**Основы языка Swift**

**Переменные/Константы. Базовые типы. Строки и как с ними работать**

Swift — это современный, безопасный и производительный язык программирования от Apple. Он поддерживает строгую типизацию, автоматическое управление памятью и компиляцию с высокой оптимизацией.

Одной из ключевых концепций Swift является работа с переменными и константами, а также использование различных базовых типов данных. Разберём их подробно.

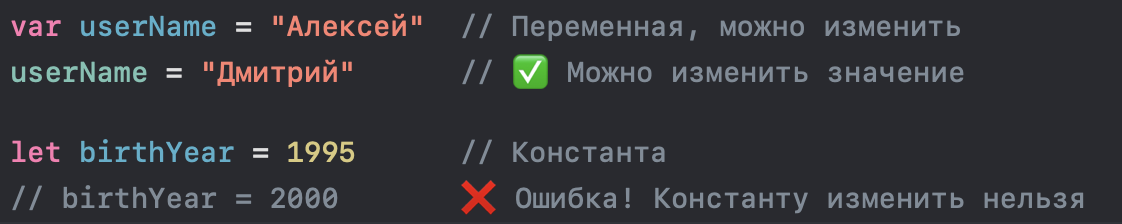
**Переменные (var) и Константы (let)**

**Объявление переменных и констант**

В Swift есть два способа хранить значения:

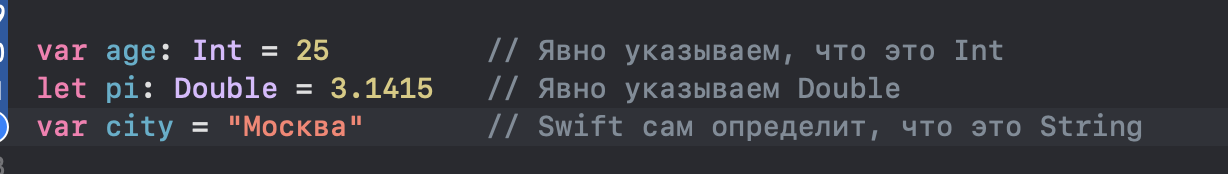
• **var** — переменная, значение которой можно изменять.

• **let** — константа, значение которой нельзя изменять после присваивания.



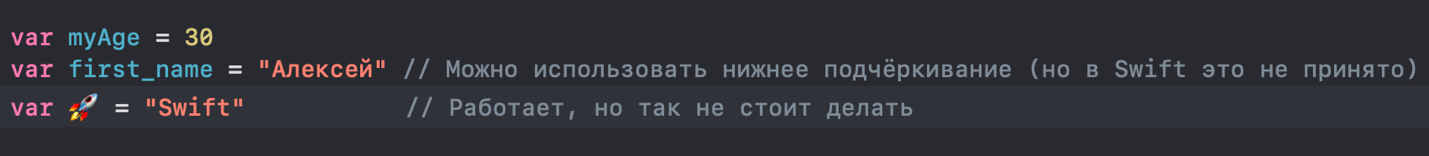
Всегда старайтесь использовать let, если значение не меняется, это делает код безопаснее и константы работают немного быстрее так как программе не нужно запоминать ее как изменяющиеся значение.

Также Swift использует автоматическое определение типа (Type Inference), но можно задавать тип вручную:



**Имена переменных**

Имена переменных могут содержать буквы, цифры, подчёркивания и даже эмодзи, но не могут начинаться с цифры:



**Базовые типы данных в Swift**

Swift поддерживает несколько встроенных типов данных, которые делятся на числовые, логические, строковые и составные.

|  |  |  |
| --- | --- | --- |
| **Тип данных** | **Описание** | **Пример** |
| **Int** | Целые числа | 42, -10 |
| **UInt** | Беззнаковые целые числа | 255, 1024 |
| **Double** | Дробные числа с высокой точностью | 3.1415, 0.001 |
| **Float** | Дробные числа с меньшей точностью | 3.14, 0.1 |
| **Bool** | Логические значения | true, false |
| **String** | Строки (текст) | "Hello, Swift" |
| **Character** | Один символ | 'A', '😃' |
| **Tuple** | Кортеж (несколько значений в одной переменной) | ("Swift", 5.7) |

Swift — язык со строгой типизацией. Это значит, что каждая переменная или константа в Swift **обязательно** имеет тип данных, который определяет, какие значения она может хранить и какие операции можно над ней выполнять.

**Почему это важно?**

• Типизация помогает **избежать ошибок** при работе с данными.

• **Оптимизация памяти**: компилятор знает, сколько места нужно для хранения значения.

• **Безопасность**: нельзя случайно передать в функцию не тот тип данных.

Теперь разберём каждый базовый тип подробно.

**Целые числа (Int, UInt)**

**Целые числа (integers)** — это числа без дробной части (1, 5, -10, 42).

В Swift их представляют два типа:

• Int (со знаком, может быть отрицательным или положительным)

• UInt (только положительные значения)

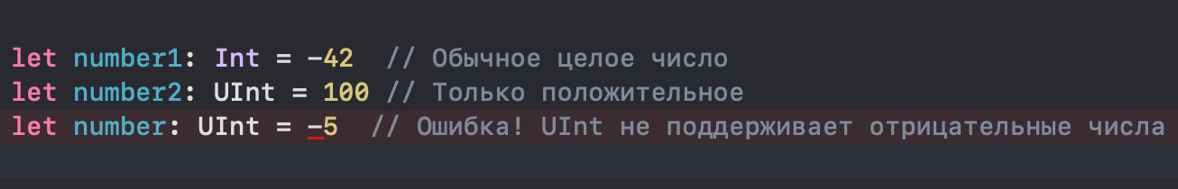
**Почему в Swift нет short, long, byte, как в других языках?**

В отличие от C или Java, Swift **по умолчанию использует** Int, который автоматически подстраивается под архитектуру процессора:

• **На 32-битных устройствах** Int занимает **32 бита**.

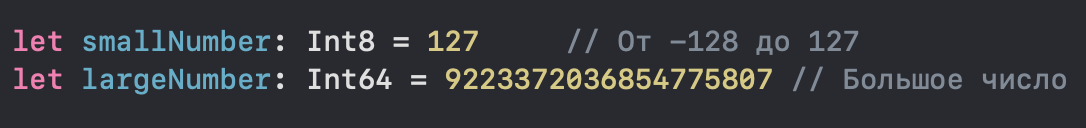
• **На 64-битных** — **64 бита**.

**Пример:**



**Явное указание размера (если нужно)**

Если важно контролировать размер, можно использовать Int8, Int16, Int32, Int64:



• Используйте Int, если нет особых требований по памяти.

• UInt подходит, если число **не может быть отрицательным** (например, для счёта очков в игре).

**Числа с плавающей запятой (Float, Double)**

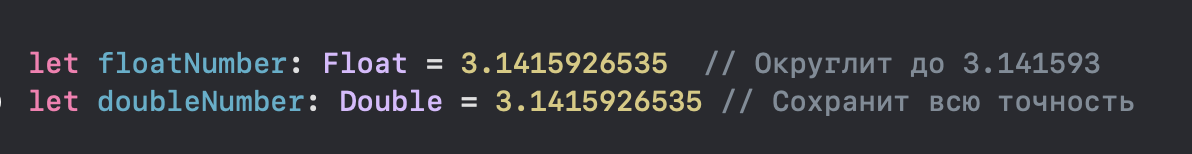
**Дробные числа (floating-point numbers)** используются, если нужно работать с числами, у которых есть **дробная часть** (3.14, 0.001, -2.5).

• Float — 32-битное число (точность **около 6-7 знаков**).

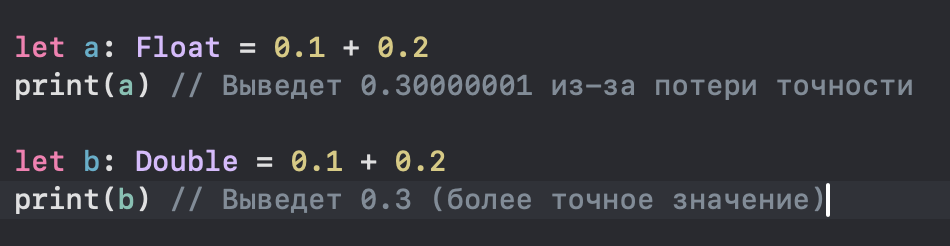
• Double — 64-битное число (**до 15-16 знаков**).

**Почему в Swift Double по умолчанию?**

Double более точный и предпочтителен в большинстве случаев, потому что Float может терять точность при вычислениях.



**Опасность округления (Float vs Double)**



**Вывод:**

• **Используйте** Double для точных вычислений (финансы, геолокация).

• Float лучше, если нужно **экономить память** (игры, графика).

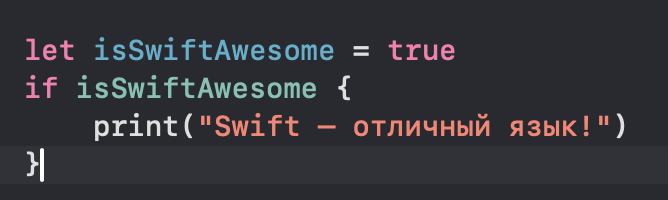
**Логический тип (Bool)**

Тип Bool принимает только два значения:

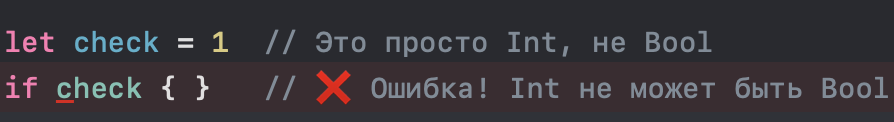
• true (**истина**)

• false (**ложь**)

Используется в условиях (if, while) и логических операциях:



**Важно:** в Swift нельзя использовать 1 и 0 вместо true и false, как в C или Python!



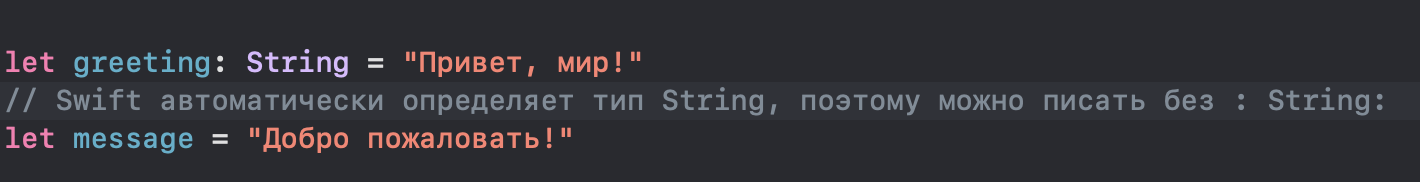
**Строки (String) в Swift**

Строки в Swift — это **упорядоченная последовательность символов**, поддерживающая **Юникод**. Они могут содержать текст на любом языке, эмодзи, специальные символы и даже составные символы (например, буква с диакритическим знаком).

Swift делает работу со строками максимально безопасной, поэтому **строки являются значением (value type)**, а не ссылочным типом.

