте возьмем этот протокол и напишем его наследника, чтобы сразу выводить информацию в консоль.

```
protocol ConsolePrintable: CustomStringConvertible{ // наш протокол наследует
другой протокол
    func printDescription()
```

```
}

extension ConsolePrintable{                          // расширяем протокол
    func printDescription() {                        // реализуем метод для вывода
в консоль поля description
        print(description)                           // само поле description
определено в CustomStringConvertible
    }
}
class Circle: PerimterCalculatable, ConsolePrintable {
    var radius: Double
    func calculatePerimiter() -> Double {
        return 2.0 * Double.pi * radius
    }
    init(radius: Double) {
        self.radius = radius
    }
    // так как протокол унаследован от CustomStringConvertible, мы должны
реализовать и его требования
    var description: String {
        return String(describing: radius)
    }
}
let circle = Circle(radius: 30)
circle.printDescription()
// выведет в консоль 30
```

Теперь мы можем имплементировать протокол «ConsolePrintable» любому объекту и сразу вызывать у него метод «printDescription». Правда, нам придется реализовать свойство «description», которое описано в протоколе «CustomStringConvertible».

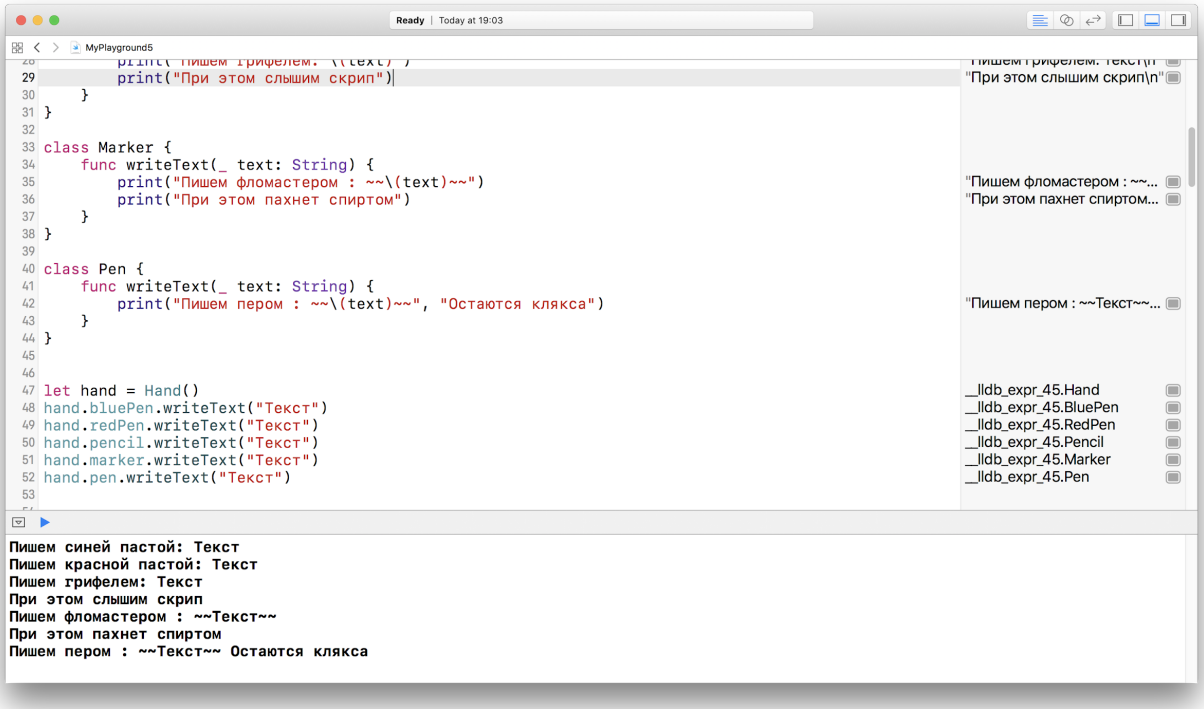
Обратите внимание: во-первых, мы унаследовали один протокол от другого, а во-вторых, имплементировали классу «Circle» сразу два протокола. Дело в том, что класс может наследоваться только от одного класса, но протоколов может имплементировать сколько угодно. И даже больше – один протокол может наследоваться от любого количества других протоколов. Проще говоря, для протоколов поддерживается множественное наследование.

