



# Протоколы, расширения, универсальные шаблоны

Основы языка Swift



# Оглавление

<b>Введение</b>	<b>3</b>
<b>Термины, используемые в лекции</b>	<b>4</b>
<b>Протоколы</b>	<b>4</b>
Объявление методов в протоколе	6
Объявление переменных в протоколе	9
Объявление инициализатора в протоколе	11
<b>Расширения</b>	<b>11</b>
Объявление свойств в расширении	12
Объявление вложенных типов в расширении	13
Дефолтная реализация протокола	13
<b>ООП</b>	<b>14</b>
Наследование	14
override	16
final	17
Полиморфизм	17
Инкапсуляция	18
<b>Дженерики</b>	<b>19</b>
Универсальные типы	21
Ограничения универсальных типов	22
<b>associatedtype</b>	<b>23</b>
<b>Что можно почитать еще?</b>	<b>25</b>
<b>Используемая литература</b>	<b>26</b>

# Введение

На прошлой лекции мы узнали про классы и структуры, узнали как объявить в них методы и свойства. В ходе этой лекции вы узнаете про протоколы, расширение, наследование, полиморфизм, инкапсуляцию.

На этой лекции вы узнаете:

- Что такое протокол
- Что такое расширение
- Про наследование инкапсуляцию и полиморфизм в Swift
- Что такое универсальные шаблоны

# Термины, используемые в лекции

**Наследование** - это один из принципов ООП, в ходе которого один класс может наследоваться от другого.

**Полиморфизм** - один из принципов ООП, который предоставляет возможность взаимозаменяемости типов, которые находятся в одной иерархии.

**Инкапсуляция** - один из принципов ООП, который подразумевает сокрытие методов и свойств, которые не нужны при использовании данного класса.

# Протоколы

В ходе прошлой лекции мы создали класс с кафе

```
class Cafe {
    private var coffee: [Coffee]

    init(coffee: [Coffee]) {
        self.coffee = coffee
    }

    convenience init() {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        self.init(coffee: [latte, cappuccino])
    }

    func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}
```

Теперь же давайте откроем целую сеть, а значит создадим сразу несколько классов.

```
class FirstCafe {
    private var coffee: [Coffee]

    init(coffee: [Coffee]) {
        self.coffee = coffee
    }

    convenience init() {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        self.init(coffee: [latte, cappuccino])
    }

    func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}
```

```

class SecondCafe {
    private var coffee: [Coffee]

    init(coffee: [Coffee]) {
        self.coffee = coffee
    }

    convenience init() {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        self.init(coffee: [latte, cappuccino])
    }

    func addNewCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}

class ThirdCafe {
    private var coffee: [Coffee]

    init(coffee: [Coffee]) {
        self.coffee = coffee
    }

    convenience init() {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        self.init(coffee: [latte, cappuccino])
    }

    func add(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}

```

Теперь у нас сразу несколько кафе, попытаемся добавить им новый напиток и окажется, что у каждого из них свой метод для добавления. У FirstCafe это addCoffee, у SecondCafe это addNewCoffee, а у ThirdCafe это add

```

let espresso = Coffee(name: "Espresso", isSugar: true, isIce:

```

```

false, cost: 50, size: .1)
let firstCafe = FirstCafe()
let secondCafe = SecondCafe()
let thirdCafe = ThirdCafe()

firstCafe.addCoffee(coffee: espresso)
secondCafe.addNewCoffee(coffee: espresso)
thirdCafe.add(coffee: espresso)

```

А если будут еще кафе? Тогда получится у каждого будет свой метод и это становится все менее удобно, к тому же это все кафе одной сети. В этом случае на помощь приходят протоколы. Протокол определяет методы, свойства и другие требования, которые соответствуют определенной функциональности. Структуры, классы и перечисления могут реализовать этот протокол, то есть удовлетворять его требованиям.

Протокол имеет следующий синтаксис:

```

protocol <Имя протокола> {
    // функциональность, которую необходимо реализовать в
    // перечислении, структуре или классе
}

```

В протоколе только описывается функциональность, но не реализовывается.

