



Спринт 2, тема «Классы»

Яндекс Практикум

Мы сделали для вас памятку по теме «Классы». Здесь вы найдёте короткое изложение пройденного в уроке материала и ключевые фрагменты кода. Используйте шпаргалку, чтобы быстро восстановить в памяти пройденный материал.

 Скачайте документ, чтобы обращаться к нему при необходимости, пока не доведёте навыки до автоматизма.

Классы

 **Классы** в Swift помогают создавать объекты, которые содержат свойства и методы, а также наследуются от других классов. Классы передаются по ссылке. Подробнее об этом — ниже.

Классы объявляют с помощью ключевого слова `class`.

Для классов Swift не генерирует инициализатор по умолчанию, поэтому нужно явным образом определять `init`, который всем не опциональным свойствам (при необходимости им тоже) присвоит начальные значения.

```
class Vehicle {
    var numberOfWheels = 0


    func drive() {
        // Код для движения транспортного средства
    }

    init(numberOfWheels: Int) {
        self.numberOfWheels = numberOfWheels
    }
}
```

Создать экземпляр класса можно так:

```
let monowheel = Vehicle(numberOfWheels: 1)
```

Наследование

 **Наследование** — это механизм объектно-ориентированного программирования, который позволяет создавать новый класс на основе уже существующего класса (родительского). При этом новый класс наследует свойства и методы родительского, а также может иметь собственные свойства и методы.

Класс-наследник обычно называют дочерним классом, иногда сыновым. В более формальной литературе родительский класс называют супер-классом, а дочерний — под-классом.

В Swift наследование реализуется с помощью ключевого слова `class`, за которым следует имя класса, после него — имя родительского класса.

```
class Car: Vehicle {
    var color = ""

    func honk() {
        // Код для звукового сигнала автомобиля
    }
}
```

В примере `Vehicle` — родительский класс, `Car` — дочерний. Класс `Car`:

- наследует свойство `numberOfWheels` и метод `drive()` от `Vehicle`.
- добавляет свойство `color` и метод `honk()`.

Объекты `Car` могут вызывать как методы из `Vehicle`, так и собственные методы.

В Swift можно создавать цепочки наследования, когда одни классы наследуются от других классов, которые в свою очередь наследуются от третьих и т.д. В некоторых ситуациях это полезно, но иногда чрезмерное использование наследования приводит к проблемам с проектированием. Поэтому механизм наследования стоит использовать осознанно.

Инициализаторы и наследование

Если дочерний класс объявляет собственные свойства, то при их инициализации необходимо вызвать инициализатор родительского класса. Так мы правильно инициализируем все свойства в цепочке наследования.

Для этого в Swift используют ключевое слово `super`, которое обращается к родительскому классу. В методе инициализации дочернего класса нужно вызвать инициализатор родительского класса, используя синтаксис `super.init()`, передавая ему все необходимые параметры (см. пример ниже).

Например, есть класс `Vehicle` со свойством `numberOfWheels` и дочерний класс `Car` с собственным свойством `color`. Для инициализации свойств `color` и `numberOfWheels` в классе `Car` нужно вызвать инициализатор родительского класса с помощью `super.init()`.

Например, так:

```
class Vehicle {
    var numberOfWheels: Int

    init(numberOfWheels: Int) {
        self.numberOfWheels = numberOfWheels
    }
}

class Car: Vehicle {
    var color: String

    init(numberOfWheels: Int, color: String) {
        self.color = color
        super.init(numberOfWheels: numberOfWheels)
    }
}
```

В примере:

- Класс `Vehicle` имеет инициализатор, который принимает один параметр `numberOfWheels`.
- Класс `Car` имеет свойство `color` и инициализатор, который принимает два параметра: `numberOfWheels` и `color`.


- Внутри инициализатора `Car` мы инициализируем свойство `color`, а затем вызываем инициализатор родительского класса, передавая ему параметр `numberOfWheels`.

В итоге оба свойства `numberOfWheels` и `color` будут правильно инициализированы при создании объекта класса `Car`.

Если не вызывать `super.init` в дочернем классе, то получим ошибку:

```
'super.init' isn't called on all paths before returning from initializer
```

Полиморфизм

 **Полиморфизм** — это способность объекта использовать методы родительского класса или интерфейса, а также переопределённые методы в дочерних классах.

Чтобы использовать полиморфизм в Swift, нужно объявить переменную родительского класса, а затем инициализировать её объектом любого дочернего класса. Теперь через переменную родительского класса можно вызывать как методы родительского класса, так и переопределённые методы дочернего класса.

