





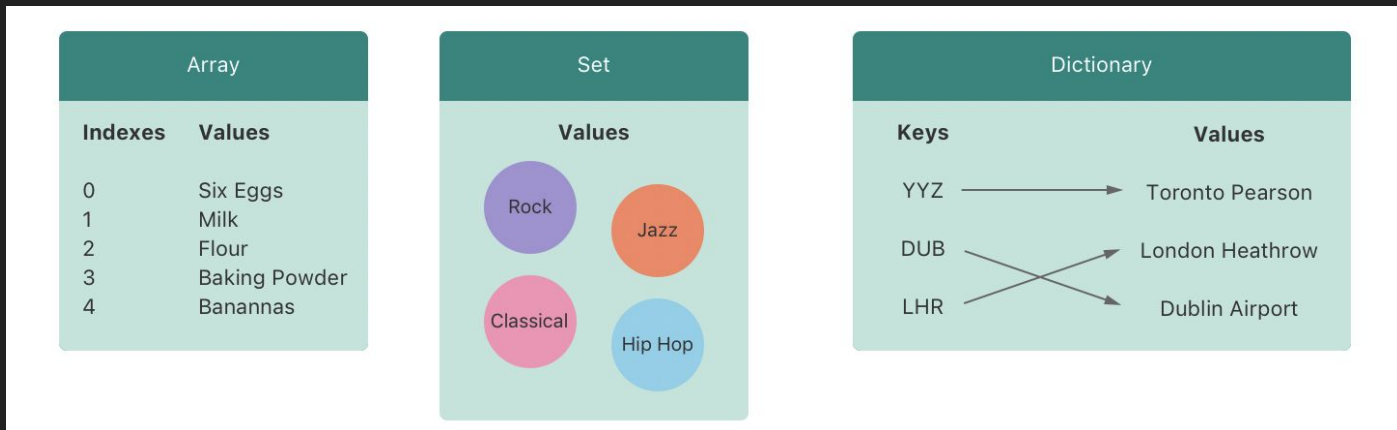
Содержание

- ★ Массивы (Array) 
- ★ Сеты (Set) 
- ★ Словари (Dictionary) 
- ★ Условия (If - Else, Switch) ?
- ★ Циклы (For - loop, While - loop) 

Типы коллекций

Swift предоставляет **три основных типа коллекций**, известных как **массивы**, **сеты** и **словари**, для хранения данных.

- ❑ **Массивы** — это **упорядоченные** наборы значений
- ❑ **Сеты** — это **неупорядоченные** наборы **уникальных** значений
- ❑ **Словари** — это **неупорядоченные** наборы **ассоциаций** **ключ-значение**



Изменяемость коллекций

Если вы создаете **массив**, **сет** или **словарь** и присваиваете их переменной, созданная коллекция будет **изменяемой**. Это означает, что вы можете изменить (или видоизменить) коллекцию после ее создания, **добавив**, **удалив** или **изменив** элементы в коллекции.

Если вы присваиваете константе **массив**, **сет** или **словарь**, эта коллекция становится **неизменной**, а ее размер и содержимое нельзя изменить.

Массивы (Arrays)

- ❑ **Массив** хранит значения **одного типа в упорядоченном списке**
- ❑ Одно и то же значение может появляться в массиве несколько раз в разных позициях.
- ❑ Тип **массива** Swift полностью записывается как **Array<Element>**, где **Element** — это тип значений, которые разрешено хранить в массиве.
- ❑ Вы также можете записать тип **массива** в сокращенной форме как **[Element]**.
- ❑ Хотя эти две формы функционально идентичны, сокращенная форма предпочтительнее

```
3 let array: [Int] = [1, 3, 2, 4, 4, 5]
4 let secondArray: Array<Int> = [6, 7, 6, 1, 3]
```

Создание пустого массива

Вы можете создать **пустой массив определенного типа**, используя синтаксис инициализатора:

```
1 var someInts: [Int] = []  
2 print("someInts is of type [Int] with \(${someInts.count}) items.")  
3 // Prints "someInts is of type [Int] with 0 items."
```

Создание массива со значением по умолчанию

Тип Swift `Array` также предоставляет инициализатор для создания **массива** определенного размера со всеми его значениями, установленными на одно и то же значение по умолчанию.