## Объявление методов в протоколе

Теперь же создадим протокол для Cafe и добавим метод по добавлению кофе, а именно addCoffee. Для этого указываем функцию, а именно ее название и параметры, которые она принимает. Если бы функция еще и возвращала какое-либо значение, то его тоже нужно было бы указать. Также обратите внимание, что после функции нет фигурных скобок, так как реализована она будет только в классе, структуре или перечислении.

```

protocol CafeProtocol {
    func addCoffee(coffee: Coffee)
}

```

Затем нужно указать, что класс принимает протокол. Для этого после имени протокола указывается двоеточие и имя протокола.

```

class/struct/enum <Имя>: <Имя протокола> {}

```

Укажем, что FirstCoffee принимает протокол

```

class FirstCafe: CafeProtocol {
    private var coffee: [Coffee]
}

```

```

    init(coffee: [Coffee]) {
        self.coffee = coffee
    }

    convenience init() {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        self.init(coffee: [latte, cappuccino])
    }

    func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}

```


В FirstCafe уже реализована функция addCoffee, в ней к массиву кофе передается новый напиток. Если подписать на протокол SecondCafe, то выскочит ошибка, что класс не соответствует протоколу.

```

class SecondCafe: CafeProtocol {
    private var coffee: [Coffee]

    init(coffee: [Coffee]) {
        self.coffee = coffee
    }
}

```

 Type 'SecondCafe' does not conform to protocol 'CafeProtocol'  
 Do you want to add protocol stubs?

Fix

Чтобы поправить ошибку удалим функцию addNewCoffe и добавим функцию addCoffee.

```

class SecondCafe: CafeProtocol {
    private var coffee: [Coffee]

    init(coffee: [Coffee]) {
        self.coffee = coffee
    }

    convenience init() {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        self.init(coffee: [latte, cappuccino])
    }

    func addCoffee(coffee: Coffee) {

```



```

        self.coffee.append(coffee)
    }
}

```

Функция `addCoffee` в `Second` не обязательно должна повторять код `addCoffee` в `SecondCafe`, например, пусть функция `addCoffee` не только добавляет новый кофе в массив, но и печатает количество напитков в массиве.

```

func addCoffee(coffee: Coffee) {
    self.coffee.append(coffee)
    print(self.coffee.count)
}

```

Теперь подпишем `ThirdCafe` на этот протокол

```

class ThirdCafe: CafeProtocol {
    private var coffee: [Coffee]

    init(coffee: [Coffee]) {
        self.coffee = coffee
    }

    convenience init() {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        self.init(coffee: [latte, cappuccino])
    }

    func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}

```

Все три кафе реализуют необходимый протокол, а значит мы можем использовать одну и ту же функцию для добавления кофе

```

firstCafe.addCoffee(coffee: espresso)
secondCafe.addCoffee(coffee: espresso)
thirdCafe.addCoffee(coffee: espresso)

```

Чтобы все кафе были объединены соберем их в один массив объектов. Но не просто объектов, а только тех, которые соответствуют протоколу `CafeProtocol`. Для этого при объявлении типа массива укажем в квадратных скобках `CafeProtocol`.

```

var cafe: [CafeProtocol] = [firstCafe, secondCafe, thirdCafe]

```



В такой массив нельзя добавить объект, который не соответствует протоколу `CafeProtocol`

Объединение в массив позволяет нам не писать каждый раз `cafe.addCoffee` для каждого кафе, а добавить напиток сразу всем кафе в массиве

```
for oneCafe in cafe {  
    oneCafe.addCoffee(coffee: espresso)  
}
```

В данном случае будет вызван метод для каждого из классов. То есть для первого кафе, а именно `FirstCoffee` будет вызвана функция, которая реализована именно в этом классе, для второго(`SecondCafe`) та, что реализована там, то есть не только будет добавлен новый напиток, но и напечатано количество кофе в массиве.

## Объявление переменных в протоколе

Выше мы рассмотрели как объявить метод в протоколе, теперь рассмотрим, как добавить переменную. Добавим переменную `coffee` в протокол `CafeProtocol`, чтобы указать, что все кафе должны содержать массив `Coffee`.

Чтобы добавить в протокол переменную необходимо указать ключевое слово `var`

```
var <Имя переменной>: <Тип переменной> { get set }
```



Ключевое слово должно быть именно `var`, если указать `let` будет ошибка

После типа переменной указываются фигурные скобки, в которых указывается либо `get set`, либо только `get`. Чем же они отличаются?