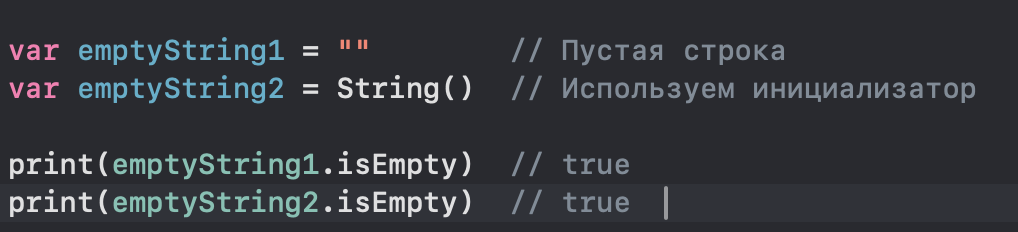
**Объявление и создание строк**

Создать строку можно с помощью "" (двойных кавычек):



**Пустые строки**

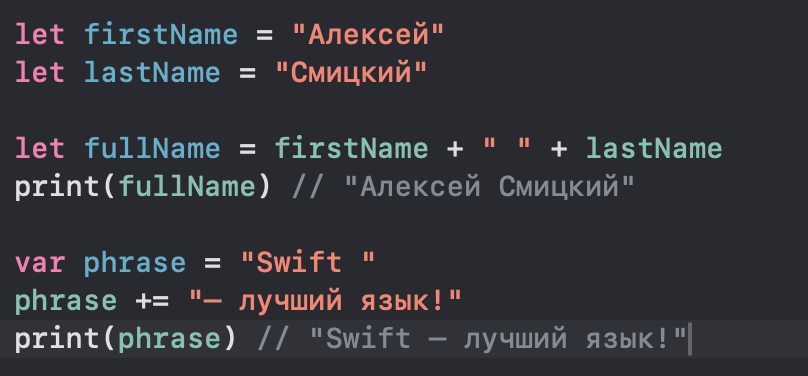
Чтобы создать пустую строку, есть два способа:



Метод .isEmpty проверяет, пустая ли строка.

**Конкатенация строк (сложение строк)**

В Swift можно объединять строки с помощью + или +=:

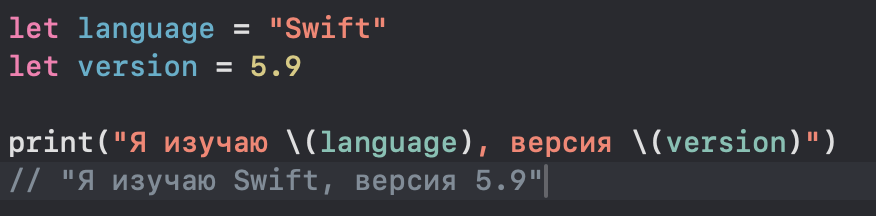


+ создаёт новую строку, а не изменяет существующую.

+= изменяет строку, если она объявлена с var (переменная).

**Интерполяция строк (вставка переменных в строку)**

Интерполяция позволяет вставлять переменные внутрь строки с помощью \(переменная):



**Преимущество интерполяции перед** +

Интерполяция **безопаснее и эффективнее**, чем +, потому что Swift автоматически конвертирует числа и другие типы в строку:

**Кортежи (Tupples)**

Кортежи — это тип данных, который позволяет **собирать несколько значений разных типов в одну переменную**. Кортежи полезны, когда нужно вернуть несколько значений из функции или просто собрать данные, которые логически связаны.

**Создание кортежа**

Чтобы создать кортеж, можно использовать круглые скобки и перечислить элементы через запятую:

**let person = ("Алексей", 25)  // Кортеж с именем и возрастом**

В этом примере person — это кортеж, который состоит из двух элементов: строки и числа.

**Кортежи с именованными элементами**

Можно дать имя каждому элементу в кортеже:

**let person = (name: "Алексей", age: 25)**

**print(person.name) // "Алексей"**

**print(person.age) // 25**

**Character**

Character в Swift представляет собой **один символ**, который является элементом строки. Хотя строки состоят из нескольких символов, каждый символ в Swift имеет свой собственный тип Character.

Когда мы работаем со строками в Swift, каждый символ представляет собой отдельный объект типа Character, даже если этот символ состоит из нескольких байт, как в случае с эмодзи.

**Пример использования Character:**

**let char: Character = "A"**

**print(char) // "A"**

**Коллекции. Массивы. Словари. Множества. Кортежи**

Коллекции позволяют хранить несколько значений под одной переменной, и каждый элемент в коллекции может быть доступен и изменен. Это одна из самых важных и интересных тем в программировании в целом – коллекции.

В Swift три основных типа коллекций:

1. **Массивы** (Array)

2. **Словари** (Dictionary)

3. **Множества** (Set)

**Массивы (Array)**

Массивы — это **упорядоченные коллекции**, где каждый элемент имеет свой индекс. В Swift массивы могут быть **гетерогенными** (смешанный тип данных) или **гомогенными** (все элементы одного типа). Чаще всего массивы используются для хранения элементов одного типа.

**Создание массива**

Массив можно создать несколькими способами:

****

**Доступ к элементам**

Можно обращаться к элементам массива через индекс:

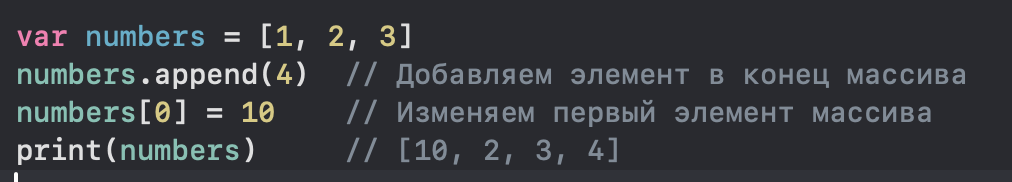
**let fruits = ["яблоко", "банан", "груша"]**

**print(fruits[0]) // "яблоко"**

Индексация начинается с нуля, то есть первый элемент массива имеет индекс 0. Также если вы обратитесь к несуществующему индексу в массиве, то произойдет ошибка исполнения программы.

**Изменение массива**

Массивы, созданные с var (переменной), можно изменять:

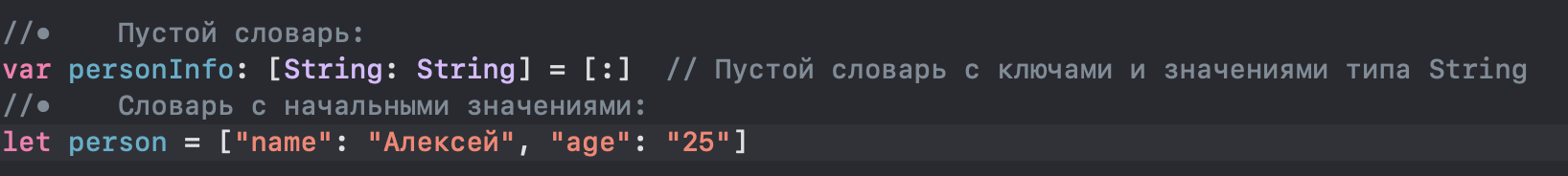


**Словари (Dictionary)**

Словарь в Swift — это **неупорядоченная коллекция**, где каждый элемент хранится в виде **пары ключ-значение**. Ключи должны быть уникальными, а значения могут быть любыми типами.

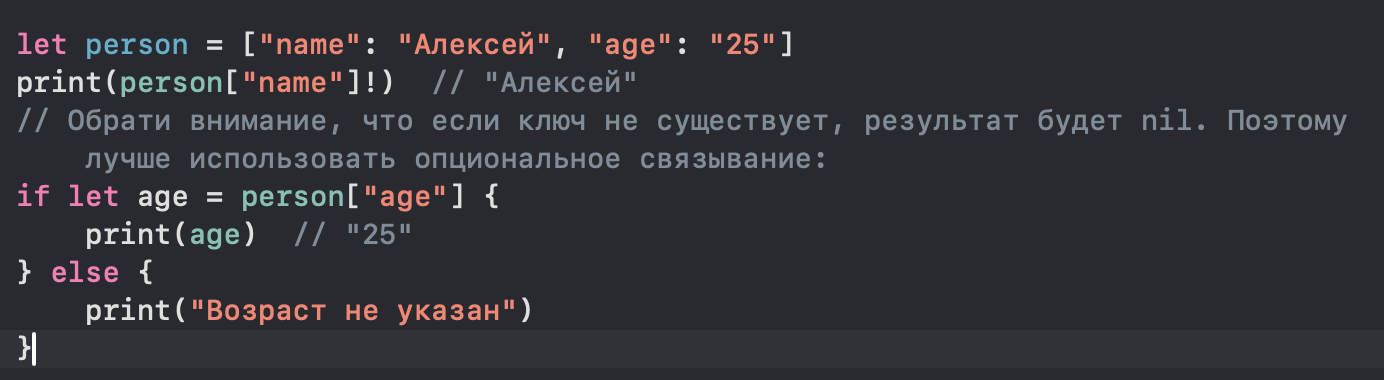
**Создание словаря**

Чтобы создать словарь, нужно объявить тип переменной/константы как словарь в виде [тип: тип] и затем либо оставить его пустым, либо наполнить данными сразу в формате ключ-значение.



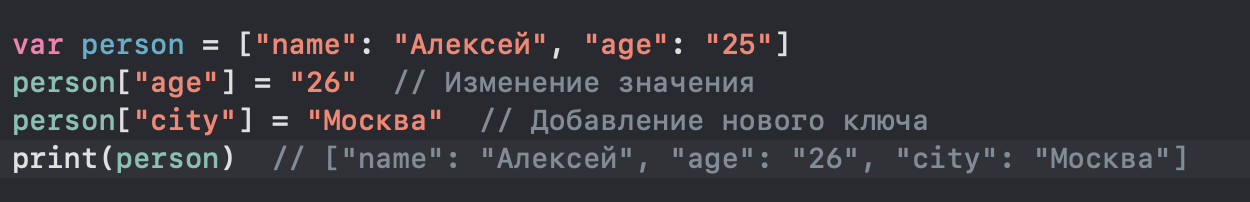
**Доступ к элементам словаря**

Чтобы получить значение по ключу, используем квадратные скобки:



**Добавление и изменение значений**

Чтобы добавить или изменить элемент в словаре, используем ключ:

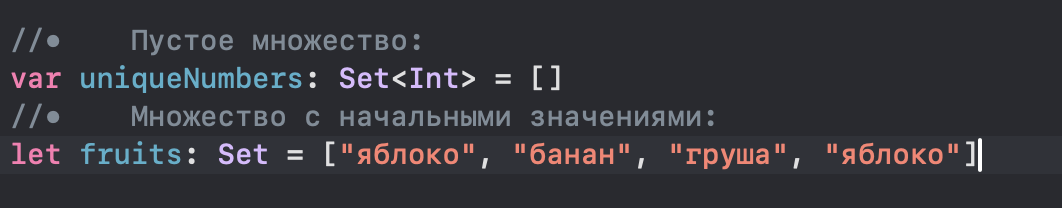


**Множества (Set)**

Множество — это **неупорядоченная коллекция**, которая не может содержать одинаковых элементов. В множестве могут храниться только уникальные значения, и порядок этих значений не гарантируется.

**Создание множества**

Схоже с массивами за исключением явного объявления типа данных как множество

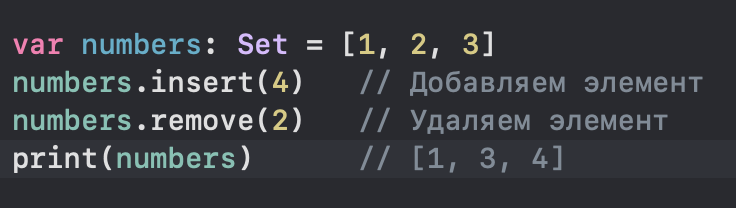


В результате, при создании множества, повторяющиеся значения будут удалены, и мы получим уникальные элементы:

**print(fruits)  // ["банан", "груша", "яблоко"]**

**Добавление и удаление элементов**

Для добавления или удаления элементов из множества используется следующий синтаксис:



**Итого мы имеем**

• **Массивы** — упорядоченные коллекции с доступом через индекс.

• **Словари** — неупорядоченные коллекции с доступом через уникальные ключи.