Вы передаете этому инициализатору значение по умолчанию соответствующего типа (называемое **repeating**): и количество раз, которое это значение повторяется в новом массиве (называемое **count**):

```
1 var threeDoubles = Array(repeating: 0.0, count: 3)
2 // threeDoubles is of type [Double], and equals [0.0, 0.0, 0.0]
```

Создание массива путем сложения двух массивов вместе

- ❑ Вы можете создать новый **массив**, объединив два существующих массива с совместимыми типами с помощью оператора сложения (+)
- ❑ Тип нового массива выводится из типа двух массивов, которые вы складываете вместе

```
1 var anotherThreeDoubles = Array(repeating: 2.5, count: 3)
2 // anotherThreeDoubles is of type [Double], and equals [2.5, 2.5, 2.5]
3
4 var sixDoubles = threeDoubles + anotherThreeDoubles
5 // sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5, 2.5,
   // 2.5]
```

Создание массива с литералом массива

Вы также можете инициализировать массив **литералом массива**, что является сокращенным способом записи одного или нескольких значений в виде коллекции массивов.

Литерал массива записывается в виде **списка значений**, разделенных запятыми и **окруженных парой квадратных скобок**

```
[value 1, value 2, value 3]
```

The example below creates an array called `shoppingList` to store `String` values:

```
1 var shoppingList: [String] = ["Eggs", "Milk"]
2 // shoppingList has been initialized with two initial items
```


Доступ и изменение значений массива

Вы получаете доступ к массиву и изменяете его с помощью его методов и свойств или с помощью синтаксиса нижнего индекса

- ❏ Чтобы узнать количество элементов в массиве, проверьте его свойство `count`:

```
1 print("The shopping list contains \$(shoppingList.count) items.")
2 // Prints "The shopping list contains 2 items."
```

Доступ и изменение значений массива

Вы можете добавить новый элемент в конец массива, вызвав метод `append(_)` массива

```
1 shoppingList.append("Flour")
2 // shoppingList now contains 3 items, and someone is making pancakes
```

В качестве альтернативы добавьте массив из одного или нескольких совместимых элементов с помощью оператора присваивания сложения (`+=`):

```
1 shoppingList += ["Baking Powder"]
2 // shoppingList now contains 4 items
3 shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
4 // shoppingList now contains 7 items
```

Доступ и изменение значений массива

Получите значение из массива, используя **синтаксис нижнего индекса**, передав **индекс значения, которое вы хотите получить**, в **квадратных скобках** сразу после имени массива

```
1 var firstItem = shoppingList[0]
2 // firstItem is equal to "Eggs"
```

Доступ и изменение значений массива

Вы можете использовать синтаксис нижнего индекса, чтобы **изменить существующее значение в данном индексе**

```
1 shoppingList[0] = "Six eggs"
2 // the first item in the list is now equal to "Six eggs" rather than "Eggs"
```

- ❑ Когда вы используете синтаксис нижнего индекса, указанный вами индекс должен быть допустимым
- ❑ Например, запись `shoppingList[shoppingList.count] = "Salt"`, чтобы попытаться добавить элемент в конец массива, приведет к ошибке (runtime error)

Доступ и изменение значений массива

Вы также можете использовать **синтаксис нижнего индекса** для одновременного изменения диапазона значений, даже если замещающий набор значений имеет другую длину, чем диапазон, который вы заменяете. В следующем примере «Шоколадная паста», «Сыр» и «Масло» заменены на «Бананы» и «Яблоки»

```
1 shoppingList[4...6] = ["Bananas", "Apples"]  
2 // shoppingList now contains 6 items
```

Доступ и изменение значений массива

Чтобы вставить элемент в массив по указанному индексу, вызовите метод `insert(_:at:)` массива

```
1 shoppingList.insert("Maple Syrup", at: 0)
2 // shoppingList now contains 7 items
3 // "Maple Syrup" is now the first item in the list
```