Когда мы указываем `get set`, значит мы можем и получить переменную вне класса и изменить. А вот если указать только `get`, то мы можем получить переменную вне класса, а изменить только внутри класса.

Давайте рассмотрим на примере `coffee` в `CafeProtocol`. Сначала укажем `get set`.

```
protocol CafeProtocol {  
    var coffee: [Coffee] { get set }  
    func addCoffee(coffee: Coffee)  
}
```

```
class FirstCafe: CafeProtocol {  
    var coffee: [Coffee]  
  
    init(coffee: [Coffee]) {  
        self.coffee = coffee  
    }  
}
```

```

    }

    convenience init() {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        self.init(coffee: [latte, cappuccino])
    }

    func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}

```

Создадим экземпляр класса FirstCafe и посмотрим, что мы можем сделать с переменной coffee.

```

var firstCafe: CafeProtocol = FirstCafe()
firstCafe.coffee = [espresso]
print(firstCafe.coffee)

```

То есть мы можем и изменить coffee и напечатать. Теперь изменим get set только на get.

```

protocol CafeProtocol {
    var coffee: [Coffee] { get }
    func addCoffee(coffee: Coffee)
}

```

Тут же выскочит ошибка, ведь теперь coffee read only свойство, а значит его нельзя изменять извне классе.

```

var firstCafe: CafeProtocol = FirstCafe()
firstCafe.coffee = [espresso]
print(firstCafe.coffee)

```

✖ Cannot assign to property: 'coffee' is a get-only property ✖

## Объявление инициализатора в протоколе

Протокол может требовать не только методы и свойства, но и инициализаторы. Для этого необходимо указать ключевое слово `init` и если требуется, параметры, которые просит данный инициализатор. Такой инициализатор будет обязательным для реализации, поэтому помечается словом `required`.

Добавим инициализатор в CafeProtocol.

```
protocol CafeProtocol {
    var coffee: [Coffee] { get }
    func addCoffee(coffee: Coffee)

    init(coffee: [Coffee])
}
```

Теперь, хоть FirstCafe имеет такой init, как в протоколе, все равно выскочит ошибка

```
init(coffee: [Coffee]) {
    self.coffee = coffee
}

convenience init() {
    let latte = Coffee("latte")
}
```

Initializer requirement 'init(coffee:)' can only be satisfied by a 'required' initializer in non-final class 'FirstCafe'

Insert 'required '

Fix

Чтобы поправить эту ошибку достаточно указать перед init ключевое слово required

```
required init(coffee: [Coffee]) {
    self.coffee = coffee
}
```

## Расширения

Давайте добавим CakeProtocol, в нем будет указан метод addCake.

```
struct Cake {
    var name: String
    var cost: Double
}

protocol CakeProtocol {
    func addCake(cake: Cake)
}
```

И подпишем на этот протокол FirstCafe. Для этого можно указать через запятую после CafeProtocol CakeProtocol.

```
class FirstCafe: CafeProtocol, CakeProtocol
```

А теперь представим, что таких протоколов будет все больше и больше, количество протоколов через запятую будет расти и расти, класс будет все больше, что снижает читаемость кода.

Здесь на помощь придут расширения. Расширения (или extensions) добавляют новую функциональность для структуры, перечисления или класса и имеет ключевое слово extension. Расширения имеют следующую конструкцию:

```
extension <Имя класса, структуры или перечисления, для которого
необходимо расширение> {
    // код расширения
}
```

Вынесем соответствие класса FirstCafe протоколу CakeProtocol в расширение

```
extension FirstCafe: CakeProtocol {
    func addCake(cake: Cake) {
        print(cake)
    }
}
```

Теперь код для реализации CakeProtocol вынесен и читаемость кода повышена.

