



Урок 2

Синтаксис Swift. Основные операторы

Продолжение изучения синтаксических конструкций языка и основных операторов

[Базовые операторы](#)

[Полиморфизм операторов](#)

[Присваивание](#)

[Арифметические операторы](#)

[Составные выражения](#)

[Составное присваивание](#)

[Операторы сравнения](#)

[Логические операторы](#)

[Ветвление и выбор](#)

[If](#)

[Тернарный оператор](#)

[Switch](#)

[Циклические операторы](#)

[For-in](#)

[While](#)

[Repeat-while](#)

[Операторы передачи управления](#)

[Функции](#)

[Параметры](#)

[Область видимости](#)

[Домашнее задание](#)

[Практика](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Базовые операторы

На прошлом уроке мы познакомились с переменными и коллекциями. Они предназначены для представления данных в коде программы, но этого мало, чтобы заставить программу работать. Необходим механизм изменения данных, и в этой роли выступают операторы. Вспомните математику – в ней были операции (умножение, сложение, деление и т.д.). Операторы в Swift очень на них похожи, но имеют особенности.

Операторы получают один или несколько аргументов и создают на их основе новое значение. В качестве примера рассмотрим оператор сложения «+». Он принимает два аргумента, слева и справа от него, и в результате создает новое значение: 1 + 4 будет 5.

Как и в математике, операторы имеют приоритет. Это означает, что если в одном выражении будет использовано несколько операторов, то одни из них выполняются раньше, другие – позже.

Полиморфизм операторов

Важно понимать, что действие, выполняемое оператором, зависит не столько от него самого, сколько от значений, с которыми он работает. Так, «+» (сложение), примененное к двум числам, действительно будет их складывать, но если его применить к двум строкам, оно будет их склеивать. Более того, многие типы значений вообще не поддерживают большинство операторов.

<pre>7 + 5 // сложение "Привет " + "мир!" // склеивание 7 * 5 // умножение "Привет " * "мир!" // ошибка, так как строки не поддерживают оператор "*"</pre>	<pre>12 привет мир 35 error</pre>
--	-----------------------------------

Присваивание

Инициализирует или изменяет значение переменной, расположенной слева от оператора, значением, расположенным справа.

Этот оператор настолько прост и привычен всем, кто знаком с программированием, что его часто вообще не причисляют к операторам, а воспринимают как нечто само собой разумеющееся. Тем не менее это оператор, и он тоже участвует в выражении.

<pre>var a = 5 // Инициализирует переменную "a" значением 5 var b = 7 // Инициализирует переменную "b" значением 7 a = 9 // Присваивает переменной "a" новое значение 9 b = a // Присваивает переменной "b" новое значение, равное значению "a"</pre>	<pre>5 7 9 9</pre>
--	---------------------

Арифметические операторы

Операторы, знакомые вам по школьному курсу математики. На основе двух значений, расположенных слева и справа от них, создают новое значение:

- «+» – сложение.
- «-» – вычитание.
- «*» – умножение.
- «/» – деление.
- «%» – остаток от деления.

4 + 7	11
4 - 7	-3
4 * 7	28
4 / 7	0
4 % 7	4

Составные выражения

Операторы можно комбинировать в сложные выражения. Наиболее часто используемая комбинация – это какой-нибудь оператор и оператор присваивания. В результате такого выражения вычисленное значение сразу будет сохранено в переменную. Согласитесь, получить новое значение и никак его не использовать – бессмысленно.

Как уже было сказано, в выражении операторы будут выполняться не слева направо, а в порядке приоритета. Как и в математике, умножение выполняется раньше сложения. Порядок выполнения также можно менять скобками «()».