• **Множества** — неупорядоченные коллекции с уникальными элементами.

**Опциональные типы. Switch/case. If/else. For/in. While. Break. Return**

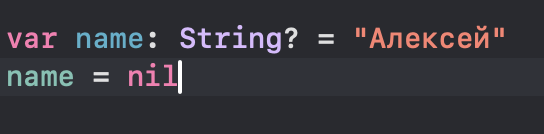
**Опциональные типы**

В Swift **опциональные типы** — это очень мощная особенность, которая помогает нам безопасно работать с отсутствующими значениями. Когда мы говорим, что значение может быть “отсутствующим” или “неизвестным”, мы имеем в виду **опционал**.

**Что такое опционал?**

Опционал в Swift — это специальный тип, который может **содержать значение** какого-либо типа или быть nil, что означает отсутствие значения.

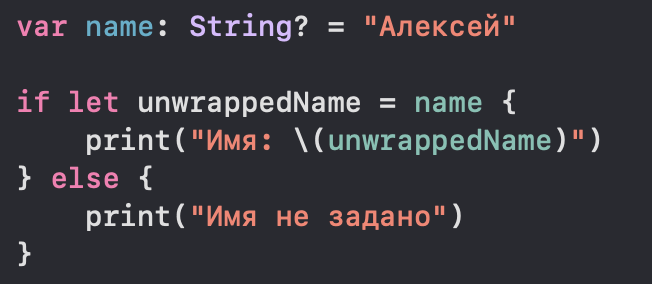
Пример:



Здесь переменная name может быть либо строкой, либо nil. То есть, она может не иметь значения. Это дает нам гибкость при работе с переменными, которые могут быть пустыми, например, когда мы запрашиваем данные из сети.

**Пример использования опционала**

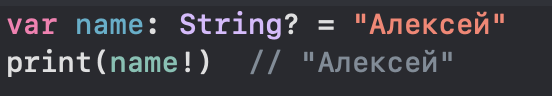
Допустим, у нас есть переменная с опциональным значением, и нам нужно безопасно проверить, есть ли у нее значение.



В этом примере мы используем **опциональную привязку (**if let**)**, чтобы безопасно извлечь значение из опционала. Если переменная name содержит строку, то это значение присваивается константе unwrappedName. Если же name равно nil, то срабатывает ветка else.

**Принудительное извлечение значения**

Иногда, если мы уверены, что опционал не nil, можно использовать принудительное извлечение с помощью оператора «!», но быть осторожным, так как это может привести к ошибке, если значение все-таки окажется nil.



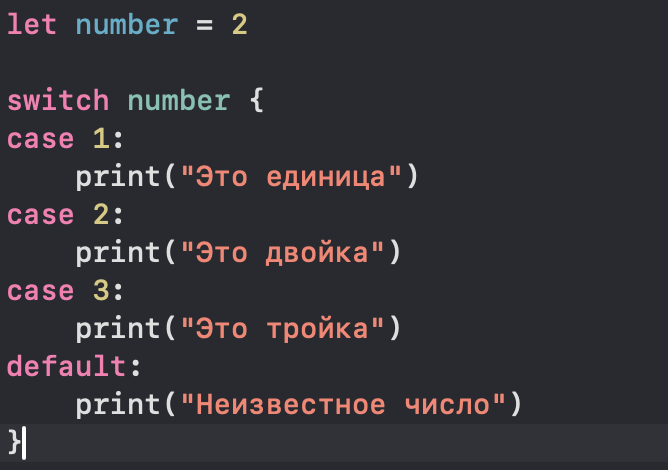
Однако, если name равно nil, программа крашнется с ошибкой. Поэтому лучше использовать безопасные способы извлечения.

**Switch/Case**

Перейдем к одной из самых удобных конструкций управления — **switch/case**. Это гораздо более мощная альтернатива if/else, особенно когда нужно работать с несколькими возможными значениями.

**Пример с числами**

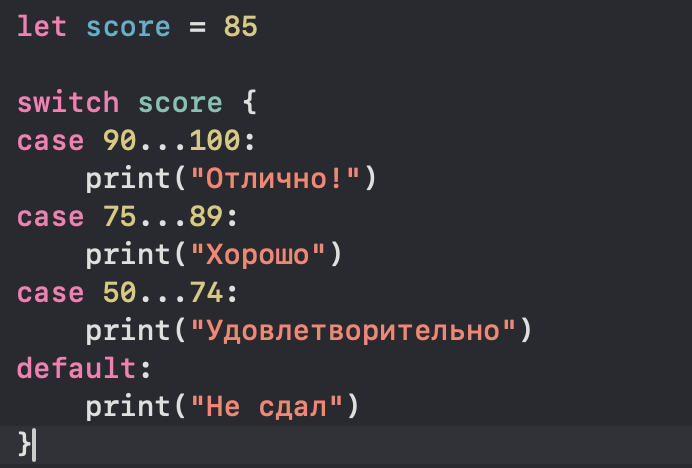
Допустим, мы хотим проверить число и выполнить разные действия в зависимости от его значения:



Здесь конструкция switch проверяет значение переменной number. Если оно равно 2, сработает соответствующий блок и выведется сообщение “Это двойка”. Если число не совпадает ни с одним из вариантов, сработает default.

**Использование диапазонов в switch**

В Swift switch поддерживает использование диапазонов, что делает код более выразительным:

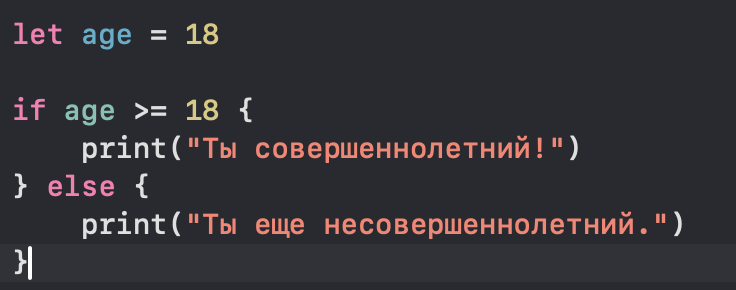


Здесь мы проверяем диапазоны, и код выглядит гораздо чище и понятнее, чем если бы мы использовали несколько if/else.

**If/Else**

В Swift классический способ ветвления — это конструкция if/else. Это самый базовый способ выполнения различных действий в зависимости от условий.

**Простой пример**

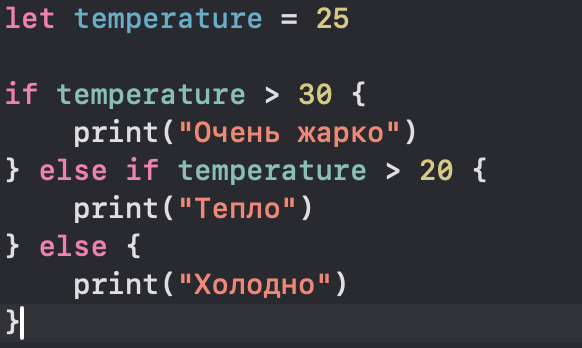


Это простой пример, где мы проверяем возраст, и если он больше или равен 18, выводим сообщение о совершеннолетии. Работает следующим образом – «если (if) age БОЛЬШЕ ИЛИ РАВЕН (>=) 18, то выводится один результат, если нет (else) выводится другой).  
Также существуют другие операторы:

**>** (больше) 19 **>** 20 “false”, **<** (меньше) 5 **<** 10 “true”  
 **>=, <=** (больше или равно меньше или равно)  
**==** (равно), **!=** (не равно) соответственно

**Множественные условия**

Можно использовать несколько else if, чтобы проверять несколько условий:

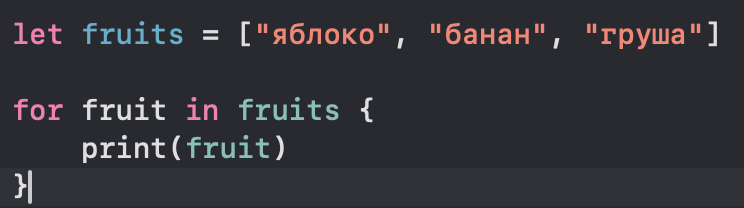


Здесь проверяется, какая температура, и в зависимости от этого выводится соответствующее сообщение.

**For/In**

Когда нам нужно пройтись по коллекции, мы используем конструкцию for/in. Эта конструкция позволяет перебирать элементы массива, словаря, множества или строки.

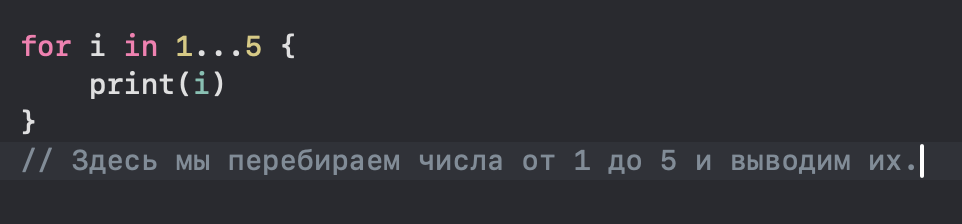
**Пример с массивом**



Этот код выведет каждый элемент массива поочередно.

**Пример с диапазоном**

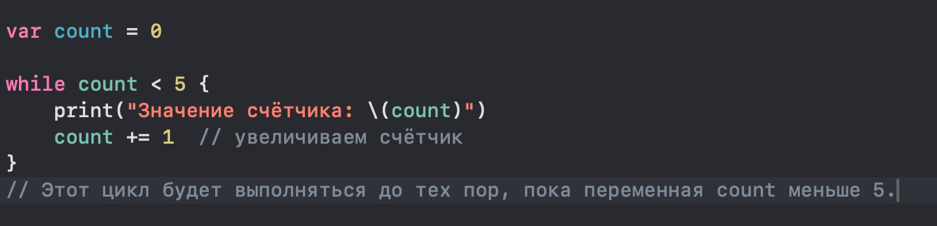
Можно также использовать диапазоны для перебора чисел:



**While**

Конструкция while позволяет выполнять блок кода до тех пор, пока условие остаётся истинным.

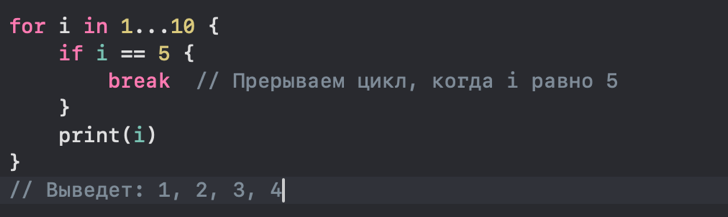
**Пример с циклом while**



**Break и Return**

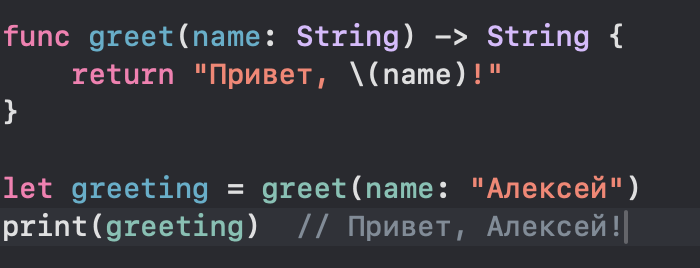
**Break**

Ключевое слово break используется для **выхода из цикла** раньше, чем он завершится по условию.



**Return**

Ключевое слово return используется для **выхода из функции**. Оно также может вернуть значение, если функция имеет тип, отличный от Void.



**Итак, мы рассмотрели…**

• **Опциональные типы** — с их безопасным извлечением значений.

• **Switch/case** — для удобной работы с несколькими вариантами значений.