Этот вызов метода `insert(_:at:)` вставляет новый элемент со значением «Кленовый сироп» в самое начало списка покупок, обозначенного индексом 0

Доступ и изменение значений массива

Точно так же вы удаляете элемент из массива с помощью метода `remove(at:)`

Этот метод удаляет элемент по указанному индексу и возвращает удаленный элемент (хотя вы можете игнорировать возвращаемое значение, если оно вам не нужно)

```
1 let mapleSyrup = shoppingList.remove(at: 0)
2 // the item that was at index 0 has just been removed
3 // shoppingList now contains 6 items, and no Maple Syrup
4 // the mapleSyrup constant is now equal to the removed "Maple Syrup" string
```

Любые пробелы в массиве закрываются при удалении элемента, поэтому значение с индексом 0 снова равно «Шесть яиц»

```
1 firstItem = shoppingList[0]
2 // firstItem is now equal to "Six eggs"
```

Доступ и изменение значений массива

- ❑ Если вы хотите удалить последний элемент из массива, используйте метод `removeLast()`, а не метод `remove(at:)`, чтобы избежать необходимости запрашивать свойство `count` массива
- ❑ Как и метод `remove(at:)`, `removeLast()` возвращает удаленный элемент

```
1 let apples = shoppingList.removeLast()
2 // the last item in the array has just been removed
3 // shoppingList now contains 5 items, and no apples
4 // the apples constant is now equal to the removed "Apples" string
```


Сеты (Sets)

Сет хранит различные значения одного и того же типа в коллекции без определенного порядка

Вы можете использовать **сет** вместо массива, **когда порядок элементов не важен** или **когда вам нужно убедиться, что элемент появляется только один раз**

Хэш-значения для типов Сет

- ❑ Тип должен быть **хешируемым**, чтобы его можно было хранить в **Сете**, то есть *тип должен обеспечивать способ вычисления хэш-значения для самого себя*
- ❑ **Хэш-значение** — это значение **Int**, одинаковое для всех объектов, которые сравниваются одинаково, так что если **a == b**, **хэш-значение a равно хэш-значению b**
- ❑ Все базовые типы Swift (такие как **String**, **Int**, **Double** и **Bool**) по умолчанию хешируются и могут использоваться как типы значений **сет** или типы **ключей словаря**

Синтаксис типа Сет

Тип **Сет** Swift записывается как **Set<Element>**, где **Element** — это тип, который набору разрешено хранить.

В отличие от массивов, **Сеты** не имеют эквивалентной сокращенной формы.

```
3 let set: Set<Int> = [1, 2, 3, 4]
```

Создание и инициализация пустого Сета

Вы можете создать пустой **Сет** определенного типа, используя синтаксис инициализатора

```
1 var letters = Set<Character>()
2 print("letters is of type Set<Character> with \${letters.count} items.")
3 // Prints "letters is of type Set<Character> with 0 items."
```

Создание Сета с литералом массива

- ❑ Вы также можете инициализировать **Сет** с литералом массива, как сокращенный способ записи одного или нескольких значений в виде коллекции сета
- ❑ В приведенном ниже примере создается сет с именем **favoriteGenres** для хранения значений **String**

```
1 var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]
2 // favoriteGenres has been initialized with three initial items
```

- Переменная **favoriteGenres** объявлена как «сет строковых значений», записанная как **Set<String>**
- Поскольку для этого конкретного набора указан тип значения **String**, ему разрешено хранить только значения **String**
- Здесь набор **favoriteGenres** инициализируется тремя строковыми значениями («Рок», «Классика» и «Хип-хоп»), записанными внутри литерала массива

Создание Сета с литералом массива

- ❑ Тип **Сета** нельзя вывести только из литерала массива, поэтому тип **Set** должен быть объявлен явно
- ❑ Однако из-за **вывода типов (type inference)** Swift вам не нужно записывать тип элементов набора, если вы инициализируете его литералом массива, который содержит значения только одного типа.
- ❑ Вместо этого инициализацию **favoriteGenres** можно было бы записать в более короткой форме

```
var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"]
```

Доступ и изменение элементов Сета (Sets)

Вы получаете доступ к **Сету** и изменяете его через его методы и свойства.