Рассмотрим пример. Есть класс `Vehicle` и его дочерний класс `Car`, который переопределяет метод `drive()`:

```
class Vehicle {
    func drive() {
        print("Vehicle is driving")
    }
}

class Car: Vehicle {
    override func drive() {
        print("Car is driving")
    }
}
```

Теперь можно создать объект класса `Car` и использовать его через переменную родительского класса `Vehicle`:

```
let vehicle: Vehicle = Car()
vehicle.drive() // "Car is driving"
```

В примере мы:

1. Создаём объект `Car` и присваиваем его переменной `vehicle` типа `Vehicle`.
2. Вызываем метод `drive()` через переменную `vehicle`. Эта переменная была объявлена как `Vehicle`, но она содержит объект `Car`. Поэтому при вызове метода `drive()` получим вывод `"Car is driving"`. Это и есть полиморфизм.

Полиморфизм позволяет использовать объекты дочерних классов вместо объектов родительского класса и вызывать их переопределённые методы. Это расширяет функциональность приложения и упрощает его проектирование.

Передача аргументов по ссылке и по значению

В Swift есть два способа передать параметры и объекты в функции: по значению и по ссылке.

Когда объект передаётся **по значению** (англ. "value type" — тип значение), то каждый раз создаётся его копия. Если в функции объект меняется, то меняется его копия, а сам он остаётся тем же и вне функции эти изменения не видны. По значению в Swift передаются структуры и перечисления.

Например:

```
struct Point {
    var x: Int
    var y: Int
}

func modifyPoint(point: Point) {
    var newPoint = point
    newPoint.x += 1
    newPoint.y += 1
    print("New point: (\\(newPoint.x), \\(newPoint.y))")
}

var myPoint = Point(x: 0, y: 0)
modifyPoint(point: myPoint)
```

```
print("Original point: (\\(myPoint.x), \\(myPoint.y))")
// Напечатает: Original point: (0, 0)
```

В примере мы:

1. Создаём структуру `Point` со свойствами `x` и `y`.
2. Объявляем функцию `modifyPoint`, которая принимает копию структуры `Point`, изменяет её значения и печатает новую точку.
3. Вызываем функцию `modifyPoint` и передаём ей нашу структуру `myPoint`.
4. Печатаем значения свойств нашей структуры.

В результате видно, что исходная структура не изменилась, так как её передавали по значению.

Когда объект передаётся **по ссылке** (англ. "reference type" — ссылочный тип), то он существует в единственном экземпляре и передаётся ссылка на него — адрес места в памяти, где он «живёт». Если в функции объект меняется, то меняется он сам, и изменения видны за пределами функции. По ссылке в Swift передаются классы.

Например:

```
class Person {
    var name: String

    init(name: String) {
        self.name = name
    }
}

func changeName(person: Person) {
    person.name = "John"
}

var myPerson = Person(name: "Jane")
changeName(person: myPerson)
print("New name: \\(myPerson.name)")
```

В этом примере мы:

1. Создаём класс `Person` со свойством `name`,

2. Создаём функцию `changeName`, которая принимает объект класса `Person` и меняет его свойство `name`.
3. Создаём объект `myPerson` класса `Person` с именем `"Jane"`.
4. Передаём объект `myPerson` в функцию `changeName`.
5. Печатаем свойство `name`.

В результате видно, что свойство `name` изменилось, так как объект класса передали по ссылке.

Классы и структуры

Класс и структура похожи: у обоих есть свойства и методы.

Основные отличия между этими объектами представлены в таблице:

	Класс	Структура
Как передаётся	По ссылке	По значению
Возможность наследования	Есть	Нет
Инициализатор по умолчанию	Не генерируется	Генерируется

Особенности объектов помогают выбрать, какой из них использовать:

- **Структуры** используют, когда нужно подойдёт передача по значению. Например, если нужно создать простой объект, который не будет меняться после инициализации. С помощью структур реализуют легковесные типы данных, например, координаты, размеры, цвета.
- **Классы** выбирают, когда важна передача по ссылке. Например, когда создают объект, который будет иметь сложную структуру или меняться во времени. Передача по ссылке помогает работать с большими объёмами данных. Копирование больших объектов занимает много времени и памяти, а передача по ссылке экономит ресурсы. С помощью классов создают модели данных, UI-компоненты.

Яндекс Практикум