## Объявление свойств в расширении

В расширении можно указывать не только код для реализации протокола, но и переменные.

```
extension FirstCafe {
    var cafeName: String {
        "Cafe Name"
    }
}
```

Но стоит помнить, что в расширении не может быть stored property, только computed. Если попытаться добавить свойство хранения в расширение - будет ошибка

```
extension FirstCafe {
    var cafeName: String = "CafeName"
}
```

✖ Extensions must not contain stored properties ✖

## Объявление вложенных типов в расширении

В расширении можно объявить вложенный тип, делается это как и в обычном классе/структуре/перечислении. Давайте добавим вложенный тип Tea в FirstCafe

```
extension FirstCafe {
    struct Tea {
        var name: String
        var cost: Double
    }
}
```

```
}
```



У класса/структуры/перечисления может быть не одно расширение

## Дефолтная реализация протокола

При помощи расширения можно задать дефолтную реализацию для протокола. Например, у нас есть метод `addCoffee`, давайте вынесем его реализацию в протокол. Для этого сначала указываем `extension`, затем имя протокола, а именно `CafeProtocol`, после фигурных скобок указываем функцию `addCoffee` и реализуем ее, то есть пишем код для функции, пусть сейчас функция печатает название переданного кофе

```
extension CafeProtocol {  
    func addCoffee(coffee: Coffee) {  
        print(coffee.name)  
    }  
}
```

Теперь, если в классе `Cafe` не реализовывать метод `addCoffee`, то будет выполняться код, указанный в дефолтной реализации

```
var cafe = Cafe()  
let espresso = Coffee(name: "Espresso", isSugar: true, isIce:  
false, cost: 50, size: .l)  
cafe.addCoffee(coffee: espresso) // Выведет в консоль Espresso
```

А если функцию реализовать, то выполняться будет функция, указанная в классе, а не в дефолтной реализации

```
var cafe = Cafe()  
let espresso = Coffee(name: "Espresso", isSugar: true, isIce:  
false, cost: 50, size: .l)  
cafe.addCoffee(coffee: espresso) // Добавит в массив coffee  
espresso
```

## ООП

ООП (объектно-ориентированное программирование) - методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

# Наследование

Наследование - это один из принципов ООП, в ходе которого один класс может наследоваться от другого. Класс от которого происходит наследование - суперкласс, а класс, который наследует - подкласс.

Создадим класс Cafe, он будет суперклассом, от которого будут наследоваться другие кафе. Cafe подписан на CafeProtocol, так как все кафе должны реализовывать этот протокол

```
class Cafe: CafeProtocol {
    var coffee: [Coffee]

    required init(coffee: [Coffee]) {
        self.coffee = coffee
    }

    convenience init() {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        self.init(coffee: [latte, cappuccino])
    }

    func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}
```

Наследование имеет следующую конструкцию:

```
class <Имя класса наследника>: <Имя класса родителя> { }
```

Для того, чтобы FirstCafe, SecondCafe и ThirdCafe наследовались от Cafe - необходимо после двоеточия указать класс родитель.



Если помимо наследования класс подписан на протоколы - калл родитель должен идти перед протоколами.

Удалим все методы, переменные и инициализаторы из классов наследников.

```
class Cafe: CafeProtocol {
```

```

var coffee: [Coffee]

required init(coffee: [Coffee]) {
    self.coffee = coffee
}

convenience init() {
    let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
    let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
    self.init(coffee: [latte, cappuccino])
}

func addCoffee(coffee: Coffee) {
    self.coffee.append(coffee)
}
}

class FirstCafe: Cafe {}
class SecondCafe: Cafe {}
class ThirdCafe: Cafe {}

```

Теперь, когда мы создадим экземпляры классов, несмотря на то, что никаких методов и свойств в классах наследниках не реализовано - мы все равно можем использовать свойства и методы суперкласса.

```

let firstCafe = FirstCafe()
let secondCafe = SecondCafe()
let thirdCafe = ThirdCafe()

firstCafe.addCoffee(coffee: espresso) // в массив coffee класса
FirstCafe будет добавлен espresso
print(secondCafe.coffee) // в консоль будет выведен массив кофе
класса SecondCafe
thirdCafe.addCoffee(coffee: espresso) // в массив coffee класса
ThirdCafe будет добавлен espresso

```

## override

Вспомним про SecondCafe, ведь в этом классе метод addCoffee не просто добавлял напиток в массив, но и выводил количество кофе в массиве, а значит метод класса родителя не удовлетворяет всем требованиям. Здесь на помощь приходит override, это ключевое слово указывается перед вычисляемым свойством, методом или инициализатором, чтобы переопределить его.