<code>var a = 4 + 7</code> // сначала выполнится оператор сложения, и его результат будет присвоен переменной "a"	11
<code>a = 4 + 7 + 5</code> // сначала выполняется первый оператор сложения, результат его выполнения будет использован вторым оператором сложения, итоговый результат будет использован оператором присваивания	16
<code>a = 4 + 7 * 5</code> // сначала выполняется умножение, потом сложение, затем присваивание	39
<code>a = (4 + 7) * 5</code> // сначала выполняется сложение, потом умножение, затем присваивание	55
<code>a = a + a</code> // оператор сложения сложит 55 и 55, результат присвоит переменной "a"	110
<code>a = a + 18</code> // оператор сложения возьмет значение переменной "a" и 18, результат присвоит переменной "a"	128

Составное присваивание

Ситуации, когда необходимо взять значение переменной, модифицировать его, а результат снова присвоить этой же переменной, очень часто встречаются в программировании. В Swift специально для этого ввели операторы составного присваивания: «+=», «-=», «*=», «/=», «%=» и другие. Фактически вы можете взять любой оператор и добавить к нему «=».

<code>var x = 4</code>	4
<code>x = x + 4</code> // использовать значение переменной x в качестве аргумента оператора присваивания, результат присвоить переменной "x"	8
<code>x += 2</code> // эквивалентно выражению выше, но используется составной оператор	10
<code>x *= 3</code>	30
<code>x /= 15</code>	2

Операторы сравнения

Операторы этой группы сравнивают два значения и создают новое, логическое значение. Как вы помните, логический (Bool) тип может иметь два значения: «false» («ложь») и «true» («истина»):

- «<» – меньше.
- «>» – больше.
- «==» – равно.
- «!=» – не равно.
- «<=» – меньше либо равно.
- «>=» – больше либо равно.

<code>5 < 4</code>	false
<code>5 > 4</code>	true
<code>5 == 4</code>	false
<code>5 != 4</code>	true
<code>5 <= 4</code>	false
<code>5 >= 4</code>	true
<code>let a = 10</code>	10
<code>let b = 2</code>	2
<code>let x = a > b</code>	true

Логические операторы

Значение логического типа также можно преобразовывать. Для этого существуют логические операторы:

- «&&» – логическое И.
- «||» – логическое ИЛИ.
- «!» – логическое НЕ.

Логическое И («&&») возвращает «true» только в том случае, когда оба его аргумента равны «true».

<code>true && true</code>	true
<code>false && true</code>	false
<code>true && false</code>	false
<code>false && false</code>	false

Логическое ИЛИ («||») возвращает «true», если хотя бы один из его аргументов равен «true».

<pre>true true false true true false false false</pre>	<pre>true true true false</pre>
--	---------------------------------

Логическое НЕ («!») меняет значение на противоположное. В отличие от остальных, он принимает только один аргумент, точнее, просто ставится перед значением.

<pre>!false !true</pre>	<pre>true false</pre>
-------------------------	-----------------------

Из логических операторов также можно составлять выражения. При этом рекомендуется указывать круглые скобки, даже если вам не нужно менять приоритет выполнения операторов. Это позволит улучшить читаемость.

<pre>5 > 10 && 5 < 10 (5 > 10) && (5 < 10) // эквивалентно выражению выше, но читается лучше (5 > 10) (5 < 10)</pre>	<pre>false false true</pre>
---	-----------------------------

Ветвление и выбор

Очень часто необходимо выполнять те или иные блоки кода в зависимости от текущего состояния приложения. Например, в зависимости от значения переменной вы можете увеличить его или уменьшить. Другой пример – реакция приложения на действие пользователя: на разные действия она должна быть разной. На помощь приходят операторы ветвления и выбора.

If

Этот оператор очень легко запомнить и понять, на русский его можно перевести как «если».

Если условие истинно, выполнить код:

<pre>let x = 10 if x == 10 { // условие верно, код в блоке выполнится print('perform') } if x == 20 { // условие неверно, код в блоке не выполнится print('perform again') }</pre>	<pre>"perform/n"</pre>
--	------------------------

Кроме того, можно задать несколько уточнений. Если условие ложно, проверить другое условие, и если оно истинно, выполнить другой код:

<pre> let x = 10 if x == 20 { // условие неверно, код в блоке не выполняется print('perform') } else if x == 10 { // условие верно, код в блоке выполняется print('perform again') } </pre>	<pre> "perform again/n" </pre>
--	--------------------------------

Можно добавить блок кода, который выполняется, если ни одно из условий не истинно.