• **If/else** — для проверки условий.

• **For/in** — для перебора элементов.

• **While** — для выполнения действий, пока условие истинно.

• **Break** и **Return** — для управления потоком выполнения кода.

**Функции. Замыкания. Enum (перечисления)**

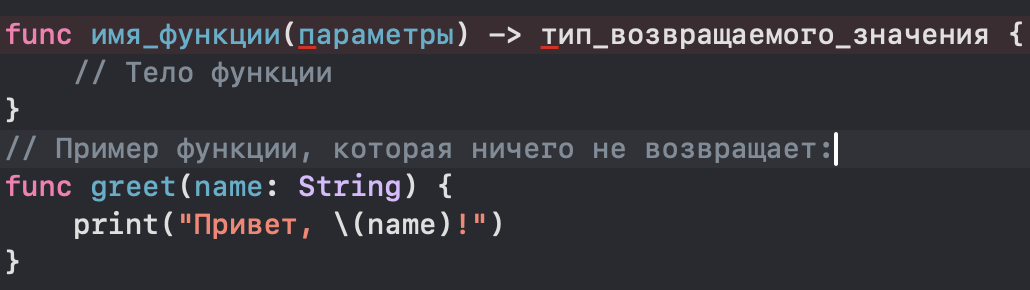
**Функции**

Функции — это основные строительные блоки программ в Swift. Они позволяют нам организовывать код, повторно использовать его и улучшать читаемость программы. Рассмотрим, как это работает и какие возможности предоставляет Swift для создания и использования функций.

Функция — это именованный блок кода, который выполняет определенную задачу. Она может принимать **параметры**, которые могут быть переданы в функцию, и **возвращать значения**, которые могут быть использованы за пределами функции. Основное назначение функций — это организация кода и улучшение его переиспользуемости.

**Создание функций**

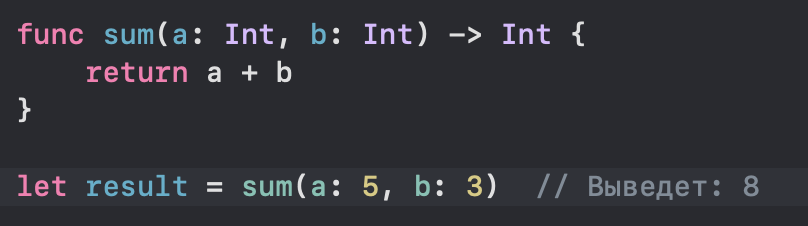
Для того чтобы создать функцию в Swift, мы используем ключевое слово func, за которым идет имя функции, параметры и возвращаемый тип. Стандартная структура функции выглядит так:



Функция greet принимает параметр name типа String и выводит строку с приветствием. Она ничего не возвращает, так как у нее нет типа возвращаемого значения (по умолчанию это Void).

**Функции с возвращаемым значением**

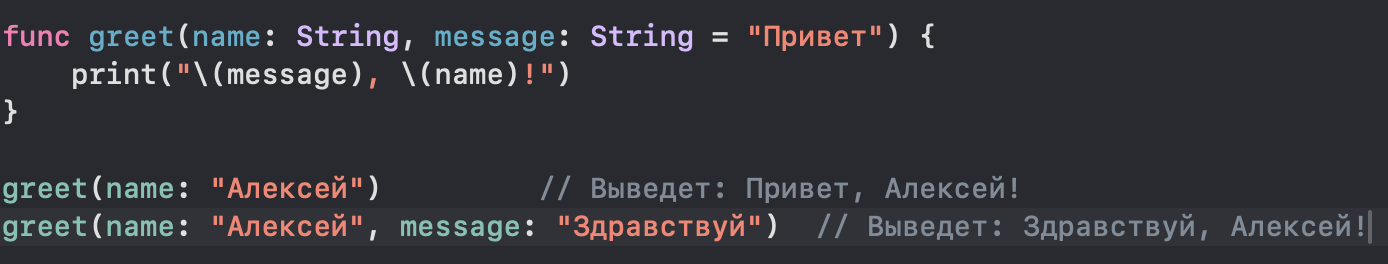
Большинство функций, которые вы будете писать, будут возвращать значение. Для этого нужно указать тип возвращаемого значения после стрелки ->. Вот пример:



Здесь функция sum принимает два параметра типа Int и возвращает их сумму. Тип возвращаемого значения (Int) указан после стрелки ->.

**Параметры по умолчанию**

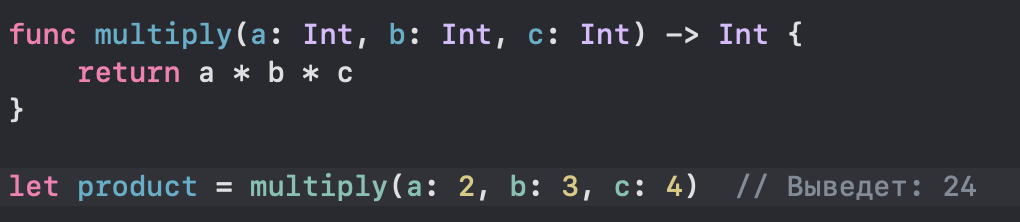
В Swift вы можете задать значения по умолчанию для параметров функции. Это удобно, когда вы хотите, чтобы параметры могли принимать значение по умолчанию, если пользователь не передаст его.



В этом примере параметр message имеет значение по умолчанию (“Привет”). Если оно не передано при вызове функции, будет использовано это значение.

**Функции с несколькими параметрами**

Функции могут принимать несколько параметров. Чтобы передать их в функцию, просто перечислите их через запятую:



Тут multiply принимает три параметра и возвращает их произведение.

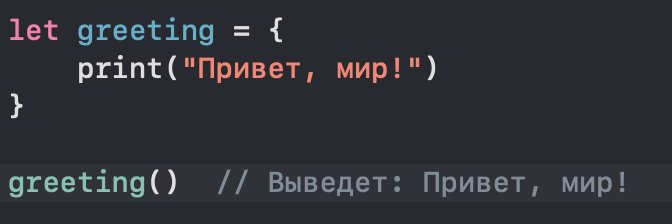
**Замыкания**

**Замыкания** (или **к closures**) — это одна из самых мощных и гибких функций Swift. Замыкания в Swift — это блоки кода, которые могут захватывать и сохранять контекст, в котором они были созданы. Это означает, что замыкания могут “запомнить” и использовать значения, которые были доступны в месте их создания.

Это анонимные функции, которые могут быть переданы в качестве параметров другим функциям. Они очень похожи на функции, но их использование значительно более гибкое. Например, замыкания часто используются в **асинхронных операциях**, **обработке событий** и **функциональных парадигмах**.

**Основы замыканий**

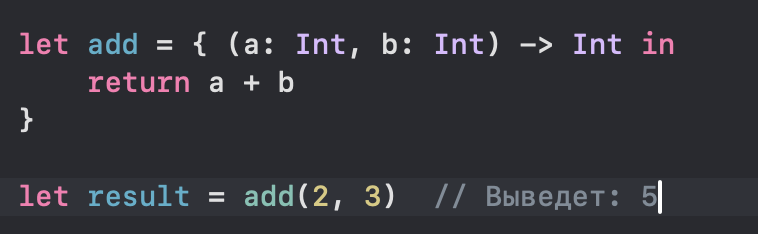
Замыкания могут выглядеть как обычные функции, только без имени. Вот пример простого замыкания:



greeting — это замыкание, которое просто выводит строку. Чтобы вызвать замыкание, нужно использовать круглые скобки, как с функцией.

**Замыкания с параметрами и возвращаемым значением**

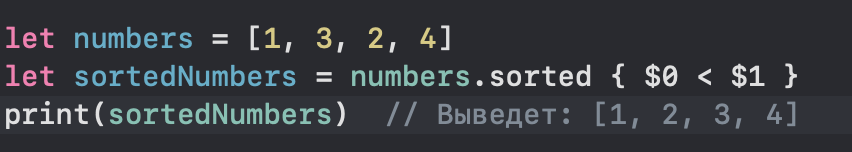
Замыкания могут также принимать параметры и возвращать значения. Их синтаксис будет немного сложнее:



Замыкание add принимает два параметра типа Int и возвращает их сумму.

**Замыкания как параметры функций**

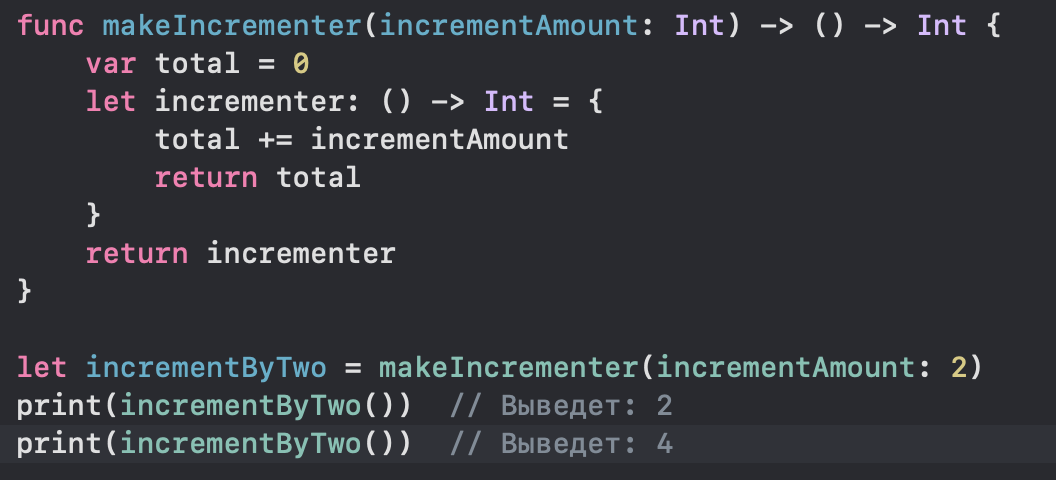
Замыкания очень часто используются в качестве параметров для других функций. Это полезно, когда нужно передать код для выполнения в будущем. Например, для сортировки массива мы можем использовать замыкания:



Здесь кложур (замыкание) передается методу sorted, который использует его для сортировки массива чисел.

**Захват значений**

Замыкания в Swift могут захватывать и сохранять значения из окружающего контекста. Это называется **замыкание захватывает значения**. Например, если замыкание используется в асинхронной задаче, оно может “запомнить” значения, которые были доступны в момент его создания.



incrementer захватывает значения переменных total и incrementAmount, и может использовать их при каждом вызове.

**Перечисления (Enum)**

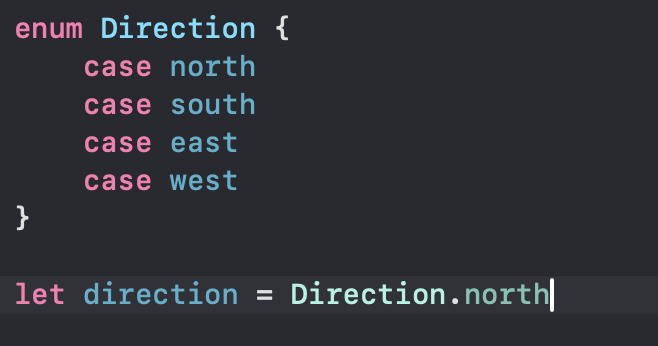
**Перечисления (enum)** в Swift — это способ организовать набор связанных значений в одну структуру. Перечисления позволяют задавать группы значений, которые могут быть использованы в разных частях программы. Каждый элемент перечисления называется **членом**.

Перечисления позволяют разработчику определять типы, которые могут принимать несколько различных значений. В Swift перечисления могут быть простыми или сложными. Они могут также содержать связанные значения.

