







# Содержание

- ★ Введение в ООП 
- ★ Инкапсуляция 
- ★ Наследование 
- ★ Полиморфизм 
- ★ Протоколы 
- ★ Расширения 

# Что такое ООП?

**ООП** означает **объектно-ориентированное программирование**

Процедурное программирование — это написание процедур или методов, выполняющих операции с данными, в то время как **объектно-ориентированное программирование** — это создание объектов, содержащих как данные, так и методы.

**Объектно-ориентированное программирование** имеет несколько преимуществ перед процедурным программированием:

- ❑ ООП быстрее и проще в исполнении
- ❑ ООП обеспечивает четкую структуру программ
- ❑ ООП помогает сохранить код DRY (DontRepeatYoureslf) и упрощает поддержку, изменение и отладку кода
- ❑ ООП позволяет создавать полностью повторно используемые приложения с меньшим количеством кода и более коротким временем разработки

# 4 принципа ООП

1. Инкапсуляция
2. Наследование
3. Полиморфизм
4. Абстракция

# Инкапсуляция (Encapsulation)

Смысл **инкапсуляции** в том, чтобы убедиться, что «**конфиденциальные**» данные скрыты от пользователей. Чтобы достичь этого, вы должны:

- объявить переменные/атрибуты класса как **private**
- предоставить общедоступные методы получения и установки для доступа и обновления значения частной (приватной) переменной

```
3 struct Person {
4     let name: String
5     let surname: String
6     private var age: Int = 0
7
8     init(name: String, surname: String) {
9         self.name = name
10        self.surname = surname
11    }
12
13    mutating func set(_ age: Int) {
14        self.age = age
15    }
16 }
17
18 let person = Person(name: "John", surname: "Smith")
19 person.age = 20
```

⊗ 'age' is inaccessible due to 'private' protection level

# Зачем нужна инкапсуляция?

- ❑ Улучшенный контроль над атрибутами и методами класса
- ❑ Атрибуты класса можно сделать доступными **только для чтения** (если вы используете только метод **get**) или **только для записи** (если вы используете только метод **set**)
- ❑ Гибкость: *программист может изменить одну часть кода, не затрагивая другие части*
- ❑ Повышенная безопасность данных

# Наследование (Inheritance)

- ❑ Класс может **наследовать** методы, свойства и другие характеристики от другого класса
- ❑ Когда один класс наследуется от другого, наследующий класс называется **подклассом (subclass)**, а класс, от которого он наследуется, называется его **суперклассом (superclass)**
- ❑ Классы в Swift могут *вызывать и получать доступ к методам, свойствам и индексам, принадлежащим их суперклассу*, и могут предоставлять свои собственные переопределяющие версии этих методов, свойств и индексов для уточнения или изменения своего поведения (известное как, **полиморфизм**)
- ❑ Классы также *могут добавлять наблюдателей свойств к унаследованным свойствам*, чтобы получать уведомления об изменении значения свойства
- ❑ Наблюдатели свойств могут быть добавлены к любому свойству, независимо от того, было ли оно первоначально определено как сохраненное или вычисляемое свойство

# Подклассы

- ❑ **Создание подклассов (наследование)** — это действие по созданию нового класса на основе существующего класса
- ❑ **Подкласс** наследует характеристики существующего класса, которые затем можно уточнить. Вы также можете добавить новые характеристики в подкласс.
- ❑ Чтобы указать, что *у подкласса есть суперкласс, напишите имя подкласса перед именем суперкласса, разделенное двоеточием (:)*

```
1 class SomeSubclass: SomeSuperclass {  
2     // subclass definition goes here  
3 }
```

# Подклассы

В следующем примере определяется подкласс *Bicycle* с суперклассом *Vehicle*

```
1 class Vehicle {
2     var currentSpeed = 0.0
3     var description: String {
4         return "traveling at \$(currentSpeed) miles per hour"
5     }
6     func makeNoise() {
7         // do nothing - an arbitrary vehicle doesn't necessarily make a
8         noise
9     }
10 }
```

```
1 class Bicycle: Vehicle {
2     var hasBasket = false
3 }
```

```
1 bicycle.currentSpeed = 15.0
2 print("Bicycle: \$(bicycle.description)")
3 // Bicycle: traveling at 15.0 miles per hour
```