Переопределим метод `addCoffee` в классе `SecondCafe`, для этого в классе `SecondCafe` напишем функцию `addCoffee`, но перед ключевым словом `func` укажем `override`, что означает, что это переопределение функции класса родителя.

```
class SecondCafe: Cafe {
    override func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
        print(self.coffee.count)
    }
}
```

Теперь внимательно рассмотрим функцию `addCoffee`, первая строка функции полностью повторяет функцию `addCoffee` родительского класса. А теперь представим, что повторяется не одна строка, а сразу 10. Что делать? Копировать их? Нет, это плохой вариант. В таком случае нужно вызвать метод класса родителя, для этого сначала указывается `super`, это ключевое слово показывает, что идет обращение к классу родителю, а затем вызвать необходимую функцию.

```
override func addCoffee(coffee: Coffee) {
    super.addCoffee(coffee: coffee)
    print(self.coffee.count)
}
```

Теперь вызовем метод `addCoffee` для всех трех кафе.

```
firstCafe.addCoffee(coffee: espresso)
secondCafe.addCoffee(coffee: espresso)
thirdCafe.addCoffee(coffee: espresso)
```

Для первого и третьего кафе будет выполнен код, который определен в функции класса родителя, а вот в случае второго кафе - функция переопределена и выполнится код, указанный в классе `SecondCafe`, то есть не только добавится кофе в массив, но и будет выведено в консоль количество напитков в массиве `coffee`.

## final

`final` - ключевое слово, которое показывает, что метод, свойство или инициализатор переопределять нельзя. Укажем `final` перед `addCoffee` в родительском классе `Cafe`

```
final func addCoffee(coffee: Coffee) {
    self.coffee.append(coffee)
}
```

Тут же появляется ошибка у класса `SecondCafe`, ведь теперь этот метод нельзя переопределять

```
class SecondCafe: Cafe {
    override fun addCoffee(coffee: Coffee) {
        super.addCoffee(
            print(self.coffee.count,
        }
    }
}
```

Instance method overrides a 'final' instance method

Также ключевым словом `final` можно помечать целый класс, тогда от него нельзя будет наследоваться. Укажем `final` перед `FirstCafe`, создадим новый класс и попытаемся наследоваться от него.

```
final class FirstCafe: Cafe {}
class FirstCafeChild: FirstCafe {}
```

Этот код работать не будет, так как `FirstCafe` помечен как `final`, а значит наследоваться от него нельзя.

```
final class FirstCafe: Cafe {}
class FirstCafeChild: FirstCafe {}
```

Inheritance from a final class 'FirstCafe'

## Полиморфизм

Полиморфизм - один из принципов ООП, который предоставляет возможность взаимозаменяемости типов, которые находятся в одной иерархии.

Давайте создадим новый класс `FoodTruck`, который будет наследником `SecondCafe`, таким образом теперь у нас есть иерархия наследования. Есть класс `Cafe`, от него наследуется `SecondCafe`, а от него в свою очередь наследуется `FoodTrack`

```
class Cafe: CafeProtocol {
    var coffee: [Coffee]

    required init(coffee: [Coffee]) {
        self.coffee = coffee
    }

    convenience init() {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        self.init(coffee: [latte, cappuccino])
    }

    funс addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}
```

```

}

class SecondCafe: Cafe {
    override func addCoffee(coffee: Coffee) {
        super.addCoffee(coffee: coffee)
        print(self.coffee.count)
    }
}

class FoodTruck: SecondCafe {}

```

Теперь создадим три переменных: `cafe`, `secondCafe` и `foodTruck`. По сути они все являются подвидами кафе, а именно наследниками `Cafe`.

```

var cafe: Cafe = Cafe()
var secondCafe: Cafe = SecondCafe()
var foodTruck: Cafe = FoodTruck()

```

Все три переменных представляют тип `Cafe`, но `cafe` хранит ссылку на объект `Cafe`, `secondCafe` на `SecondCafe`, а `foodTruck` на `FoodTruck`. Таким образом переменные одного и того же типа (в данном случае `Cafe`), могут принимать разные формы (а именно и `Cafe`, и `SecondCafe`, и `FoodTruck`)