- ❑ Чтобы узнать количество элементов в наборе, проверьте его свойство **count**

```
1 print("I have \ (favoriteGenres.count) favorite music genres.")
2 // Prints "I have 3 favorite music genres."
```

Доступ и изменение элементов Сета (Sets)

Вы можете добавить новый элемент в **Сет**, вызвав метод **insert(_:)**

```
1 favoriteGenres.insert("Jazz")  
2 // favoriteGenres now contains 4 items
```


Доступ и изменение элементов Сета (Sets)

- ❑ Вы можете удалить элемент из Сета, вызвав метод `remove(_:)` набора, который удаляет элемент, если он является членом набора, и возвращает удаленное значение или возвращает `nil`, если набор его не содержит.
- ❑ Кроме того, все элементы в наборе могут быть удалены с помощью его метода `removeAll()`

```
1  if let removedGenre = favoriteGenres.remove("Rock") {  
2      print("\(removedGenre)? I'm over it.")  
3  } else {  
4      print("I never much cared for that.")  
5  }  
6  // Prints "Rock? I'm over it."
```

Словари (Dictionary)

Словарь хранит ассоциации между **ключами** одного типа и **значениями** одного типа в коллекции без определенного порядка.

Каждое значение связано с уникальным ключом, который действует как идентификатор этого значения в **словаре**.

В отличие от элементов в массиве, элементы в словаре не имеют определенного порядка. Вы используете словарь, когда вам нужно искать значения на основе их идентификатора, почти так же, как реальный словарь используется для поиска определения определенного слова.

Сокращенный синтаксис типа Словаря

- ❑ Тип **словаря** Swift полностью записывается как `Dictionary<Key, Value>`, где **Key** — это *тип значения, которое можно использовать в качестве ключа словаря*, а **Value** — это *тип значения, которое словарь хранит для этих ключей*
- ❑ Вы также можете записать тип **словаря** в сокращенной форме как `[Key: Value]`
- ❑ Хотя эти две формы функционально идентичны, сокращённая форма предпочтительнее

```
3 let dict: Dictionary<Int, String> = [0: "Apple", 1: "Orange", 2: "Pineapple"]
4 let anotherDict: [String: String] = ["Car": "Ford", "Airplane": "Airbus", "Bike": "Harley"]
```

Создание пустого Словаря

- ❑ Как и в случае с массивами, вы можете создать пустой **словарь** определенного типа, используя синтаксис инициализатора
- ❑ В этом примере создается **пустой словарь** типа **[Int: String]** для хранения удобочитаемых имен целочисленных значений. Его ключи имеют тип **Int**, а его значения имеют тип **String**

```
1 var namesOfIntegers: [Int: String] = [:]  
2 // namesOfIntegers is an empty [Int: String] dictionary
```

Создание словаря с литералом словаря

Вы также можете инициализировать словарь литералом словаря, который имеет синтаксис, аналогичный литералу массива, рассмотренному ранее. Литерал словаря — это сокращенный способ записи одной или нескольких пар **ключ-значение** в виде коллекции **Dictionary**.

Пара **ключ-значение** представляет собой комбинацию ключа и значения. *В литерале словаря ключ и значение в каждой паре ключ-значение разделяются двоеточием.* Пары ключ-значение записываются в виде списка, разделенного запятыми и окруженного парой квадратных скобок

```
[ key 1 : value 1 , key 2 : value 2 , key 3 : value 3 ]
```

```
var airports = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

Доступ и изменение Словаря

Вы получаете доступ к **словарю** и изменяете его с помощью его методов и свойств или с помощью синтаксиса нижнего индекса

→ Как и в случае с массивом, вы узнаете количество элементов в словаре, проверив его свойство **count**