<pre> let x = 10 if x == 20 { // условие неверно, код в блоке не выполняется print('perform') } else if x == 30 { // условие неверно, код в блоке не выполняется print('perform again') } else { print("default") } </pre>	<pre> "default/n" </pre>
---	--------------------------

Если ваш «if» содержит несколько верных условий, то выполнится верхнее.

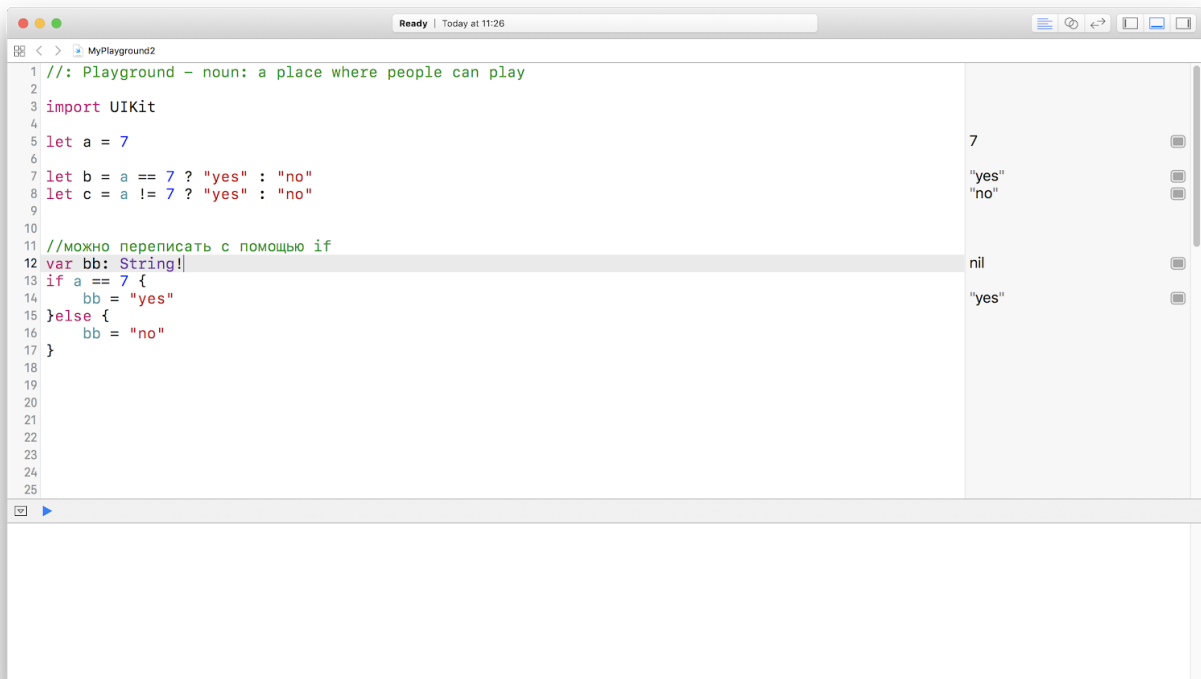
<pre> let x = 10 if x == 10 { // условие верно, код в блоке выполняется print('perform') } else if x <= 10 { // условие верно, но код в блоке уже не выполнится print('perform again') } else { print("default") } </pre>	<pre> "'perform'/n" </pre>
---	----------------------------



Тернарный оператор

Данный оператор является краткой версией «if». Использовать его необходимо крайне редко и осторожно, так как он плохо читается. Чаще всего применяется при инициализации переменной.