## Инкапсуляция

Инкапсуляция - один из принципов ООП, который подразумевает сокрытие методов и свойств, которые не нужны при использовании данного класса. В случае с инкапсуляцией очень полезны уровни доступа. Вспомним, какие уровни доступа есть:

1. `open` - разрешает доступ и переопределение в родном модуле и импортирующем модуле. Используется только для классов, их свойств и методов
2. `public` - разрешает доступ и переопределение в родном модуле, а в импортирующем модуле разрешен только доступ.
3. `internal` - разрешает использование объекта внутри любого файла исходного модуля, но не в файлах не из этого модуля
4. `fileprivate` - разрешает использование объекта только в пределах исходного файла
5. `private` - разрешает использование объекта только в пределах области реализации

Например, функция `addCoffee` - `internal`, а значит мы можем вызвать этот метод внутри любого файла исходного модуля. Почему же мы делаем его `internal`? Потому, как не только мы можем добавлять новый напиток, но и любой посетитель может добавить или порекомендовать напиток.

Теперь добавим в Cafe метод sale.

```
private func sale() {}
```

Этот метод будет являться приватным и запустить его можно только внутри класса, то есть внутри кафе. Сделано так потому, что только мы можем запустить скидки, но не любой посетитель, который этого захотел.

Это пример того, как можно скрыть функции и свойства от внешних изменений и использования.

## Дженерики

Универсальные шаблоны (или дженерики) позволяют писать многократно используемые функции и типы данных, которые могут работать с любым типом, отвечающим определенным требованиям.

Например, добавим в протокол CafeProtocol и класс Cafe переменную tea, а также создадим структуру Tea, которая будет содержать название и стоимость чая. Для удобства мы также добавим параметр tea в инициализаторах.

```
struct Tea {
    var name: String
    var cost: Double
}

protocol CafeProtocol {
    var coffee: [Coffee] { get }
    var tea: [Tea] { get }

    func addCoffee(coffee: Coffee)

    init(coffee: [Coffee], tea: [Tea])
}

class Cafe: CafeProtocol {
    var coffee: [Coffee]
    var tea: [Tea]

    required init(coffee: [Coffee], tea: [Tea]) {
        self.coffee = coffee
        self.tea = tea
    }

    convenience init() {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
```

```

        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        let blackTea = Tea(name: "Black Tea", cost: 50)
        self.init(coffee: [latte, cappuccino], tea: [blackTea])
    }

    func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }
}

```

Теперь добавим функцию, которая будет печатать переданный напиток. Но здесь начинаются проблемы, ведь теперь у нас два типа напитков: чай и кофе, а значит нужно писать две разные функции: функция, у которой входной параметр имеет тип Coffee и функция, у которой входной параметр имеет тип Tea.

```

func printDrink(drink: Coffee) {
    print(drink)
}

func printDrink(drink: Tea) {
    print(drink)
}

```

Мы видим, что функции отличаются только типом параметра drink. Здесь нам пригодятся дженерики, перепишем эту функцию, для этого после названия функции указываем <T>, вместо T может быть любая другая буква или слово, T это обозначения для типа, который будет использоваться. Затем для параметра drink указываем тип T, это значит, что drink может быть любым типом.

```

func printDrink<T>(drink: T) {
    print(drink)
}

```

Теперь в функцию printDrink мы можем передать и параметр типа Coffee, и параметр типа Tea.

```

var cafe = Cafe()
let espresso = Coffee(name: "Espresso", isSugar: true, isIce:
false, cost: 50, size: .l)
let blackTea = Tea(name: "Black Tea", cost: 50)
cafe.printDrink(drink: espresso)
cafe.printDrink(drink: blackTea)

```

То есть, когда мы вызываем функцию `printDrink` и передаем `espresso` - `T` подразумевает тип `Coffee`, а когда вызываем `printDrink` и передаем `blackTes` - `T` подразумевает тип `Tea`.



Хотя вместо `T` может быть любая другая буква или даже слово, принято в дженериках использовать для обозначения типа именно `T`

## Универсальные типы

Выше мы рассмотрели пример универсальной функции, теперь перейдем к универсальным типам - это классы, структуры и перечисления, которые могут работать с любым типом.