```
1 print("The airports dictionary contains \(airports.count) items.")
2 // Prints "The airports dictionary contains 2 items."
```

Доступ и изменение Словаря

- ❑ Вы можете добавить новый элемент в словарь с **синтаксисом нижнего индекса**.
- ❑ *Используйте новый ключ соответствующего типа в качестве индекса нижнего индекса и назначьте новое значение соответствующего типа*

```
1 airports["LHR"] = "London"
2 // the airports dictionary now contains 3 items
```

- ❑ *Вы также можете использовать синтаксис нижнего индекса, чтобы изменить значение, связанное с определенным ключом*

```
1 airports["LHR"] = "London Heathrow"
2 // the value for "LHR" has been changed to "London Heathrow"
```

Доступ и изменение Словаря

- ❑ В качестве альтернативы подписке используйте метод словаря `updateValue(_:forKey:)`, чтобы установить или обновить значение для определенного ключа
- ❑ Как и в приведенных выше примерах нижнего индекса, метод `updateValue(_:forKey:)` *устанавливает значение для ключа, если он не существует, или обновляет значение, если этот ключ уже существует*
- ❑ Однако, в отличие от индекса, метод `updateValue(_:forKey:)` возвращает старое значение после выполнения обновления. Это позволяет проверить, произошло ли обновление

```
1  if let oldValue = airports.updateValue("Dublin Airport", forKey: "DUB") {
2      print("The old value for DUB was \(oldValue).")
3  }
4  // Prints "The old value for DUB was Dublin."
```


Доступ и изменение Словаря

- ❑ Вы можете использовать синтаксис нижнего индекса, чтобы удалить пару ключ-значение из словаря, назначив значение `nil` для этого ключа

```
1 airports["APL"] = "Apple International"
2 // "Apple International" isn't the real airport for APL, so delete it
3 airports["APL"] = nil
4 // APL has now been removed from the dictionary
```

- ❑ В качестве альтернативы удалите пару ключ-значение из словаря с помощью метода `removeValue(forKey:)`
- ❑ Этот метод удаляет пару ключ-значение, если она существует, и возвращает удаленное значение или возвращает `nil`, если значение не существовало

```
1 if let removedValue = airports.removeValue(forKey: "DUB") {
2     print("The removed airport's name is \(removedValue).")
3 } else {
4     print("The airports dictionary doesn't contain a value for DUB.")
5 }
6 // Prints "The removed airport's name is Dublin Airport."
```

Control Flow

Swift предоставляет множество операторов управления (control flow). К ним относятся

- циклы `while` для многократного выполнения задачи;
- операторы `if`, `guard` и `switch` для выполнения различных ветвей кода на основе определенных условий;
- и такие операторы, как `break` и `continue`, переводят поток выполнения в другую точку вашего кода.

Swift также предоставляет цикл `for-in`, который упрощает перебор массивов (array), словарей (dictionary), диапазонов (range), строк (string) и других последовательностей.

For-In Цикл

Вы используете цикл **for-in** для перебора последовательности, такой как элементы в массиве, диапазоны чисел или символы в строке

→ В этом примере используется цикл **for-in** для перебора элементов массива

```
1  let names = ["Anna", "Alex", "Brian", "Jack"]
2  for name in names {
3      print("Hello, \(name)!")
4  }
5  // Hello, Anna!
6  // Hello, Alex!
7  // Hello, Brian!
8  // Hello, Jack!
```

For-In Цикл

- ❑ Вы также можете перебирать словарь, чтобы получить доступ к его парам ключ-значение.
- ❑ Каждый элемент в словаре возвращается как кортеж (**tuple**) (**ключ, значение**) при повторении словаря, и вы можете разложить элементы кортежа (ключ, значение) как константы с явным именем для использования в теле цикла **for-in**.
- ❑ В приведенном ниже примере кода ключи словаря разбиваются на константу с именем **animalName**, а значения словаря разлагаются на константу с именем **legCount**
- ❑ *Содержимое словаря по своей природе неупорядочено, и повторение по ним не гарантирует порядок, в котором они будут извлечены.* В частности, порядок вставки элементов в словарь не определяет порядок их повторения.