# Полиморфизм (Polymorphism)

**Полиморфизм** означает «много форм», и он возникает, когда у нас есть много классов, связанных друг с другом путем наследования

Вы можете *переопределить унаследованный метод объекта или типа, чтобы обеспечить адаптированную или альтернативную реализацию метода* в вашем подклассе

```
1 class Train: Vehicle {
2     override func makeNoise() {
3         print("Choo Choo")
4     }
5 }
```

```
1 let train = Train()
2 train.makeNoise()
3 // Prints "Choo Choo"
```

# Предотвращение переопределения

- ❑ Вы можете **предотвратить переопределение** метода, свойства или индекса, пометив их как **окончательные**. Для этого напишите модификатор **final** перед вводным словом метода, свойства или нижнего индекса (например, *final var*, *final func*, *final class func* и *final subscript*)
- ❑ О любой попытке переопределить окончательный метод, свойство или индекс в подклассе сообщается как об **ошибке времени компиляции** (**compile-time error**)
- ❑ Методы, свойства или индексы, которые вы добавляете к классу в расширении, также могут быть помечены как окончательные в определении расширения (*final extension*)
- ❑ Вы можете пометить весь класс как окончательный, написав модификатор **final** перед ключевым словом **class** в его определении класса (*final class*)
- ❑ О любой попытке создать подкласс конечного(базового) класса сообщается как об ошибке времени компиляции

# Абстракция

**Абстракция данных** — это процесс сокрытия определенных деталей и показа пользователю только важной информации

Абстракция может быть достигнута с помощью **протоколов (интерфейсов)**

# Протоколы (Protocols)

**Протокол(protocols)** определяет **шаблон (blueprint)** методов, свойств и других требований, которые подходят для конкретной задачи или функциональной части

Затем **протокол** может быть **принят (implemented)** классом, структурой или перечислением, чтобы обеспечить фактическую реализацию этих требований

Говорят, что *любой тип, удовлетворяющий требованиям протокола, соответствует этому протоколу*

# Синтаксис Протоколов

- ❏ Вы определяете протоколы также как и классы, структуры и перечисления, используя ключевое слово *protocol*

```
1 protocol SomeProtocol {  
2     // protocol definition goes here  
3 }
```

# Синтаксис Протоколов

- ❑ Пользовательские типы (классы, структуры, перечисления) утверждают, что они используют определенный протокол, помещая имя протокола после имени типа, разделенного двоеточием(:), как часть их определения. Можно указать несколько протоколов, разделенных запятыми

```
1 struct SomeStructure: FirstProtocol, AnotherProtocol {  
2     // structure definition goes here  
3 }
```

# Синтаксис Протоколов

- ❑ Если у класса есть суперкласс, укажите имя суперкласса перед любыми протоколами, которые он принимает, после запятой

```
1 class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {  
2     // class definition goes here  
3 }
```

# Требования к Свойствам (атрибутам)

- ❑ **Протокол** может *потребовать от любого соответствующего типа предоставить свойство объекта или свойство типа с определенным именем и типом*
- ❑ Протокол не указывает, должно ли свойство быть хранимым свойством или вычисляемым свойством — он только указывает требуемое имя и тип свойства
- ❑ Протокол также указывает, должно ли каждое свойство быть доступным только для получения данных (**get**) или доступным для получения и записи данных (**get/set**)



# Требования к Свойствам (атрибутам)

- ❑ Если протокол требует, чтобы свойство было доступным для получения и записи данных, это требование свойства не может быть выполнено **постоянным хранимым свойством (let)** или вычисляемым свойством, доступным только для чтения
- ❑ Требования к свойствам всегда объявляются как переменные свойства с префиксом ключевого слова **var**
- ❑ **Получаемые и записываемые свойства** обозначаются записью **{get set}** после объявления их типа, а **получаемые свойства** обозначаются записью **{get}**

```
1 protocol SomeProtocol {  
2     var mustBeSettable: Int { get set }  
3     var doesNotNeedToBeSettable: Int { get }  
4 }
```