выражение ? действие1 : действие2



Switch

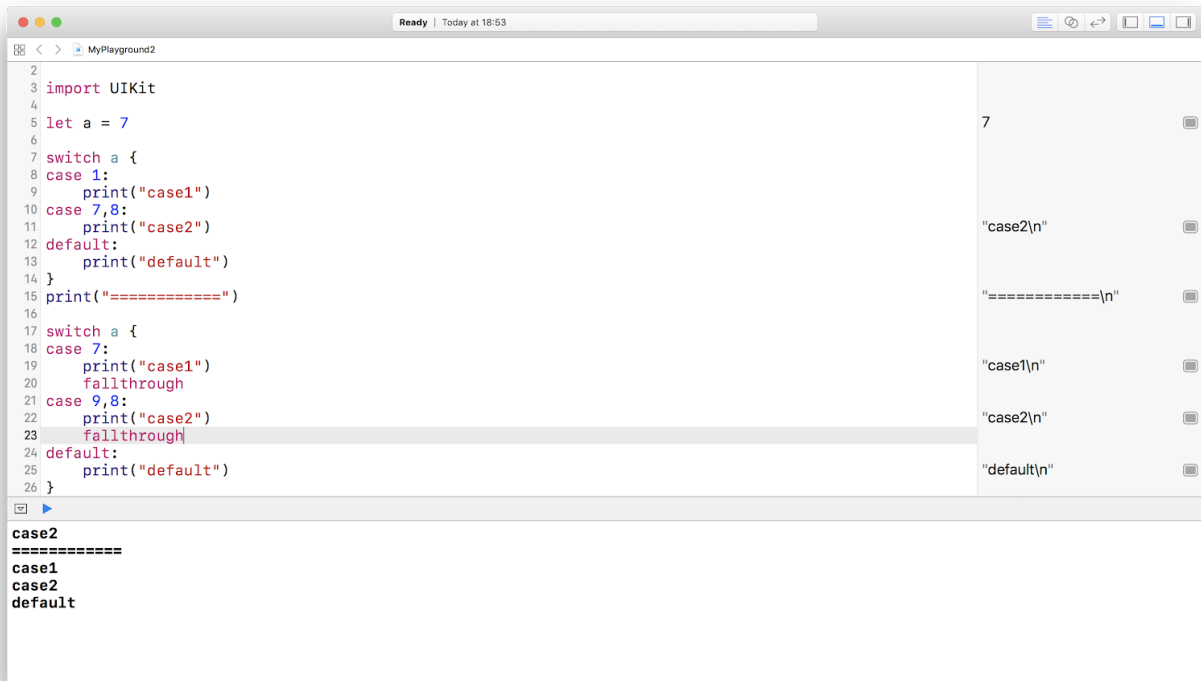
Данный оператор не проверяет условие, как это делает «if». В нем описываются возможные значения некоторой переменной. Если значение совпадает с описанием, выполняется определенный блок кода. В каждом блоке «case» можно описать несколько вариантов через «,».

```
switch значение для проверки {
    case вариант 1:
        блок кода
    case вариант 2, вариант 3:
        блок кода
    default:
        блок кода // выполняется, если ни один из вариантов не будет
                    соответствовать значению
}
```