```
1 let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
2 for (animalName, legCount) in numberOfLegs {
3     print("\(animalName)s have \(legCount) legs")
4 }
5 // cats have 4 legs
6 // ants have 6 legs
7 // spiders have 8 legs
```

For-In Цикл

Вы также можете использовать циклы **for-in** с числовыми диапазонами (**range**). В этом примере выводятся первые несколько записей в таблице умножения на пять

```
1  for index in 1...5 {  
2      print("\(index) times 5 is \(index * 5)")  
3  }  
4  // 1 times 5 is 5  
5  // 2 times 5 is 10  
6  // 3 times 5 is 15  
7  // 4 times 5 is 20  
8  // 5 times 5 is 25
```

For-In Цикл

1. **Итерируемая последовательность** представляет собой диапазон чисел от 1 до 5 включительно, на что указывает использование оператора закрытого диапазона (`...`)
2. Значение индекса устанавливается равным первому числу в диапазоне (`1`), и операторы внутри цикла выполняются
3. В этом случае цикл содержит только один оператор, который выводит запись из таблицы умножения на пять для текущего значения `index`
4. После выполнения оператора значение `index` обновляется, чтобы содержать второе значение в диапазоне (`2`), и снова вызывается функция `print()`
5. Этот процесс продолжается до тех пор, пока не будет достигнут конец диапазона

For-In Цикл

- ❑ В некоторых ситуациях вы можете не захотеть использовать закрытые диапазоны, которые включают обе конечные точки.
- ❑ Рассмотрите возможность рисования делений для каждой минуты на циферблате. Вы хотите нарисовать 60 делений, начиная с 0 минуты.
- ❑ Используйте оператор полуоткрытого диапазона (`..<>`), чтобы *включить нижнюю границу, но не верхнюю границу*.

```
1 let minutes = 60
2 for tickMark in 0..
```

For-In Цикл

Также доступны закрытые диапазоны, используя вместо этого `stride(from:through:by:)`

```
1  let hours = 12
2  let hourInterval = 3
3  for tickMark in stride(from: 3, through: hours, by: hourInterval) {
4      // render the tick mark every 3 hours (3, 6, 9, 12)
5  }
```


While цикл

Цикл **while** *выполняет набор операторов до тех пор, пока условие не станет ложным*. Такие циклы лучше всего использовать, когда количество итераций неизвестно до начала первой итерации. Swift предоставляет два вида циклов **while**:

- **while** оценивает свое состояние в начале каждого прохода цикла.
- **repeat-while** оценивает свое состояние в конце каждого прохода цикла.

While цикл

- ❑ Цикл **while** начинается с оценки одного условия
- ❑ Если условие истинно, набор утверждений повторяется до тех пор, пока условие не станет ложным
- ❑ Вот общая форма цикла **while**:

```
while condition {  
    statements  
}
```

```
3 var number = 1  
4  
5 while number < 10 {  
6     number += 1  
7 }
```

Repeat-While цикл

Другой вариант цикла **while**, известный как цикл **repeat-while**

1. Сначала выполняет один проход через блок цикла, прежде чем рассматривать состояние цикла
2. Затем он продолжает повторять цикл, пока условие не станет ложным

```
repeat {  
  statements  
} while condition
```

```
3 var number = 11  
4  
5 repeat {  
6   number += 1 // but repeat goes before while check, and number will be 12  
7 } while number < 10 // 11 is more than 10, so its false
```

Условные операторы (Conditions)

Часто полезно выполнять разные фрагменты кода на основе определенных условий. Вы можете захотеть запустить дополнительный фрагмент кода при возникновении ошибки или отобразить сообщение, когда значение становится слишком высоким или слишком низким. Для этого вы делаете части своего кода условными.

Swift предоставляет два способа добавления условных ветвей в ваш код: оператор `if` и оператор `switch`. Как правило, оператор `if` используется для оценки простых условий с несколькими возможными результатами. Оператор `switch` лучше подходит для более сложных условий с несколькими возможными перестановками и полезен в ситуациях, когда сопоставление с образцом может помочь выбрать подходящую ветвь кода для выполнения.

If-else оператор

- ❑ В своей простейшей форме оператор **if** имеет одно условие `if`. Он выполняет набор операторов, только если это условие истинно
- ❑ В примере проверяется, меньше или равна ли температура 32 градусам по Фаренгейту (точка замерзания воды). Если это так, сообщение печатается. В противном случае сообщение не печатается, а выполнение кода продолжается после закрывающей скобки оператора **if**