Полиморфизм

Давайте напишем забавную программу. В ней будет рука, которая может держать красную ручку, синюю ручку, маркер, карандаш или перо, а также писать тем, что держит.

```
class Hand{
    let redPen = RedPen()
    let bluePen = BluePen()
    let pencil = Pencil()
    let marker = Marker()
    let pen = Pen()
}
class RedPen {
    func writeText(_ text: String) {
```

```
        print("Пишем красной пастой: \(text)")
    }
}
class BluePen {
    func writeText(_ text: String) {
        print("Пишем синей пастой: \(text)")
    }
}
class Pencil {
    func writeText(_ text: String) {
        print("Пишем грифелем: \(text)")
        print("При этом слышим скрип")
    }
}
class Marker {
    func writeText(_ text: String) {
        print("Пишем фломастером : ~~\(text)~~")
        print("При этом пахнет спиртом")
    }
}
class Pen {
    func writeText(_ text: String) {
        print("Пишем пером : ~~\(text)~~", "Остаются кляксы")
    }
}
let hand = Hand()
hand.bluePen.writeText("Текст")
hand.redPen.writeText("Текст")
hand.pencil.writeText("Текст")
hand.marker.writeText("Текст")
hand.pen.writeText("Текст")
```



Это оказалось не так сложно, все работает. Жалко, нельзя форматировать текст в консоли, чтобы изменить его цвет, но давайте проявим фантазию. И все же эта программа не очень хорошая. Во-первых, в руке сразу 5 предметов для письма, что не совсем естественно. Во-вторых, выходит, что рука не в состоянии писать, например, зеленой ручкой или красным фломастером, хотя работают они все по одному принципу. Давайте добавим в нашу программу немного полиморфизма.

В отличие от всего, что мы проходили до этого, полиморфизм не является какой-либо технологией или оператором языка – это принцип объектно-ориентированного программирования.

Этот термин не имеет конкретного определения, но его можно описать как независимость методов или свойств от конкретной реализации, а только от интерфейса. Интерфейсом называют описание объекта без его реализации. Я думаю, вы уже поняли, что в Swift в роли интерфейса выступают протоколы.

Давайте перепишем программу. Сделаем так, чтобы в руку можно было взять любую принадлежность для письма и вывести в консоль сообщение. Принадлежность для письма будет нашим интерфейсом и опишет только базовые возможности конкретных объектов, которые мы сможем взять в руку. Руке не обязательно знать о конкретных классах, ей не важно, писать ли ручкой, карандашом или фломастером, но результат будет отличаться.

```
protocol LetterBelonging{                       // объявляем протокол
принадлежности для письма
    func writeText(_ text: String)
}
class Hand{
    var letterBelonging:LetterBelonging?   // объявляем свойство руки держать
принадлежность
    func write(_ text: String) {           // говорим руке писать
        letterBelonging?.writeText(text) // и она уже пишет принадлежностью
    }
}
class RedPen: LetterBelonging {
    func writeText(_ text: String) {
        print("Пишем красной пастой: \(text)")
    }
}
class BluePen: LetterBelonging {
    func writeText(_ text: String) {
        print("Пишем синей пастой: \(text)")
    }
}
class Pencil: LetterBelonging {
    func writeText(_ text: String) {
        print("Пишем грифелем: \(text)")
        print("При этом слышим скрип")
    }
}
class Marker: LetterBelonging {
    func writeText(_ text: String) {
        print("Пишем фломастером : ~~\(text)~~")
        print("При этом пахнет спиртом")
    }
}
class Pen: LetterBelonging {
    func writeText(_ text: String) {
        print("Пишем пером : ~~\(text)~~", "Остаются кляксы")
    }
}
let hand = Hand()
hand.letterBelonging = RedPen()
hand.write("Привет")
hand.letterBelonging = BluePen()
hand.write("Мир")
hand.letterBelonging = Pencil()
hand.write("Я написал")
hand.letterBelonging = Marker()
hand.write("первую программу")
hand.letterBelonging = Pen()
hand.write("с полиморфизмом!")
```