**Пример простого перечисления**

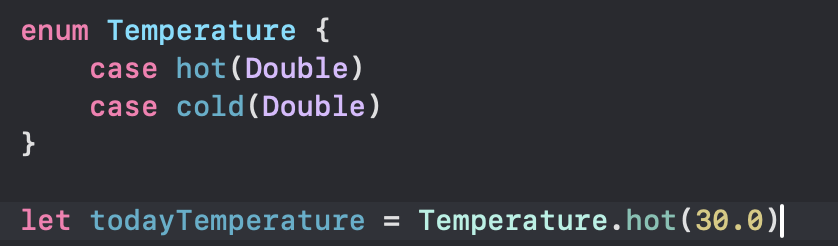
Простой пример перечисления с несколькими членами:

Перечисление Direction имеет четыре возможных значения: north, south, east, и west.



**Перечисление с ассоциированными значениями**

Перечисления в Swift могут также иметь **ассоциированные значения**. Это значит, что члены перечисления могут иметь дополнительные данные.



**Перечисления с методами**

Перечисления в Swift могут также иметь методы. Это позволяет создавать логику внутри перечисления, чтобы лучше организовывать код.



Для того, чтобы пришло полноценное понимание как пользоваться функциями/замыканиями/перечислениями и их тонкостями нужно практиковаться в написании разных программ. Попробуйте придумать программу, которая будет решать какую-то бытовую задачу для вас, к примеру подсчет оценок в семестре. Не забудьте про использование массивов и других типов вперемешку с функциями.

Эти инструменты — основа многих практических приложений, и их использование позволяет писать чистый, читаемый и поддерживаемый код.

**Классы/Структуры. Свойства. Свойства типа. Subscript**

Вы наверняка слышали про **объектно-ориентированное программирование (ООП)**. Swift поддерживает этот подход, и две ключевые конструкции, которые позволяют работать с объектами, — это **классы (class)** и **структуры (struct)**. А если не слышали про ООП, то обязательно ознакомьтесь с этим понятием, в дальнейшем это поможет вам лучше понимать работу сущностей вашего проекта и их взаимодействие друг с другом.

Обе эти сущности позволяют создавать **типы данных** с определёнными свойствами и методами. Однако у них есть важные различия, которые стоит понимать.

**Классы vs. Структуры: в чём разница?**

Классы и структуры очень похожи, но имеют **принципиальные отличия**:

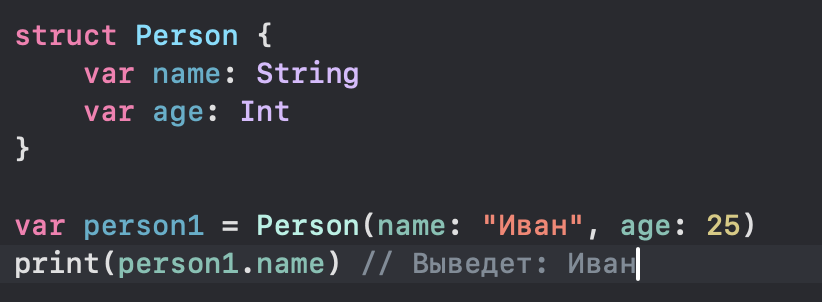
|  |  |  |
| --- | --- | --- |
| **Свойство** | **Классы (class)** | **Структуры (struct)** |
| **Передача данных** | Передаются по ссылке (reference type) | Передаются по значению (value type) |
| **Наследование** | Поддерживается | Не поддерживается |
| **Деинициализация** | Есть deinit | Отсутствует |
| **Модификация экземпляров внутри метода** | Необходимо указывать self | Нужно использовать mutating |

Если вы хотите создать **простой объект, который не требует наследования** и должен быть **эффективно передаваем по значению**, то выбирайте **структуру**. Если же объект сложный, требует **наследования и управления памятью**, стоит использовать **класс**.

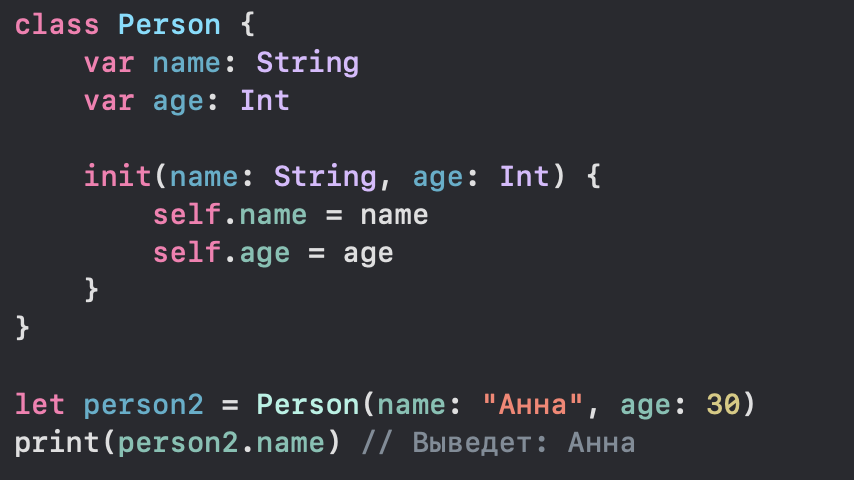
Наследование – один из принципов ООП (о чем мы говорили ранее). Он заключается в том, что если есть необходимость создать схожий класс, который будет реализовывать весь функционал «родителя» и иметь собственный (другой), то вы можете воспользоваться наследованием. Но нужно быть аккуратным в изменении родительского класса, так как его изменения скажутся на всех наследниках.

**Создание классов и структур**

Создать **структуру** (struct) в Swift можно следующим образом:



А вот аналогичный **класс** (class):

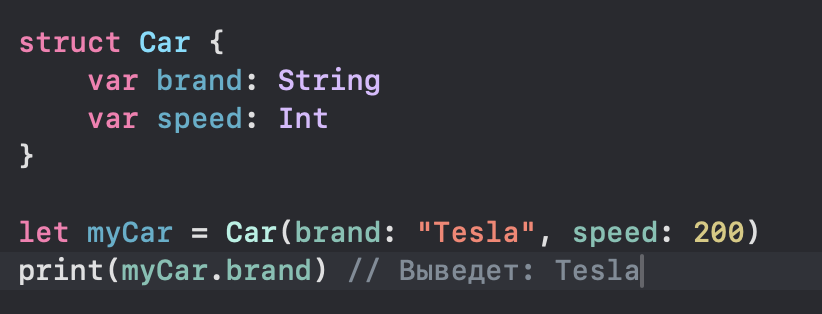


Заметили разницу? В **структуре** мы не использовали init, так как Swift автоматически создаёт инициализатор. В **классе** же мы вручную определили init, поскольку классы не имеют автогенерируемого инициализатора, если у них есть нестандартные свойства.

С правилами и особенностями инициализации более подробно мы ознакомимся позже.  
  
**Свойства классов и структур**

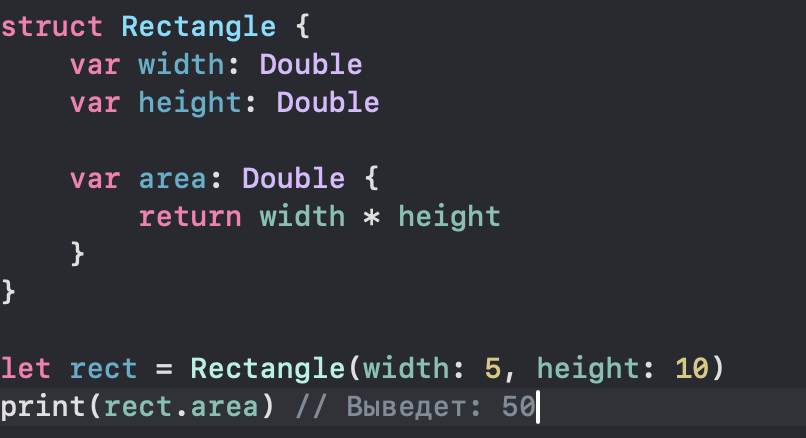
**Хранимые свойства (stored property)**

Они содержат конкретные значения, которые принадлежат экземпляру:



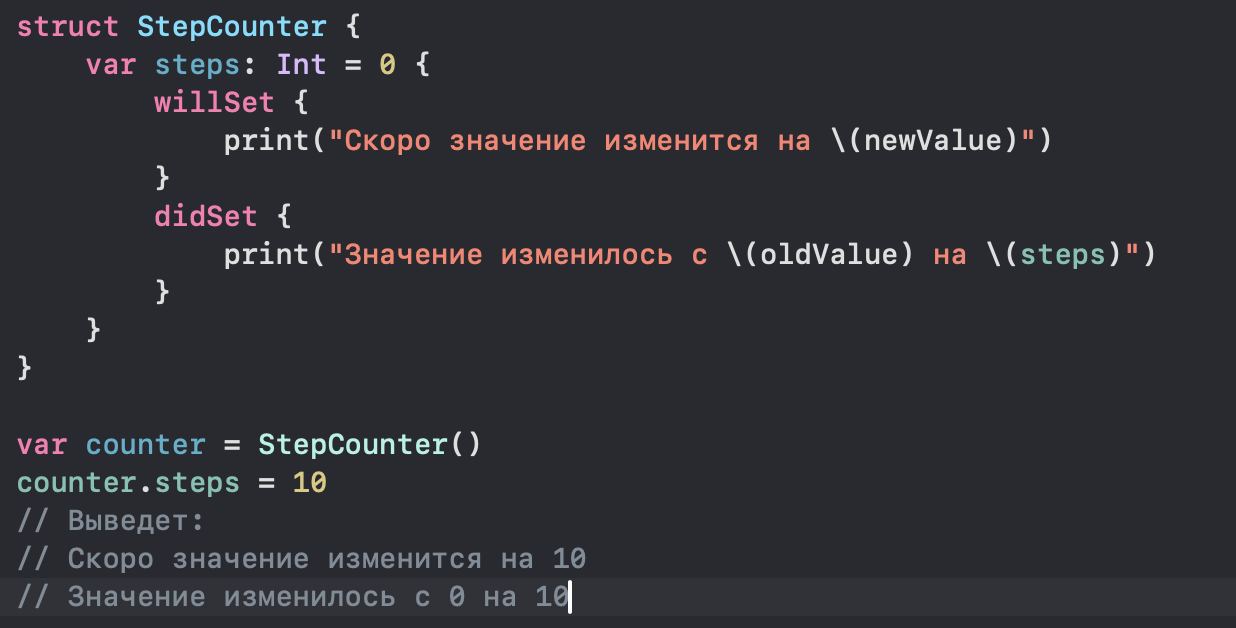
**Вычисляемые свойства (computed property)**

Эти свойства не хранят значение, а вычисляют его при каждом обращении (area):



**Свойства-наблюдатели (willSet и didSet)**

Позволяют реагировать на изменение значения свойства:

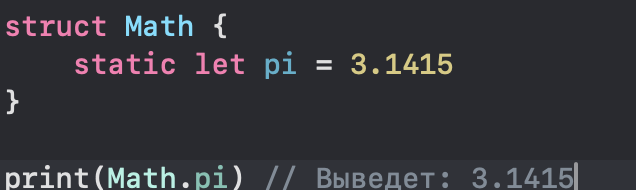


**Свойства типа (static и class)**

Бывает, что свойство относится **не к конкретному экземпляру**, а ко всему типу. Для этого используются **свойства типа**.