```
1  var temperatureInFahrenheit = 30
2  if temperatureInFahrenheit <= 32 {
3      print("It's very cold. Consider wearing a scarf.")
4  }
5  // Prints "It's very cold. Consider wearing a scarf."
```

If-else оператор

- ❑ Оператор `if` может предоставить альтернативный набор операторов, известный как предложение `else`, для ситуаций, когда условие `if` ложно. Эти операторы обозначаются ключевым словом `else`
- ❑ Одна из этих двух ветвей всегда выполняется. Поскольку температура повысилась до 40 градусов по Фаренгейту, уже недостаточно холодно, чтобы советовать носить шарф, и вместо этого срабатывает ветвь `else`

```
1 temperatureInFahrenheit = 40
2 if temperatureInFahrenheit <= 32 {
3     print("It's very cold. Consider wearing a scarf.")
4 } else {
5     print("It's not that cold. Wear a t-shirt.")
6 }
7 // Prints "It's not that cold. Wear a t-shirt."
```

If-else оператор

- ❑ Вы можете связать несколько операторов **if** вместе, чтобы учесть дополнительные предложения
- ❑ Здесь был добавлен дополнительный оператор **if**, чтобы реагировать на особенно высокие температуры
- ❑ Последнее предложение **else** остается, и оно печатает ответ для любых температур, которые не являются ни слишком высокими, ни слишком низкими

```
1 temperatureInFahrenheit = 90
2 if temperatureInFahrenheit <= 32 {
3     print("It's very cold. Consider wearing a scarf.")
4 } else if temperatureInFahrenheit >= 86 {
5     print("It's really warm. Don't forget to wear sunscreen.")
6 } else {
7     print("It's not that cold. Wear a t-shirt.")
8 }
9 // Prints "It's really warm. Don't forget to wear sunscreen."
```

Swift

- ❑ Оператор **switch** рассматривает значение и сравнивает его с несколькими возможными шаблонами соответствия. Затем он выполняет соответствующий блок кода, основываясь на первом успешно совпавшем шаблоне. Оператор **switch** представляет собой альтернативу оператору **if** для реагирования на несколько потенциальных состояний
- ❑ В своей простейшей форме оператор **switch** сравнивает значение с одним или несколькими значениями того же типа

```
switch some value to consider {  
  case value 1 :  
    respond to value 1  
  case value 2 ,  
       value 3 :  
    respond to value 2 or 3  
  default :  
    otherwise, do something else  
}
```


Switch

- ❑ Каждый оператор **switch** должен быть исчерпывающим. То есть каждое возможное значение рассматриваемого типа должно соответствовать одному из вариантов переключения. *Если нецелесообразно предоставлять случай для каждого возможного значения, вы можете определить случай по умолчанию, чтобы охватить любые значения, которые не указаны явно.*
- ❑ Этот регистр по умолчанию обозначается ключевым словом **default** и всегда должен стоять последним

```
1  let someCharacter: Character = "z"
2  switch someCharacter {
3  case "a":
4      print("The first letter of the alphabet")
5  case "z":
6      print("The last letter of the alphabet")
7  default:
8      print("Some other character")
9  }
10 // Prints "The last letter of the alphabet"
```

Switch - Интервальное сопоставление

- ❑ Значения в `switch case` могут быть проверены на их наличие в интервале (`range`).
- ❑ В этом примере числовые интервалы используются для подсчета чисел любого размера на естественном языке