Обратите внимание: теперь рука абсолютно не зависит от того, какой класс она держит. Единственное, что она знает, – он поддерживает интерфейс пишущей принадлежности. Всего у нас 5 разных пишущих принадлежностей, и каждая из них работает по-разному. Когда мы берем в руку маркер, вывод в консоли отличается от того, что выводила в консоль ручка. Если потребуется добавить еще несколько принадлежностей для письма, мы легко это сделаем, даже не придется менять класс «Hand».

Следуя принципам полиморфизма, можно сделать программу более гибкой и простой.

Композиция

На прошлом занятии мы говорили о наследовании как о способе сделать программу проще, писать меньше кода и расширять возможности классов. Наследование – действительно потрясающая особенность ООП, но у нее есть ряд минусов.

Если вы разработали большую программу со сложной иерархией наследования, а через какое-то время поняли, что класс, находящийся в цепочке наследования, должен иметь какие-то особые свойства, которых не должно быть у его наследников, вам придется попотеть, чтобы спроектировать и реализовать новую иерархию.

Иногда наследование не может решить проблемы с дублированием или избыточностью кода. Например, мы пишем игру и у нас есть три класса:

- База со свойством «health».
- Башня с методом «shoot».
- Враг с методом «move».

```
class Base {
    var health = 0
}
class Tower {
    func shoot() {
    }
}
class Enemy  {
    func move() {
    }
}
```

И вот нам понадобилось добавить в игру нового врага «shoot enemy» с методами «move» и «shoot». От какого класса его наследовать? Если мы наследуем его от башни или от обычного врага, получим либо метод «shoot», либо «move». Второй метод нам придется копировать.

```
class ShootEnemy: Tower {            // если мы наследуемся от башни
    func move() {                    // надо скопировать метод move от врага
    }
}
```

Мы, конечно, можем поступить наоборот, реализовать в нем оба метода, сделать башню и врага его наследниками. Но, во-первых, будет нарушена обычная логика: почему дружественная башня – наследник стреляющего врага? Во-вторых, теперь у башни и врага имеется лишний метод. Башня не может двигаться, но унаследовала метод «move». Обычный враг не может стрелять, но унаследовал

метод «shoot». Мы можем просто пообещать себе никогда не пользоваться этими лишними методами, но это обещание будет сложно сдержать.

```
class Base {
    var health = 0
}
class ShootEnemy {
    func move() {
    }
    func shoot() {
    }
}
class Tower: ShootEnemy { // теперь наш башня умеет двигаться
}
class Enemy: ShootEnemy {// а наш обычный враг стрелять
}
```

Чем больше будет программа, тем больше будет таких компромиссов, и вы либо будете дублировать код, либо классы будут обрастать лишними свойствами и методами. Есть много различных техник, которые могут помочь избежать этой проблемы, и одна из них – композиция.

Давайте переделаем нашу программу. Вместо наследования будем использовать композицию. Создадим три протокола «Healthable», «Movable», «Shootable», добавим им реализации по умолчанию. После этого нужным классам имплементируем протоколы с необходимым поведением. Используя такой подход, вы сможете собирать классы по кусочкам, как из блоков конструктора.

```
protocol Healthable {                    // Протокол, добавляющий жизни
    var health: Int { get set }
}
protocol Movable {                       // Протокол передвижения
    func move()
}
extension Movable {                      // Реализация передвижения по умолчанию
    func move(){}
}
protocol Shootable {                     // Протокол стрельбы
    func shoot()
}
extension Shootable{                     // Реализация стрельбы по умолчанию
    func shoot() {}
}
class Base: Healthable {                 // База теперь имеет жизни
    var health = 0
}
class Tower: Shootable {                 // Башня может стрелять
}
class Enemy: Movable {                   // Враг – двигаться
}
class ShootEnemy: Shootable, Movable { // Стреляющий враг может стрелять и
двигаться
}
```

Стандартные протоколы

В заключение занятия следует сказать, что в последнее время Apple старается использовать композицию везде, где только можно. Почти все стандартные классы в Swift собраны из протоколов. Это называют «protocol oriented programming», что, по сути, является синонимом композиции.