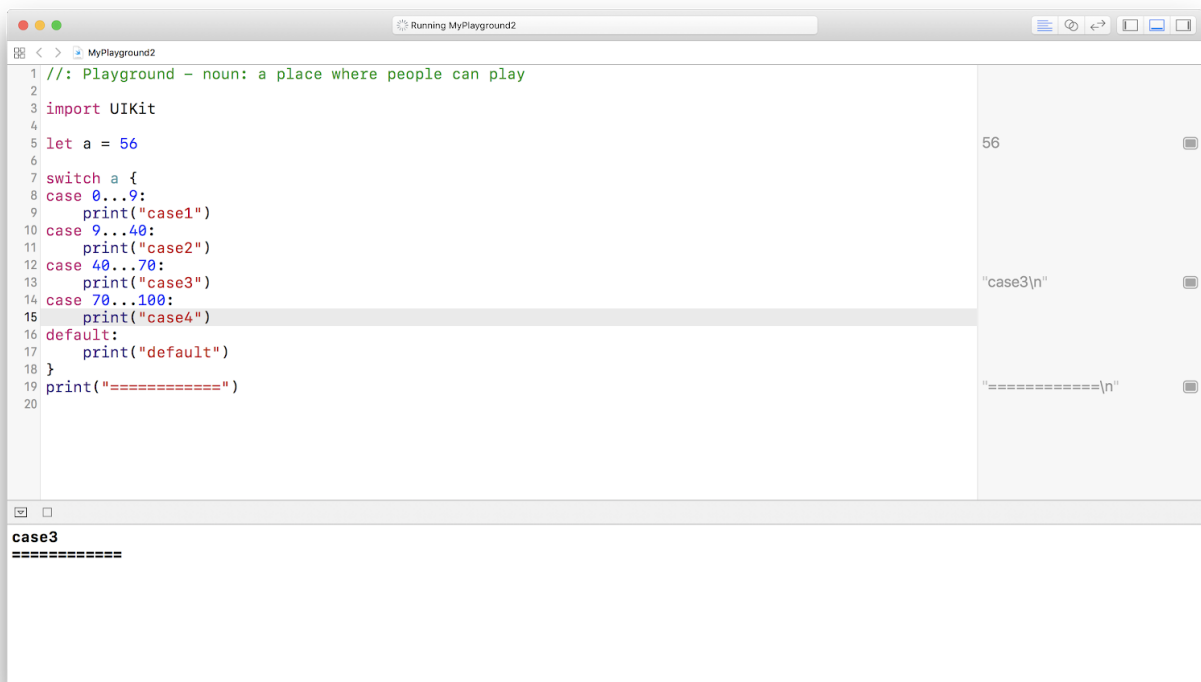
Сравнение вариантов со значением идет сверху вниз. После того как будет найден подходящий вариант и выполнится его блок кода, остальные варианты сравниваться не будут. Так, если в «switch» будет несколько вариантов, соответствующих значению, выполнится только один.



В других языках по умолчанию выполняются все блоки «case», что находятся ниже подходящего варианта. Но в Swift такое поведение выключено. Чтобы его включить, надо добавить ключевое слово «fallthrough» в «case».



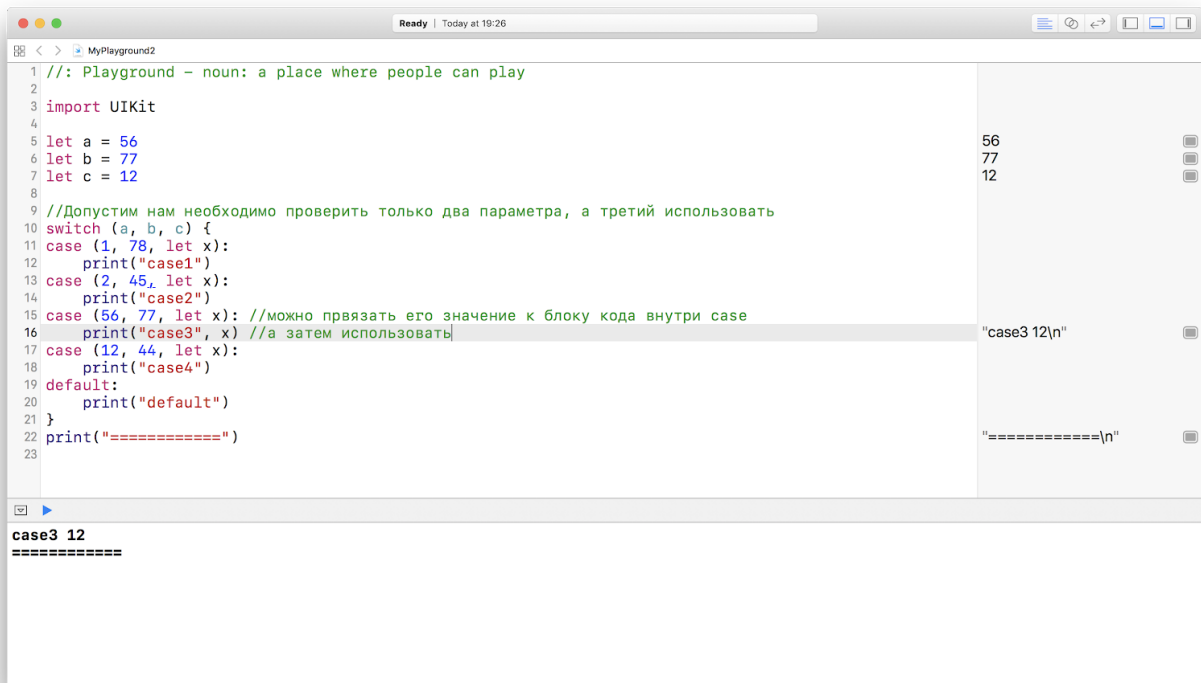
Поддерживается проверка значения на входжение в диапазон:



Каждый «case» может сравнивать несколько значений, если их передать в кортеже. В результате будет выбран тот вариант, который будет соответствовать всем значениям. Если какой-то из элементов кортежа не важен при сравнении в «case», можно поставить на его месте прочерк.



Можно сравнить только несколько параметров кортежа, а остальные использовать в блоке кода «case».



Последняя отличительная возможность «switch» в Swift – это уточнение вариантов с помощью блока «where». Так вы сможете не просто проверить соответствие переданного значения какому-либо из вариантов, но и проверить какие-то внешние условия.



Циклические операторы

Циклические операторы необходимы для повторения одного и того же блока кода по несколько раз. Например, если вам необходимо вывести в консоль пять раз слово «повтор», вы можете просто написать пять строк кода, а можете написать цикл, который пять раз повторит вывод строки в консоль.

The screenshot shows a Swift playground window titled "MyPlayground2". The code editor contains the following code:

```
1 //: Playground – noun: a place where people can play
2
3 import UIKit
4
5 print("Повтор")
6 print("Повтор")
7 print("Повтор")
8 print("Повтор")
9 print("Повтор")
```

On the right side, the output of each line is shown as a string: "Повтор\n". Below the code editor, the console displays the word "Повтор" five times, one on each line.

The screenshot shows a Swift playground window titled "MyPlayground2". The code editor contains the following code:

```
1 //: Playground – noun: a place where people can play
2
3 import UIKit
4
5 for _ in (1...5) {
6     print("Повтор")
7 }
```

On the right side, the output of the loop is shown as "(5 times)". Below the code editor, the console displays the word "Повтор" five times, one on each line.

Чтобы в полной мере прочувствовать пользу циклов, представьте, что вам внезапно понадобилось изменить количество выводов в консоль с 5 до 5000. В первом случае будет довольно сложно скопировать строку еще 4995 раз. В случае с циклом достаточно изменить 5 на 5000 – и дело в шляпе!