**Статические свойства (static)**

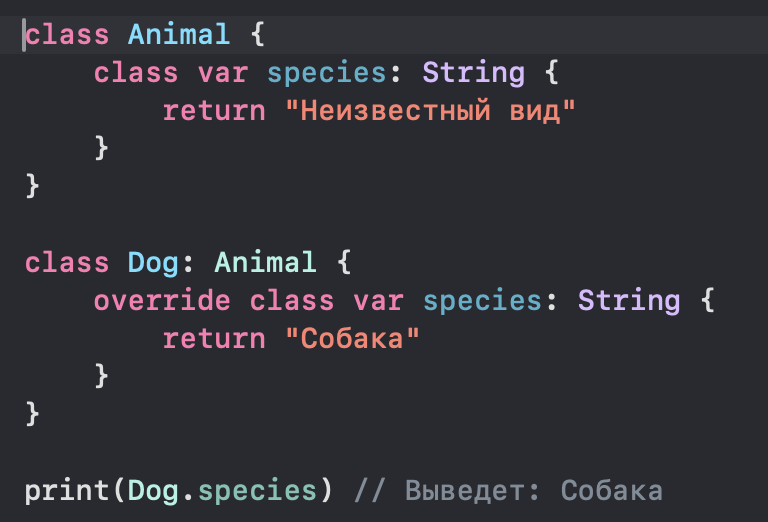
Для структур и классов можно объявлять **статические свойства**, которые принадлежат самому типу, а не его экземпляру:



Значение Math.pi принадлежит **всему типу**, а не его экземплярам.

**Свойства типа для классов (class)**

Ключевое слово class используется для **переопределяемых** свойств типа в классах:

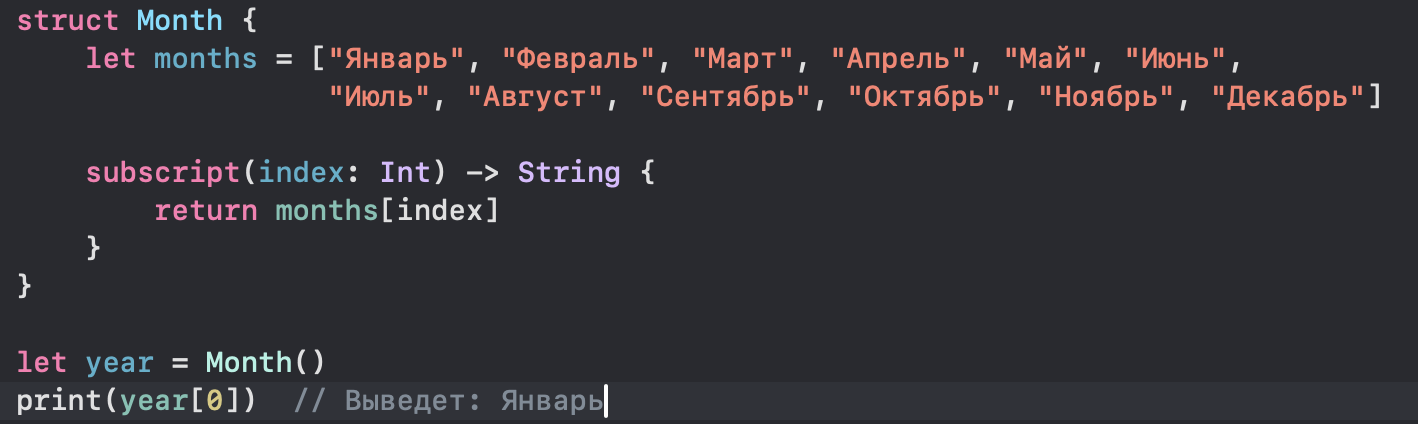


Если бы мы использовали static, переопределить species в Dog было бы невозможно.

**Subscript (сабскрипты)**

Сабскрипты позволяют обращаться к элементам объекта **по индексу**, как в массиве. Это удобно для создания собственных коллекций.

Простой пример — структура для хранения набора значений:



Сабскрипт позволяет обращаться к элементу структуры Month по индексу (year[0]), как если бы это был массив.

**Итого**

• **Классы и структуры** позволяют создавать пользовательские типы данных, но классы передаются по ссылке, а структуры — по значению.

• **Свойства бывают хранимыми, вычисляемыми и со свойствами-наблюдателями** (willSet / didSet).

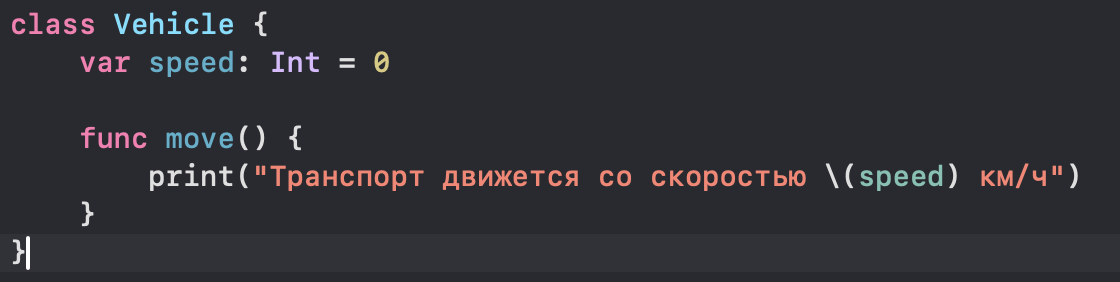
• **Свойства типа (**static **и** class**)** используются, когда значение должно принадлежать **всему типу, а не его экземпляру**.

• **Subscript (сабскрипты)** позволяют обращаться к данным объекта, как в массиве.

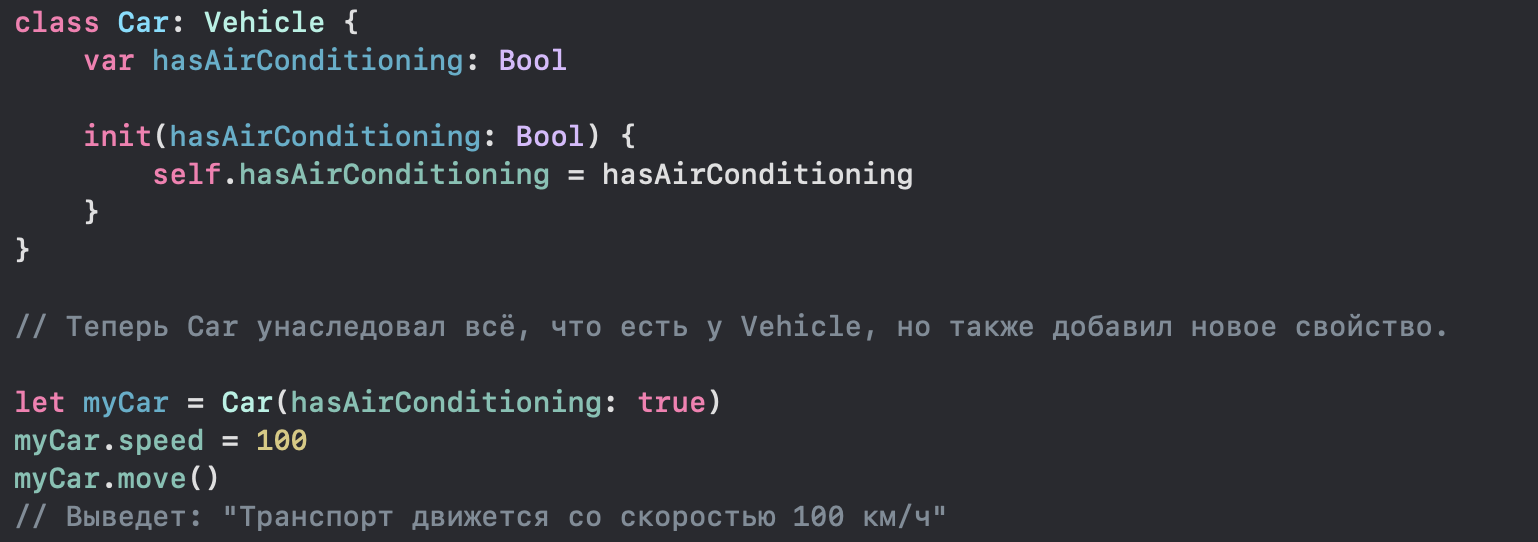
**Наследование. Опциональные цепочки. Приведение типов. Расширения**

Наследование позволяет одному классу **унаследовать свойства и методы другого класса**. Это мощный инструмент, который помогает избежать дублирования кода и создавать иерархию объектов.

Представьте, что у вас есть **базовый класс** Vehicle, который описывает общее поведение транспорта:

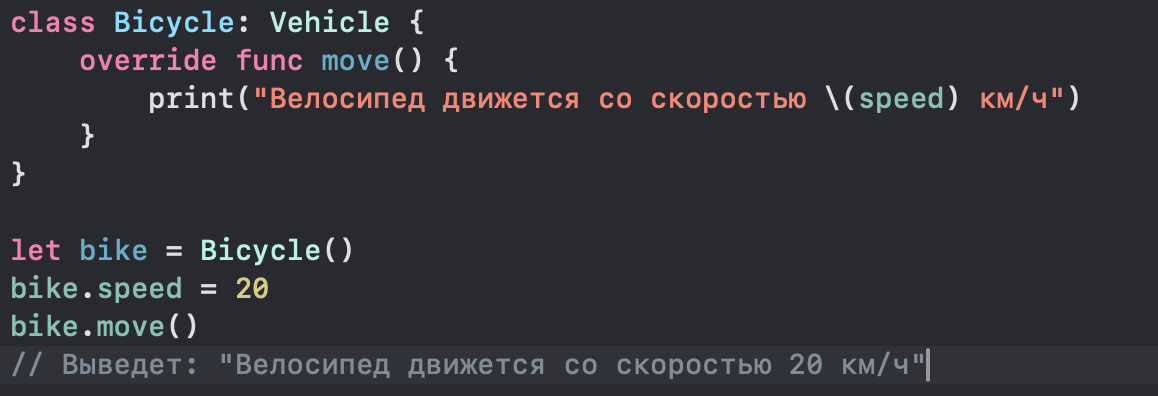


Теперь создадим класс Car, который будет наследоваться от Vehicle и добавим в него уникальное свойство hasAirConditioning:



**Переопределение методов (override)**

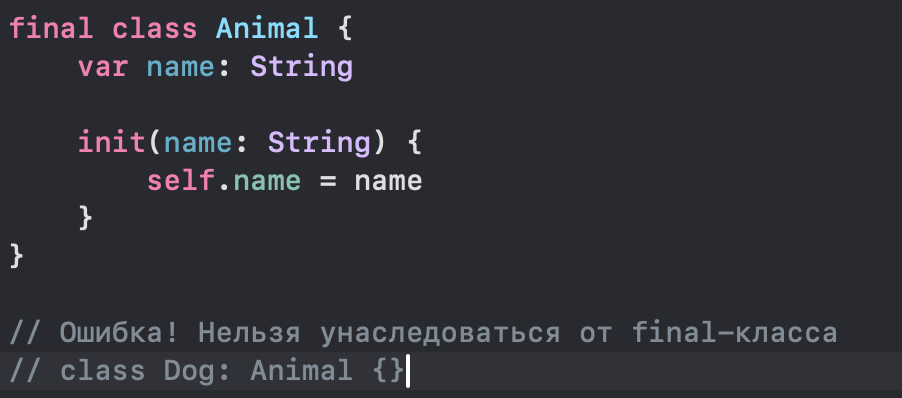
Если поведение метода в родительском классе не устраивает, его можно **переопределить** в дочернем классе, используя override:



Ключевое слово override говорит компилятору, что мы **изменяем метод из родительского класса**.

