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.

```
var userName = "Алексей"   // Переменная, можно изменить
userName = "Дмитрий"       // ✅ Можно изменить значение

let birthYear = 1995       // Константа
// birthYear = 2000        ❌ Ошибка! Константу изменить нельзя
```

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:

```
var age: Int = 25          // Явно указываем, что это Int
let pi: Double = 3.1415    // Явно указываем Double
var city = "Москва"        // Swift сам определит, что это String
```

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

```
var myAge = 30
var first_name = "Алексей" // Можно использовать нижнее подчёркивание (но в Swift это не принято)
var 🚀 = "Swift"           // Работает, но так не стоит делать
```

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:

```
let number1: Int = -42  // Обычное целое число
let number2: UInt = 100 // Только положительное
let number: UInt = -5  // Ошибка! UInt не поддерживает отрицательные числа
```

Явное указание размера (если нужно)
Если важно контролировать размер, можно использовать Int8, Int16, Int32, Int64:

```
let smallNumber: Int8 = 127     // От -128 до 127
let largeNumber: Int64 = 9223372036854775807 // Большое число
```

• 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.

```
let floatNumber: Float = 3.1415926535  // Округлит до 3.141593
let doubleNumber: Double = 3.1415926535 // Сохранит всю точность
```

Опасность округления (Float vs Double)

```
let a: Float = 0.1 + 0.2
print(a) // Выведет 0.30000001 из-за потери точности

let b: Double = 0.1 + 0.2
print(b) // Выведет 0.3 (более точное значение)
```

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:

```swift
let isSwiftAwesome = true
if isSwiftAwesome {
    print("Swift — отличный язык!")
}
```

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

```swift
let check = 1  // Это просто Int, не Bool
if check { }   // ❌ Ошибка! Int не может быть Bool
```

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):

```swift
let greeting: String = "Привет, мир!"
// Swift автоматически определяет тип String, поэтому можно писать без : String:
let message = "Добро пожаловать!"
```

Пустые строки
Чтобы создать пустую строку, есть два способа:

```swift
var emptyString1 = ""         // Пустая строка
var emptyString2 = String()   // Используем инициализатор

print(emptyString1.isEmpty)  // true
print(emptyString2.isEmpty)  // true
```

Метод .isEmpty проверяет, пустая ли строка.

Конкатенация строк (сложение строк)
В Swift можно объединять строки с помощью + или +=:

```
let firstName = "Алексей"
let lastName = "Смицкий"

let fullName = firstName + " " + lastName
print(fullName) // "Алексей Смицкий"

var phrase = "Swift "
phrase += "— лучший язык!"
print(phrase) // "Swift — лучший язык!"
```

+ 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):

```
let language = "Swift"
let version = 5.9

print("Я изучаю \(language), версия \(version)")
// "Я изучаю Swift, версия 5.9"
```

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:

```
//•    Пустой массив:
var numbers: [Int] = []
//•    Массив с начальными значениями:
let fruits = ["яблоко", "банан", "груша"]
//•    Массив с одинаковыми значениями (используя повторение):
let repeatedNumbers = Array(repeating: 0, count: 5)  // [0, 0, 0, 0, 0]
```

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:

```
var numbers = [1, 2, 3]
numbers.append(4)  // Добавляем элемент в конец массива
numbers[0] = 10    // Изменяем первый элемент массива
print(numbers)     // [10, 2, 3, 4]
```

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.

```
//•    Пустой словарь:
var personInfo: [String: String] = [:]  // Пустой словарь с ключами и значениями типа String
//•    Словарь с начальными значениями:
let person = ["name": "Алексей", "age": "25"]
```

Доступ к элементам словаря
Чтобы получить значение по ключу, используем квадратные скобки:

```
let person = ["name": "Алексей", "age": "25"]
print(person["name"]!)  // "Алексей"
// Обрати внимание, что если ключ не существует, результат будет nil. Поэтому
    лучше использовать опциональное связывание:
if let age = person["age"] {
    print(age)  // "25"
} else {
    print("Возраст не указан")
}
```

Добавление и изменение значений
Чтобы добавить или изменить элемент в словаре, используем ключ:

```
var person = ["name": "Алексей", "age": "25"]
person["age"] = "26"  // Изменение значения
person["city"] = "Москва"  // Добавление нового ключа
print(person)  // ["name": "Алексей", "age": "26", "city": "Москва"]
```

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

```
//•    Пустое множество:
var uniqueNumbers: Set<Int> = []
//•    Множество с начальными значениями:
let fruits: Set = ["яблоко", "банан", "груша", "яблоко"]
```

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:

```
var numbers: Set = [1, 2, 3]
numbers.insert(4)   // Добавляем элемент
numbers.remove(2)   // Удаляем элемент
print(numbers)      // [1, 3, 4]
```

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:

```
var name: String? = "Алексей"
name = nil
```

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.

```
var name: String? = "Алексей"

if let unwrappedName = name {
    print("Имя: \(unwrappedName)")
} else {
    print("Имя не задано")
}
```

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.

```
var name: String? = "Алексей"
print(name!)  // "Алексей"
```