```
1 let approximateCount = 62
2 let countedThings = "moons orbiting Saturn"
3 let naturalCount: String
4 switch approximateCount {
5 case 0:
6     naturalCount = "no"
7 case 1..<5:
8     naturalCount = "a few"
9 case 5..<12:
10    naturalCount = "several"
11 case 12..<100:
12    naturalCount = "dozens of"
13 case 100..<1000:
14    naturalCount = "hundreds of"
15 default:
16    naturalCount = "many"
17 }
18 print("There are \({naturalCount}) \({countedThings}).")
19 // Prints "There are dozens of moons orbiting Saturn."
```

Switch - привязка значений

- ❑ Кейс `switch` может называть значение или значения, которые соответствуют временным константам или переменным, для использования в теле кейса.
- ❑ Такое поведение известно как привязка значений, поскольку значения привязываются к временным константам или переменным в теле обращения.

```
1  let anotherPoint = (2, 0)
2  switch anotherPoint {
3    case (let x, 0):
4      print("on the x-axis with an x value of \(x)")
5    case (0, let y):
6      print("on the y-axis with a y value of \(y)")
7    case let (x, y):
8      print("somewhere else at \(x), \(y)")
9  }
10 // Prints "on the x-axis with an x value of 2"
```

Switch - Комбинированные кейсы

- ❑ Несколько кейсов, которые имеют одно и то же тело, можно объединить, написав несколько шаблонов после регистра с запятой между каждым из шаблонов
- ❑ Если какой-либо из шаблонов совпадает, то случай считается совпавшим
- ❑ Шаблоны могут быть записаны в несколько строк, если список длинный

```
1 let someCharacter: Character = "e"
2 switch someCharacter {
3 case "a", "e", "i", "o", "u":
4     print("\(someCharacter) is a vowel")
5 case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
6     "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
7     print("\(someCharacter) is a consonant")
8 default:
9     print("\(someCharacter) isn't a vowel or a consonant")
10 }
11 // Prints "e is a vowel"
```

Операторы передачи управления (Control Transfer Statements)

Операторы передачи управления изменяют порядок выполнения вашего кода, передавая управление от одной части кода к другой. Swift имеет пять операторов передачи управления:

- `continue`
- `break`
- `fallthrough`
- `return`
- `throw`

Continue

- ❑ Оператор `continue` сообщает циклу, *что нужно остановить то, что он делает, и начать заново в начале следующей итерации цикла*. Он говорит: «Я закончил текущую итерацию цикла», не выходя из цикла полностью
- ❑ В следующем примере из строчной строки удаляются все гласные и пробелы, чтобы создать загадочную фразу-головоломку:

```
1  let puzzleInput = "great minds think alike"
2  var puzzleOutput = ""
3  let charactersToRemove: [Character] = ["a", "e", "i", "o", "u", " "]
4  for character in puzzleInput {
5      if charactersToRemove.contains(character) {
6          continue
7      }
8      puzzleOutput.append(character)
9  }
10 print(puzzleOutput)
11 // Prints "grtmndsthnlk"
```

Break

- ❑ Оператор `break` немедленно завершает выполнение всего оператора потока управления.
- ❑ Оператор `break` может использоваться внутри оператора `switch` или `цикла`, когда вы хотите прервать выполнение оператора `switch` или `цикла` раньше, чем это было бы в противном случае.

Fallthrough

- ❑ В Swift операторы `switch` не проходят через нижнюю часть каждого `case` в следующий. То есть весь оператор `switch` завершает свое выполнение, как только завершается первый соответствующий случай.
- ❑ `fallthrough` оператор позволяет после успешного кейса, провалится в следующий кейс

```
1 let integerToDescribe = 5
2 var description = "The number \(integerToDescribe) is"
3 switch integerToDescribe {
4 case 2, 3, 5, 7, 11, 13, 17, 19:
5     description += " a prime number, and also"
6     fallthrough
7 default:
8     description += " an integer."
9 }
10 print(description)
11 // Prints "The number 5 is a prime number, and also an integer."
```


Thanks for your attention!