**Запрет на наследование (final)**

Иногда бывает нужно запретить дальнейшее наследование. В Swift для этого используется final:

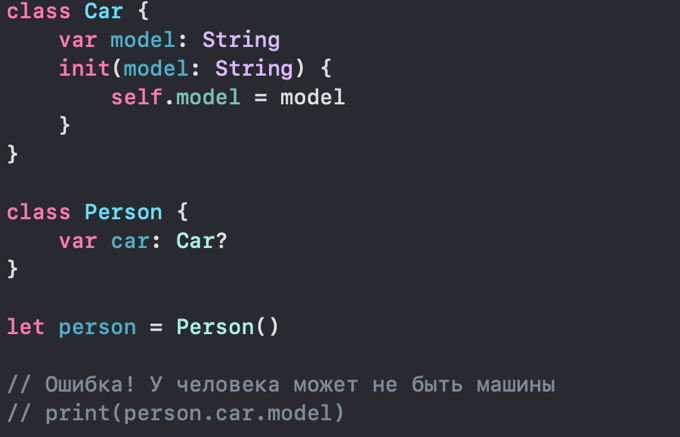


**Опциональные цепочки (Optional Chaining)**

Иногда приходится работать с **опциональными значениями**, которые могут быть nil. Например, если у объекта есть **свойство, которое само по себе является опциональным**, перед доступом к нему необходимо убедиться, что оно **не** nil.

**Проблема без опциональной цепочки**

Рассмотрим ситуацию, когда у класса Person есть Car, но **машины может и не быть**:



**Решение через Optional Chaining (?.)**

Чтобы избежать ошибки, можно использовать **опциональную цепочку** (?.):

**print(person.car?.model ?? "Машины нет")**

**// Выведет: "Машины нет"**

Если car **существует**, то программа выведет его model. Если car == nil, то выведется "Машины нет" благодаря оператору ??.

**Приведение типов (Type Casting)**

В ООП важно уметь работать с **разными типами данных**. Например, у нас может быть массив с **разными видами транспорта** (Car, Bicycle, Bus), и нам нужно проверить, какой объект перед нами.

В Swift для этого используются **операторы приведения типов**:

1. as? — **безопасное приведение** (возвращает nil, если приведение не удалось)

2. as! — **принудительное приведение** (краш, если приведение невозможно)

3. as — **явное приведение, если компилятор уже знает о типе**

Пример с as? и as!:

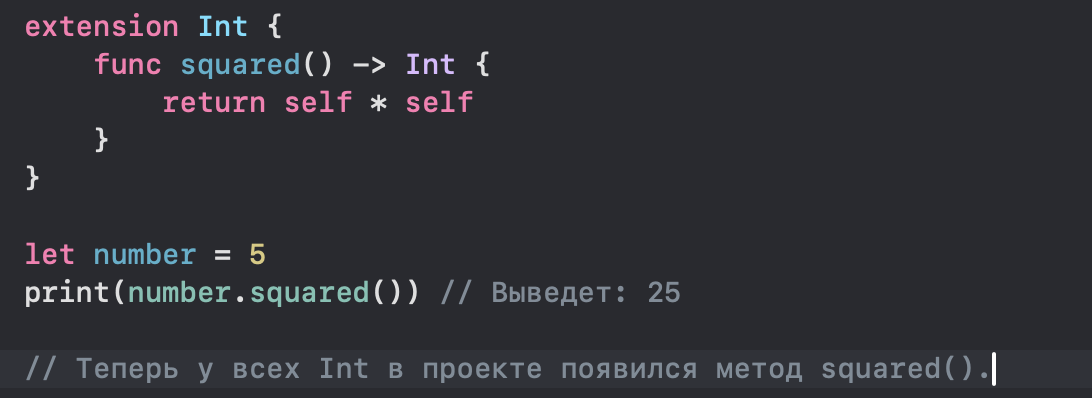


**Расширения (Extensions)**

Бывает, что у стандартного типа в Swift **не хватает какого-то метода**, но изменять исходный код мы не можем. В таких случаях нам помогают **расширения (**extension**)**.

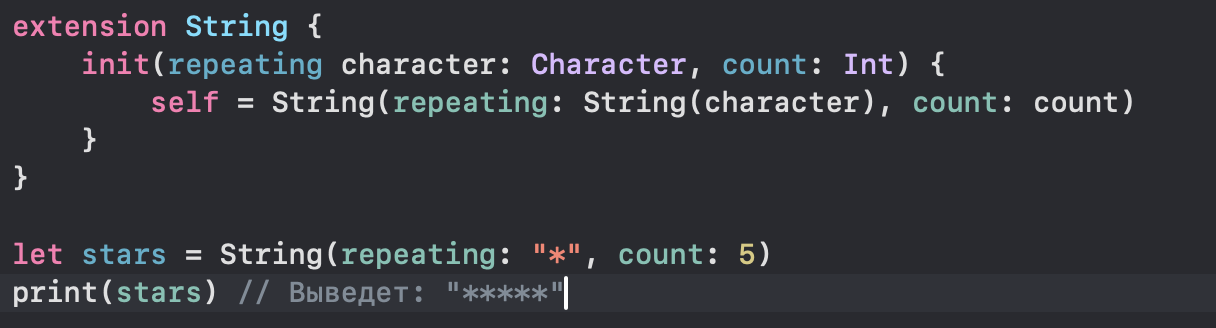
**Добавление метода через extension**

Допустим, нам хочется добавить новый метод squared() для типа Int:



**Добавление нового инициализатора**

Можно расширять **инициализаторы** существующих типов:



Позже мы изучим что такое диспетчеризация и подробнее рассмотрим работу расширений в разных ситуациях.

**В итоге**

1. **Наследование** позволяет создавать **иерархию классов**, переопределять методы (override) и запрещать наследование (final).

2. **Опциональные цепочки** (?.) используются для безопасного доступа к опциональным свойствам, когда возможен nil.

3. **Приведение типов** (as?, as!, as) позволяет работать с разными типами данных.

4. **Расширения (**extension**)** помогают добавлять **новые методы, инициализаторы и поведение** к существующим типам без изменения их исходного кода.

**Дженерики, протоколы и делегаты в Swift**

В программировании важно, чтобы код был **гибким, переиспользуемым и легко расширяемым**. Swift предлагает несколько мощных инструментов для достижения этой цели, среди которых **дженерики, протоколы и делегаты**.

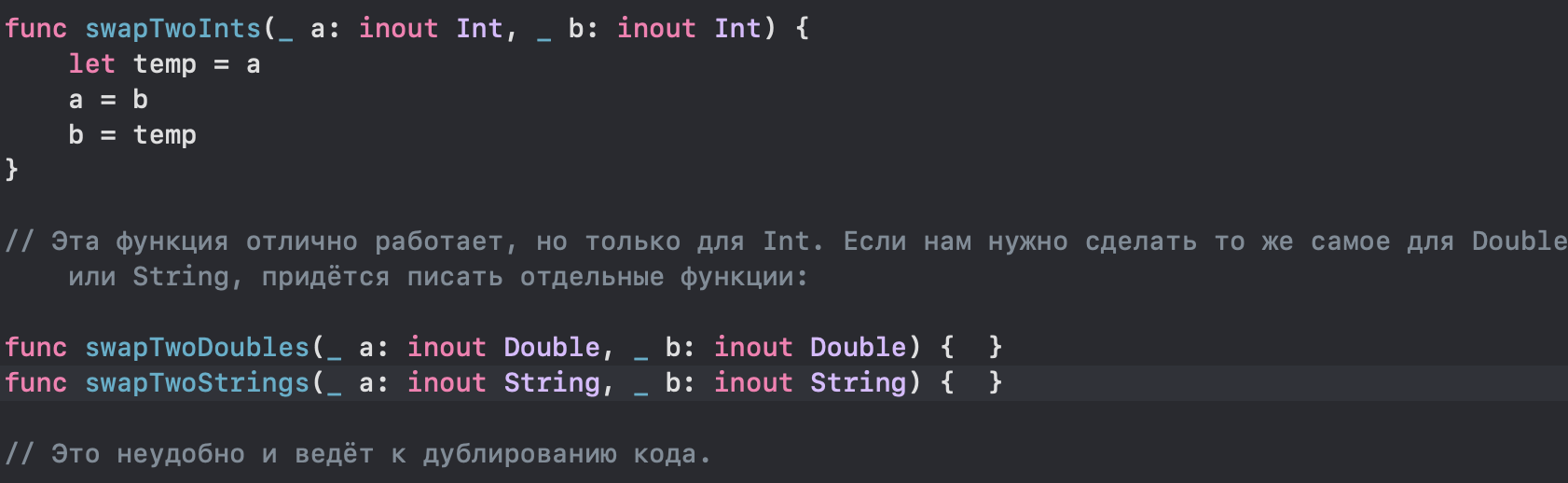
Эти концепции могут показаться сложными для новичков, но давайте разберём их **понятно и подробно**.

**Дженерики (Generics) — универсальность кода**

При разработке программ часто бывает, что одни и те же алгоритмы или структуры данных должны работать **с разными типами данных**. Вместо того чтобы дублировать код для каждого типа, Swift позволяет использовать **дженерики** — механизм, который делает код **гибким и многоразовым**.

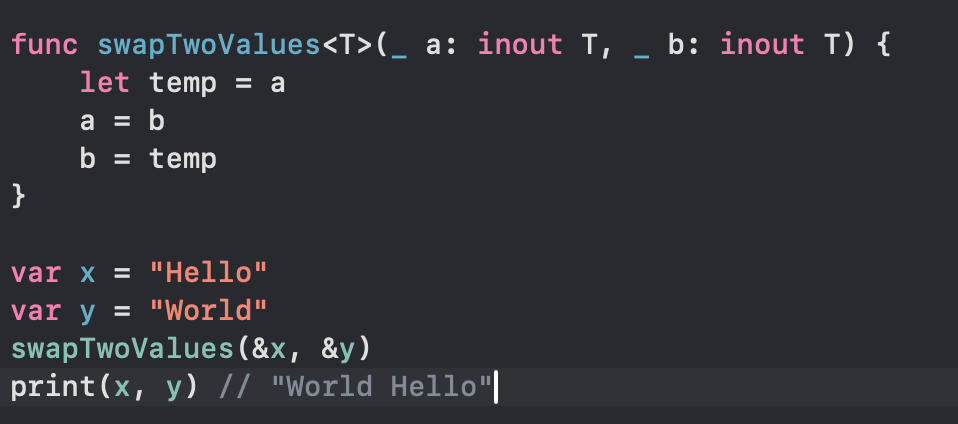
**Проблема без дженериков**

Допустим, у нас есть функция, которая меняет местами два числа:



**Решение с дженериками**

Используя **дженерики**, мы можем написать **одну универсальную функцию**, которая будет работать с любым типом:



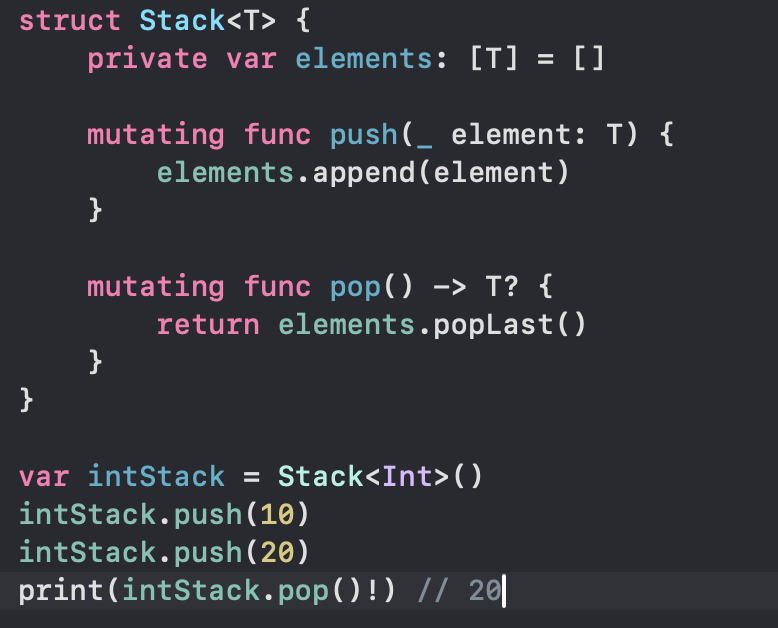
T — это **обобщённый (generic) параметр**, который обозначает **любой тип**. Он определяется в **угловых скобках** <T>.