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:

```
let number = 2

switch number {
case 1:
    print("Это единица")
case 2:
    print("Это двойка")
case 3:
    print("Это тройка")
default:
    print("Неизвестное число")
}
```

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:

```
let score = 85

switch score {
case 90...100:
    print("Отлично!")
case 75...89:
    print("Хорошо")
case 50...74:
    print("Удовлетворительно")
default:
    print("Не сдал")
}
```

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

```
let age = 18

if age >= 18 {
    print("Ты совершеннолетний!")
} else {
    print("Ты еще несовершеннолетний.")
}
```

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:

```
let temperature = 25

if temperature > 30 {
    print("Очень жарко")
} else if temperature > 20 {
    print("Тепло")
} else {
    print("Холодно")
}
```

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

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

for fruit in fruits {
    print(fruit)
}
```

Этот код выведет каждый элемент массива поочередно.

Пример с диапазоном
Можно также использовать диапазоны для перебора чисел:

```
for i in 1...5 {
    print(i)
}
// Здесь мы перебираем числа от 1 до 5 и выводим их.
```

While

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

Пример с циклом while

```
var count = 0

while count < 5 {
    print("Значение счётчика: \(count)")
    count += 1  // увеличиваем счётчик
}
// Этот цикл будет выполняться до тех пор, пока переменная count меньше 5.
```

Break и Return

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

```
for i in 1...10 {
    if i == 5 {
        break  // Прерываем цикл, когда i равно 5
    }
    print(i)
}
// Выведет: 1, 2, 3, 4
```

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

```
func greet(name: String) -> String {
    return "Привет, \(name)!"
}

let greeting = greet(name: "Алексей")
print(greeting)  // Привет, Алексей!
```

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:

```
func имя_функции(параметры) -> тип_возвращаемого_значения {
    // Тело функции
}
// Пример функции, которая ничего не возвращает:
func greet(name: String) {
    print("Привет, \(name)!")
}
```

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:

```
func sum(a: Int, b: Int) -> Int {
    return a + b
}

let result = sum(a: 5, b: 3)  // Выведет: 8
```

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.

```
func greet(name: String, message: String = "Привет") {
    print("\(message), \(name)!")
}

greet(name: "Алексей")          // Выведет: Привет, Алексей!
greet(name: "Алексей", message: "Здравствуй")  // Выведет: Здравствуй, Алексей!
```

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:

```
func multiply(a: Int, b: Int, c: Int) -> Int {
    return a * b * c
}

let product = multiply(a: 2, b: 3, c: 4)  // Выведет: 24
```

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:

```swift
let greeting = {
    print("Привет, мир!")
}

greeting()  // Выведет: Привет, мир!
```

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:

```swift
let add = { (a: Int, b: Int) -> Int in
    return a + b
}

let result = add(2, 3)  // Выведет: 5
```

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:

```swift
let numbers = [1, 3, 2, 4]
let sortedNumbers = numbers.sorted { $0 < $1 }
print(sortedNumbers)  // Выведет: [1, 2, 3, 4]
```

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.

```swift
func makeIncrementer(incrementAmount: Int) -> () -> Int {
    var total = 0
    let incrementer: () -> Int = {
        total += incrementAmount
        return total
    }
    return incrementer
}

let incrementByTwo = makeIncrementer(incrementAmount: 2)
print(incrementByTwo())  // Выведет: 2
print(incrementByTwo())  // Выведет: 4
```

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.

```swift
enum Direction {
    case north
    case south
    case east
    case west
}

let direction = Direction.north
```

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

```swift
enum Temperature {
    case hot(Double)
    case cold(Double)
}

let todayTemperature = Temperature.hot(30.0)
```

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

```swift
enum Direction {
    case north, south, east, west

    func description() -> String {
        switch self {
        case .north:
            return "Север"
        case .south:
            return "Юг"
        case .east:
            return "Восток"
        case .west:
            return "Запад"
        }
    }
}

let direction = Direction.north
print(direction.description())  // Выведет: Север
```

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:

```
struct Person {
    var name: String
    var age: Int
}

var person1 = Person(name: "Иван", age: 25)
print(person1.name) // Выведет: Иван
```

А вот аналогичный **класс** (class):

```
class Person {
    var name: String
    var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

let person2 = Person(name: "Анна", age: 30)
print(person2.name) // Выведет: Анна
```

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:

```
struct Car {
    var brand: String
    var speed: Int
}

let myCar = Car(brand: "Tesla", speed: 200)
print(myCar.brand) // Выведет: Tesla
```

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

```
struct Rectangle {
    var width: Double
    var height: Double

    var area: Double {
        return width * height
    }
}

let rect = Rectangle(width: 5, height: 10)
print(rect.area) // Выведет: 50
```

Свойства-наблюдатели (willSet и didSet)
Позволяют реагировать на изменение значения свойства:

```
struct StepCounter {
    var steps: Int = 0 {
        willSet {
            print("Скоро значение изменится на \(newValue)")
        }
        didSet {
            print("Значение изменилось с \(oldValue) на \(steps)")
        }
    }
}

var counter = StepCounter()
counter.steps = 10
// Выведет:
// Скоро значение изменится на 10
// Значение изменилось с 0 на 10
```

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:

```
struct Math {
    static let pi = 3.1415
}

print(Math.pi) // Выведет: 3.1415
```

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:

```
class Animal {
    class var species: String {
        return "Неизвестный вид"
    }
}

class Dog: Animal {
    override class var species: String {
        return "Собака"
    }
}

print(Dog.species) // Выведет: Собака
```

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:

```
struct Month {
    let months = ["Январь", "Февраль", "Март", "Апрель", "Май", "Июнь",
                  "Июль", "Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"]

    subscript(index: Int) -> String {
        return months[index]
    }
}

let year = Month()
print(year[0])  // Выведет: Январь
```

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:

```
class Vehicle {
    var speed: Int = 0

    func move() {
        print("Транспорт движется со скоростью \(speed) км/ч")
    }
}
```

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

```
class Car: Vehicle {
    var hasAirConditioning: Bool

    init(hasAirConditioning: Bool) {
        self.hasAirConditioning = hasAirConditioning
    }
}

// Теперь Car унаследовал всё, что есть у Vehicle, но также добавил новое свойство.

let myCar = Car(hasAirConditioning: true)
myCar.speed = 100
myCar.move()
// Выведет: "Транспорт движется со скоростью 100 км/ч"
```

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

```swift
class Bicycle: Vehicle {
    override func move() {
        print("Велосипед движется со скоростью \(speed) км/ч")
    }
}

let bike = Bicycle()
bike.speed = 20
bike.move()
// Выведет: "Велосипед движется со скоростью 20 км/ч"
```

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:

```swift
final class Animal {
    var name: String

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

// Ошибка! Нельзя унаследоваться от final-класса
// class Dog: Animal {}
```

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**:

```swift
class Car {
    var model: String
    init(model: String) {
        self.model = model
    }
}

class Person {
    var car: Car?
}

let person = Person()

// Ошибка! У человека может не быть машины
// print(person.car.model)
```

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!:

```
class Animal {
    var name: String
    init(name: String) {
        self.name = name
    }
}

class Dog: Animal {
    func bark() {
        print("Гав-гав!")
    }
}

let pet: Animal = Dog(name: "Шарик")

if let dog = pet as? Dog {
    dog.bark() // Выведет: "Гав-гав!"
}

// Принудительное приведение (опасно!)
// let dog = pet as! Dog
// dog.bark()
```

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:

```
extension Int {
    func squared() -> Int {
        return self * self
    }
}

let number = 5
print(number.squared()) // Выведет: 25

// Теперь у всех Int в проекте появился метод squared().
```

Добавление нового инициализатора
Можно расширять **инициализаторы** существующих типов:

```
extension String {
    init(repeating character: Character, count: Int) {
        self = String(repeating: String(character), count: count)
    }
}

let stars = String(repeating: "*", count: 5)
print(stars) // Выведет: "*****"
```

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:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temp = a
    a = b
    b = temp
}

// Эта функция отлично работает, но только для Int. Если нам нужно сделать то же самое для Double
    или String, придётся писать отдельные функции:

func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {  }
func swapTwoStrings(_ a: inout String, _ b: inout String) {  }

// Это неудобно и ведёт к дублированию кода.
```

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

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temp = a
    a = b
    b = temp
}

var x = "Hello"
var y = "World"
swapTwoValues(&x, &y)
print(x, y) // "World Hello"
```

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:

```
struct Stack<T> {
    private var elements: [T] = []

    mutating func push(_ element: T) {
        elements.append(element)
    }

    mutating func pop() -> T? {
        return elements.popLast()
    }
}

var intStack = Stack<Int>()
intStack.push(10)
intStack.push(20)
print(intStack.pop()!) // 20
```

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** :**)**:

```
func compareValues<T: Comparable>(_ a: T, _ b: T) -> Bool {
    return a > b
}

print(compareValues(5, 3)) // true
print(compareValues("Apple", "Banana")) // false
```

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

```
protocol Drivable {
    var speed: Int { get set }
    func drive()
}
// Здесь:
// •    speed — обязательное свойство (должно быть get и set),
// •    drive() — обязательный метод.
```

Реализация протокола в классе

```
class Car: Drivable {
    var speed: Int = 0

    func drive() {
        print("Машина едет со скоростью \(speed) км/ч")
    }
}
```

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:

```
protocol ButtonDelegate: AnyObject {
    func buttonDidTap()
}
```

2. Создаём класс Button, который будет оповещать делегата:

```
class Button {
    weak var delegate: ButtonDelegate?

    func tap() {
        print("Кнопка нажата!")
        delegate?.buttonDidTap()
    }
}
```

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

3. Класс UserInterface становится делегатом:

```
class UserInterface: ButtonDelegate {
    func buttonDidTap() {
        print("Пользователь нажал кнопку!")
    }
}
```

4. Настраиваем делегирование:

```
let button = Button()
let ui = UserInterface()

button.delegate = ui
button.tap()

// Выведет:
// "Кнопка нажата!"
// "Пользователь нажал кнопку!"
```

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.