# Требования к Методам

- ❑ **Протоколы** могут *требовать, чтобы определенные методы экземпляра и методы типа были реализованы соответствующими типами*
- ❑ Эти методы записываются как часть определения протокола точно так же, как и обычные методы экземпляра и типа, *но без фигурных скобок или тела метода*
- ❑ Разрешены **переменные(variadic) параметры**, подчиняющиеся тем же правилам, что и для обычных методов
- ❑ Однако *значения по умолчанию не могут быть указаны для параметров метода в определении протокола*

```
1 protocol SomeProtocol {  
2     static func someTypeMethod()  
3 }
```

# Требования к Методу мутации (mutating)

Иногда методу необходимо модифицировать (или мутировать) объект, которому он принадлежит

Для методов объекта **Типов Значений (value type)** (то есть структур и перечислений) вы помещаете ключевое слово **mutating** перед ключевым словом **func** метода, чтобы указать, что методу разрешено изменять объект, которому он принадлежит, и любые свойства этого объекта

```
1 protocol Toggleable {  
2     mutating func toggle()  
3 }
```

```
1 enum OnOffSwitch: Toggleable {  
2     case off, on  
3     mutating func toggle() {  
4         switch self {  
5             case .off:  
6                 self = .on  
7             case .on:  
8                 self = .off  
9         }  
10    }  
11 }  
12 var lightSwitch = OnOffSwitch.off  
13 lightSwitch.toggle()  
14 // lightSwitch is now equal to .on
```

# Требования к Конструктору(инициализатору)

Протоколы могут *требовать, чтобы определенные инициализаторы были реализованы соответствующими типами*

Вы пишете эти инициализаторы как часть определения протокола точно так же, как и обычные инициализаторы, но без фигурных скобок или тела инициализатора

```
1 protocol SomeProtocol {  
2     init(someParameter: Int)  
3 }
```

# Протоколы как типы данных

Протоколы сами по себе не реализуют никакой функциональности. Тем не менее, *вы можете использовать протоколы как полноценные типы в своем коде*

Вы можете использовать протокол во многих местах, где разрешены другие типы, в том числе:

- ❑ Как тип параметра или тип возвращаемого значения в функции, методе или инициализаторе
- ❑ Как тип константы, переменной или свойства
- ❑ Как тип элементов в массиве, словаре или другом контейнере

```
1 class Dice {
2     let sides: Int
3     let generator: RandomNumberGenerator
4     init(sides: Int, generator: RandomNumberGenerator) {
5         self.sides = sides
6         self.generator = generator
7     }
8     func roll() -> Int {
9         return Int(generator.random() * Double(sides)) + 1
10    }
11 }
```

# Расширения протокола

Протоколы могут быть расширены для обеспечения реализации методов, инициализаторов, индексов и вычисляемых свойств для соответствующих типов. Это позволяет вам определять поведение самих протоколов, а не индивидуальное соответствие каждого типа или глобальную функцию (может использоваться как реализация по умолчанию)

Например, протокол `RandomNumberGenerator` можно расширить, чтобы предоставить метод `randomBool()`, который использует результат обязательного метода `random()` для возврата случайного значения `Bool`

```
1  extension RandomNumberGenerator {  
2      func randomBool() -> Bool {  
3          return random() > 0.5  
4      }  
5  }
```

# Расширения (Extensions)

**Расширения(extensions)** добавляют новые функциональности к существующему классу, структуре, перечислению или типу протокола.

Это включает в себя возможность расширения типов, для которых у вас нет доступа к исходному коду (дополнить свой или чужой код, необходимой логикой)

# Расширения (Extensions)

Расширения в Swift могут:

- Добавить свойства вычисляемого объекта и свойства вычисляемого типа
- Определение методов объекта и методов типов
- Предоставить новые инициализаторы
- Определение и использование новых вложенных типов
- Сделать существующий тип соответствующим протоколу



# Синтаксис Расширений

Объявляйте расширения с помощью ключевого слова *extension*

```
1 extension SomeType {  
2     // new functionality to add to SomeType goes here  
3 }
```

# Синтаксис Расширений

- ❑ *Расширение может расширять существующий тип, чтобы он принимал один или несколько протоколов*
- ❑ Чтобы **добавить соответствие протоколу**, вы пишете имена протоколов так же, как вы пишете их для класса или структуры

```
1 extension SomeType: SomeProtocol, AnotherProtocol {  
2     // implementation of protocol requirements goes here  
3 }
```

Thanks for your attention!