Например, у нас есть класс витрины на которой могут быть выставлены десерты одного типа, а данном случае типа `Cake`

```
class Showcase {  
    var desserts: [Cake] = []  
}
```

А теперь мы хотим добавить еще одну витрину, но уже с эклерами, то есть с типом `Eclair`. Но `Showcase` может содержать десерты только типа `Cake`, получается нам нужно создавать новый класс, в котором десерты могут быть типом `Eclair`

```
struct Eclair {  
    var taste: String  
    var cost: Double  
}  
  
class ShowcaseV2 {  
    var desserts: [Eclair] = []  
}
```

А если у нас будут появляться еще типы десертов? Получается придется создавать все новые и новые классы витрин. Здесь на помощь придут универсальные типы, или generic-типы. После названия класса указываем `<T>`, это покажет, что класс может работать с типами `T`, то есть с заранее неизвестными типами. А свойству `desserts` указываем `[T]`, что указывает, что это массив типов `T`.

```
class Showcase<T> {  
    var desserts: [T] = []  
}
```

Теперь нам не нужно плодить разные классы витрин, мы можем создавать экземпляр витрины с разными десертами

```
var firstShowcase = Showcase<Cake>()
var secondShowcase = Showcase<Eclair>()
```

В firstShowcase после Showcase мы указываем <Cake>, что означает, что эта витрина будет работать именно с типом Cake, а в secondShowcase мы указываем <Eclair>, а значит эта витрина сможет работать только с типом Eclair.

## Ограничения универсальных типов

Универсальный тип может быть любого типа, из-за этого много вещей с ним сделать не получится, поэтому есть смысл по необходимости задавать ограничения для таких типов. Например, у нас есть универсальная функция, которая может складывать два разных значения и возвращать результат вычисления.

```
func add<T>(a: T, b: T) -> T {
    a + b
}
```

Но мы не сможем скомпилировать данный код, ведь под T может подразумеваться что угодно, а сложить можно не любые объекты.

```
func add<T>(a: T, b: T) -> T {
    a + b
}
```

Binary operator '+' cannot be applied to two 'T' operands

В таком случае нам помогут ограничения. Давайте укажем, что T обязательно соответствует протоколу Numeric.

```
func add<T: Numeric>(a: T, b: T) -> T {
    a + b
}
```

Ошибка ушла, ведь этому протоколу соответствует ограниченный набор типов, а значит передать тип, который не поддерживает сложение не удастся.

```
add(a: 4, b: 6)
add(a: 6.20, b: 4)
add(a: espresso, b: espresso)
```

Global function 'add(a:b:)' requires that 'Coffee' conform to 'Numeric'

## associatedtype

Давайте создадим протокол для витрины, в котором будет обозначено, что класс должен обязательно иметь метод добавления нового десерта на витрину и метод, который будет сообщать, пуста ли витрина.

```
protocol ShowcaseProtocol {  
    func addDessert(dessert: ?)  
    func isEmpty() -> Bool  
}
```

Тут появляется вопрос: а какой же тип у добавляемого десерта? Чтобы мы могли использовать и Cake и Eclair необходимо использовать associatedtype, это ключевое слово помогает задать имя-заглушку для типа, который используется в качестве части протокола.

```
protocol ShowcaseProtocol {  
    associatedtype Dessert  
    func addDessert(dessert: Dessert)  
    func isEmpty() -> Bool  
}
```

Теперь подпишем класс Showcase на протокол.

```
class Showcase<T>: ShowcaseProtocol {  
    typealias Dessert = T  
  
    var desserts: [T] = []  
  
    func addDessert(dessert: T) {  
        desserts.append(dessert)  
    }  
  
    func isEmpty() -> Bool {  
        desserts.isEmpty  
    }  
}
```

Чтобы реализовать протокол мы должны:

1. Объявить, какой тип использовать в качестве Dessert
2. Реализовать методы, которые требует протокол



## Что можно почитать еще?

1. [SwiftBook. Протоколы](#)
2. [SwiftBook. Расширения](#)
3. [SwiftBook. Универсальные шаблоны](#)
4. [Numeric](#)

# Используемая литература

1. <https://docs.swift.org/swift-book/LanguageGuide/Protocols.html>
2. <https://docs.swift.org/swift-book/LanguageGuide/Generics.html>
3. <https://docs.swift.org/swift-book/LanguageGuide/Extensions.html>