Более того, разработчикам тоже предлагается активно использовать стандартные протоколы. Например, вы создали свой класс «прямоугольник». По умолчанию ни один стандартный оператор не будет работать с этим классом, потому что не знает, как. Вы не сможете сделать вывод этого класса в консоль, потому что команда «print» не знает, как его выводить. Чтобы наделить его стандартными возможностями, вам придется имплементировать различные стандартные протоколы. Давайте имплементируем нашему прямоугольнику несколько протоколов:

- «CustomStringConvertible» – определяет, как ваш класс выводится командой «print».
- «Comparable» – определяет правила сравнения для вашего объекта.
- «Hashable» – правила хэширования объекта. После его имплементации объект может помещаться в «set» и использоваться в качестве ключа в словаре.

```
class Rectangle {
    var sideA: Double
    var sideB: Double
    func calculatePerimiter() -> Double {
        return sideA * 2 + sideB * 2
    }
    var square: Double {
        return sideA * sideB
    }
    init(sideA: Double, sideB: Double) {
        self.sideA = sideA
        self.sideB = sideB
    }
}
extension Rectangle: CustomStringConvertible{  // имплементируем протокол для
вывода в консоль
    var description: String{
        return "sideA: \(sideA), \(sideB)"    // определяем, что именно будет
выводиться
    }
}
extension Rectangle: Comparable {     // имплементируем протокол для сравнения
// в рамках протокола перегружаем оператор сравнения для нашего прямоугольника
    static func <(lhs: Rectangle, rhs: Rectangle) -> Bool{
        return lhs.square < rhs.square
    }
    static func <=(lhs: Rectangle, rhs: Rectangle) -> Bool{
        return lhs.square <= rhs.square
    }
    static func >=(lhs: Rectangle, rhs: Rectangle) -> Bool{
        return lhs.square >= rhs.square
    }
    static func >(lhs: Rectangle, rhs: Rectangle) -> Bool{
        return lhs.square > rhs.square
    }
    static func ==(lhs: Rectangle, rhs: Rectangle) -> Bool{
        return lhs.square == rhs.square
    }
}
extension Rectangle: Hashable {     // имплементируем протокол для хэширования
    var hashValue: Int{            // определяем, как рассчитывать хэш
        let hashBase = sideA + sideB + square
        return hashBase.hashValue
    }
}
let smallRectangle = Rectangle(sideA: 2, sideB: 4)
let bigRectangle = Rectangle(sideA: 20, sideB: 40)
smallRectangle < bigRectangle   // true
smallRectangle == bigRectangle // false
let dictionary = [
    smallRectangle: 7,
    bigRectangle: 90
]
```

Домашнее задание

1. Создать протокол «Car» и описать свойства, общие для автомобилей, а также метод действия.
2. Создать расширения для протокола «Car» и реализовать в них методы конкретных действий с автомобилем: открыть/закрыть окно, запустить/заглушить двигатель и т.д. (по одному методу на действие, реализовывать следует только те действия, реализация которых общая для всех автомобилей).
3. Создать два класса, имплементирующих протокол «Car»: tunkCar и sportCar. Описать в них свойства, отличающиеся для спортивного автомобиля и цистерны.
4. Для каждого класса написать расширение, имплементирующее протокол «CustomStringConvertible».
5. Создать несколько объектов каждого класса. Применить к ним различные действия.
6. Вывести сами объекты в консоль.

Дополнительные материалы

1. Официальная документация

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

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