Теперь функция **работает с любыми типами данных**, включая Int, Double, String и даже **пользовательские структуры**.

**Дженериковые структуры и классы**

Дженерики можно применять **не только к функциям**, но и к **структурам и классам**.

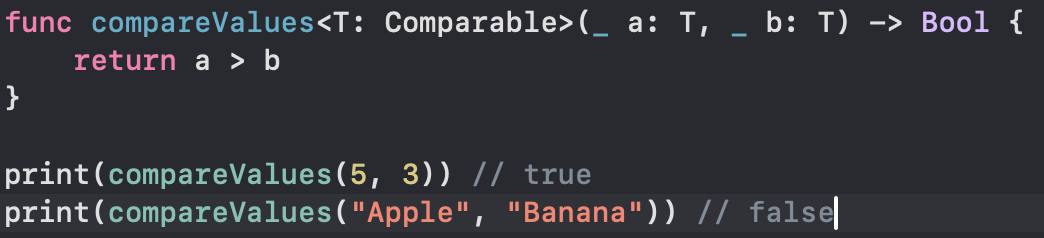
Пример: создадим **дженериковый стек** — структуру, которая хранит элементы любого типа:



Теперь Stack<T> можно использовать **для любых типов данных**, а не только Int.

**Ограничения типов (where и :)**

Бывает, что нам нужно **ограничить** возможные типы, например, использовать только те, которые поддерживают сравнение (> и <). Для этого применяются **ограничения (**where **или** :**)**:



Здесь <T: Comparable> указывает, что **T должен поддерживать сравнение**.

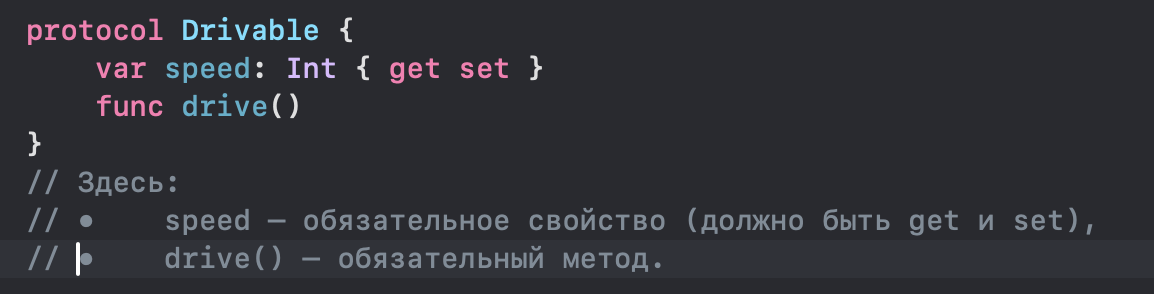
**Протоколы (Protocols) — контракты поведения**

Протоколы позволяют **описать набор требований**, которым должен соответствовать класс, структура или перечисление.

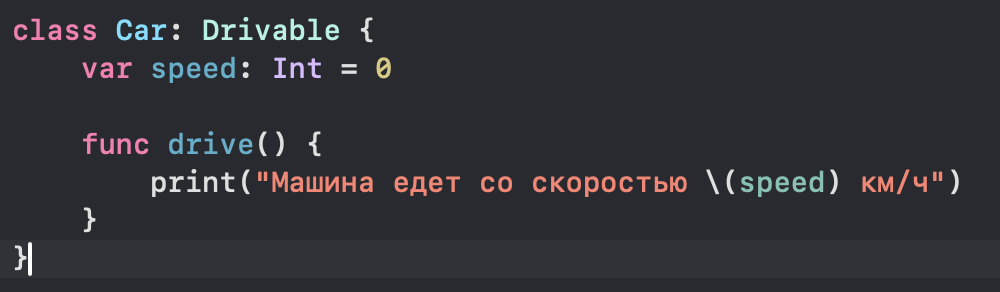
- Если класс **подписывается на протокол**, он **обязан** реализовать все его требования.

- Протокол можно сравнить с **интерфейсом**, если вы знакомы с другими языками программирования.

**Объявление протокола**



**Реализация протокола в классе**



Если мы **не реализуем** drive(), **компилятор выдаст ошибку**.

**Протоколы как типы**

Вы можете объявить переменную **типа протокола**:

var vehicle: Drivable = Car()

vehicle.speed = 120

vehicle.drive() // "Машина едет со скоростью 120 км/ч"

**Делегаты (Delegates) — передача ответственности**

Делегирование — это **передача ответственности от одного объекта к другому**.

Этот механизм активно используется в iOS-разработке, например:

• UITableViewDataSource — отвечает за данные таблицы,

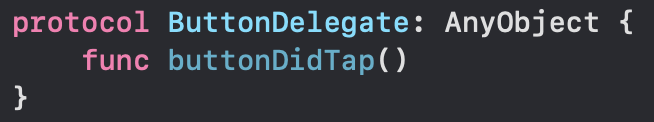
• URLSessionDelegate — управляет сетевыми запросами,

• **Собственные делегаты** для связи между объектами.

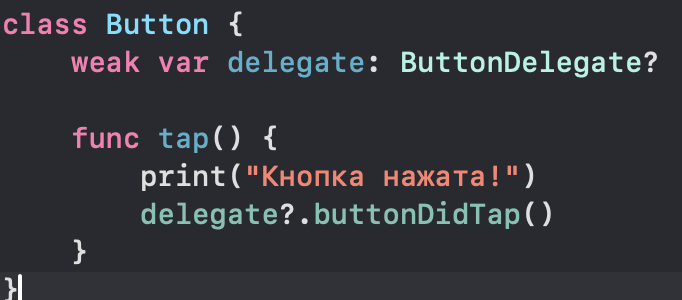
**Пример собственного делегата**

Допустим, у нас есть Button, который **должен уведомлять кого-то при нажатии**.

**1. Объявляем протокол**:

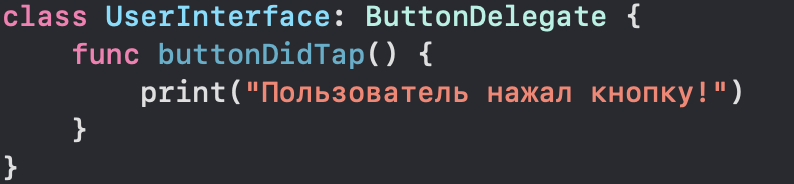


**2. Создаём класс Button, который будет оповещать делегата:**



weak var delegate — предотвращает **циклы удержания** (retain cycle). (об этом позже)

**3. Класс UserInterface становится делегатом:**



**4. Настраиваем делегирование:**



Теперь при вызове button.tap(), **кнопка оповещает делегата** (ui)

**Резюмирование пройденных тем**

Мы подробно разобрали **основные концепции языка Swift**, которые формируют фундамент для дальнейшего изучения. Давайте обобщим ключевые моменты и вспомним, что именно мы узнали.

**Переменные, константы и базовые типы**

**Переменные (**var**)** позволяют изменять значение, а **константы (**let**)** — нет.

Swift имеет **сильную типизацию**, что помогает избежать ошибок.

Базовые типы включают:

• Int — целые числа,

• Float и Double — числа с плавающей запятой,

• Bool — логические значения true/false,

• Character и String — работа с текстом.

Строки (String) в Swift **являются коллекциями символов**, поддерживают удобную интерполяцию "Hello, \(name)!" и обладают множеством методов для работы с текстом.

**Коллекции: массивы, словари, множества, кортежи**

**- Массивы (**Array**)** — упорядоченные коллекции элементов, доступ к которым осуществляется по индексу.

**- Словари (**Dictionary**)** — неупорядоченные коллекции, хранящие пары “ключ-значение”.

**- Множества (**Set**)** — неупорядоченные коллекции уникальных элементов.

**- Кортежи (**Tuple**)** позволяют объединять разные типы данных в одну переменную (let person = ("Alex", 25)).

**Опциональные типы и управляющие конструкции**

**- Опционалы (**Optional**)** — способ работы с переменными, которые могут быть nil.

**- Unwrapping (**!**,** if let**,** guard let**)** используется для безопасного извлечения значений из - опционалов.

**- Switch/case** — мощная альтернатива if/else, поддерживающая сложные проверки.

**- Циклы** (for-in, while, repeat-while) позволяют организовывать повторяющиеся действия.

Операторы **break и continue** управляют потоком выполнения в циклах.

**Функции, замыкания и перечисления (enum)**

- **Функции (**func**)** позволяют разбить код на переиспользуемые части, поддерживают входные параметры и возвращаемые значения.

- **Замыкания (**Closures**)** — блоки кода, которые можно передавать в качестве аргументов или хранить в переменных.

- **Перечисления (**enum**)** позволяют объявлять типы с ограниченным набором значений. Они могут содержать ассоциированные значения и поддерживают **связанную логику**.

**Классы и структуры. Свойства. Subscript**

- **Классы (**class**)** и **структуры (**struct**)** — основные строительные блоки для создания объектов.

- Структуры **передаются по значению**, а классы **по ссылке**.

- **Свойства** хранят данные в объектах, бывают:

• **Свойства экземпляра (**var name: String**)**,

• **Свойства типа (**static var counter: Int = 0**)**,

• **Вычисляемые свойства (**var fullName: String { firstName + " " + lastName }**)**.

- **Subscript** позволяет работать с объектами как с массивами (matrix[0, 1]).

**Наследование, приведение типов, расширения**

- **Наследование** (class Child: Parent) позволяет создавать новые классы на основе существующих.

- **Опциональные цепочки (**?.**)** помогают безопасно обращаться к свойствам объектов, которые могут быть nil.

- **Приведение типов (**as?**,** as!**,** is**)** используется для работы с разными типами.

- **Расширения (**extension**)** позволяют добавлять новые свойства и методы к существующим типам без изменения их кода.

**Дженерики, протоколы, делегаты**

- **Дженерики (**Generics**)** позволяют создавать универсальные функции и структуры, работающие с разными типами (func swap<T>(\_ a: inout T, \_ b: inout T)).

- **Протоколы (**Protocols**)** определяют **контракт**, который обязаны соблюдать подписавшиеся под него классы или структуры.

- **Делегаты (**Delegates**)** — механизм передачи ответственности между объектами, активно используемый в iOS-разработке.

**Вывод**

Теперь у вас есть твёрдая основа в языке Swift. Мы разобрали ключевые принципы работы с типами данных, коллекциями, управляющими конструкциями, функциями, объектно-ориентированным программированием, дженериками и протоколами. Эти знания формируют **фундамент для дальнейшего изучения разработки iOS-приложений**.

Однако язык программирования сам по себе — это лишь инструмент. Для создания реальных приложений нам нужны **фреймворки и библиотеки**, предоставляющие удобные механизмы для работы с пользовательским интерфейсом, сетью, базами данных и многим другим.

**Следующий этап — знакомство с UIKit и Foundation**, двумя важнейшими фреймворками, без которых невозможно представить разработку iOS-приложений. Мы разберём:

• Как строить интерфейсы на основе UIView и UIViewController;

• Как управлять навигацией и жизненным циклом приложения;

• Как работать с сетевыми запросами, файлами и многопоточностью с помощью Foundation.