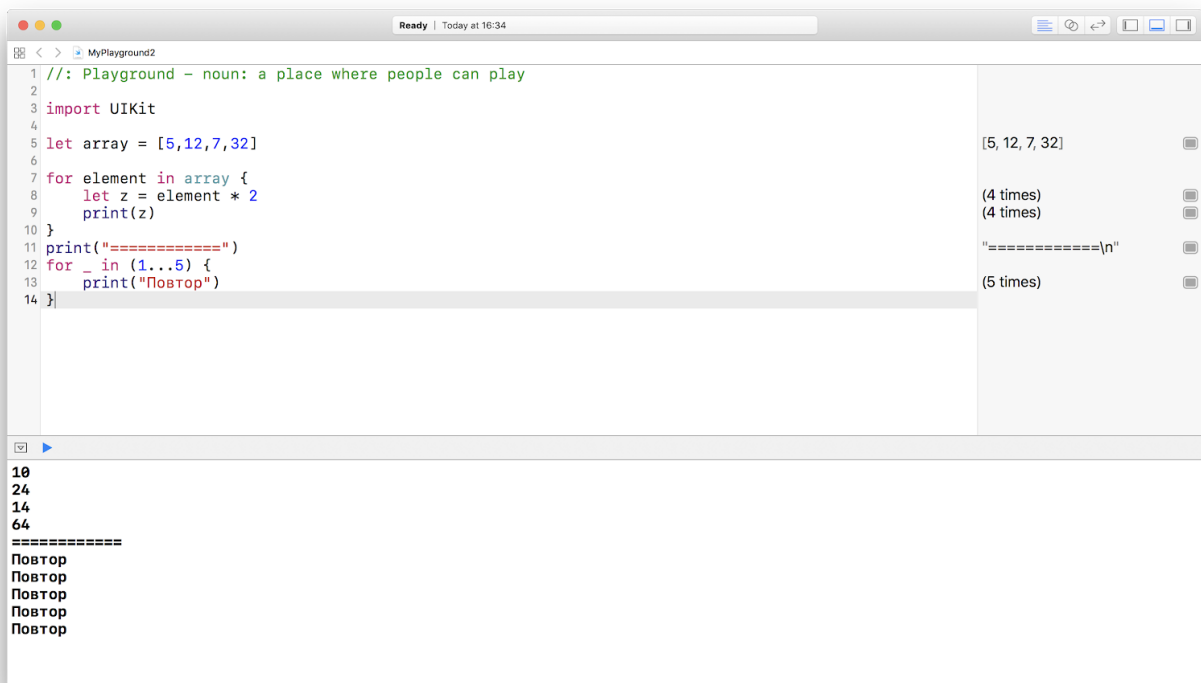
Один повтор в цикле принято называть итерацией, а блок кода, который повторяется, – телом цикла. Другими словами, цикл выполняет некоторое количество итераций. При этом во время каждой итерации выполняется тело цикла.

Следующее преимущество циклов заключается в том, что вы можете менять количество итераций в процессе выполнения программы в зависимости от внешних условий. Подставив вместо 5 переменную «a», вы получите количество итераций, равное значению переменной.

For-in

Основное назначение этого оператора – итерации по коллекциям. При этом на каждой итерации в тело цикла передается текущий элемент коллекции.

```
for имя_элемента in коллекция { тело цикла }
```



При выполнении цикла на каждой итерации в теле цикла будет создаваться новая переменная с именем, указанным при объявлении, и значением, равным текущему элементу. Эта переменная является константой, изменять её нельзя. Само тело цикла может содержать любую логику работы с этой переменной или выполнения любых других действий.

Если в обработке коллекции необходимости нет, но нужно просто повторить какой-то код несколько раз, в качестве коллекции вы можете указать диапазон. В этом случае в тело цикла будет передаваться число из диапазона. Если и в нём нет необходимости, вместо имени переменной можно поставить прочерк – в таком случае переменная в тело цикла передаваться не будет.

```
for имя_элемента in (a...b) { тело цикла } // где a и b - целые числа
for имя_элемента in (1...5) { тело цикла } // выполнится 5 итераций: 1 2 3 4 5
for _ in (1...5) { тело цикла } // вместо имени переменной можно
указать прочерк. В этом случае она не будет передаваться в тело цикла
```

Иногда возникает необходимость итерировать некоторый диапазон шагом, отличным от 1. Например, вам нужно обработать числа от 7 до 18 с шагом 2. В этом случае на помощь приходит «stride» – эта функция создает определенный диапазон с указанным шагом. Есть два варианта данной функции:

```
stride(from: a, through: b, by: n) // Создает диапазон от "a" до "b"
включительно, с шагом n
stride(from: a, to: b, by: n)      // Создает диапазон от "a" до "b", не включая
"b", с шагом n
```

```
1 //: Playground - noun: a place where people can play
2
3 import UIKit
4
5
6 for i in stride(from: 0, to: 5, by: 2) {
7     print(i)
8 }
9 print("=====\n")
10 for i in stride(from: 0, through: 9, by: 3) {
11     print(i)
12 }
```

(3 times)

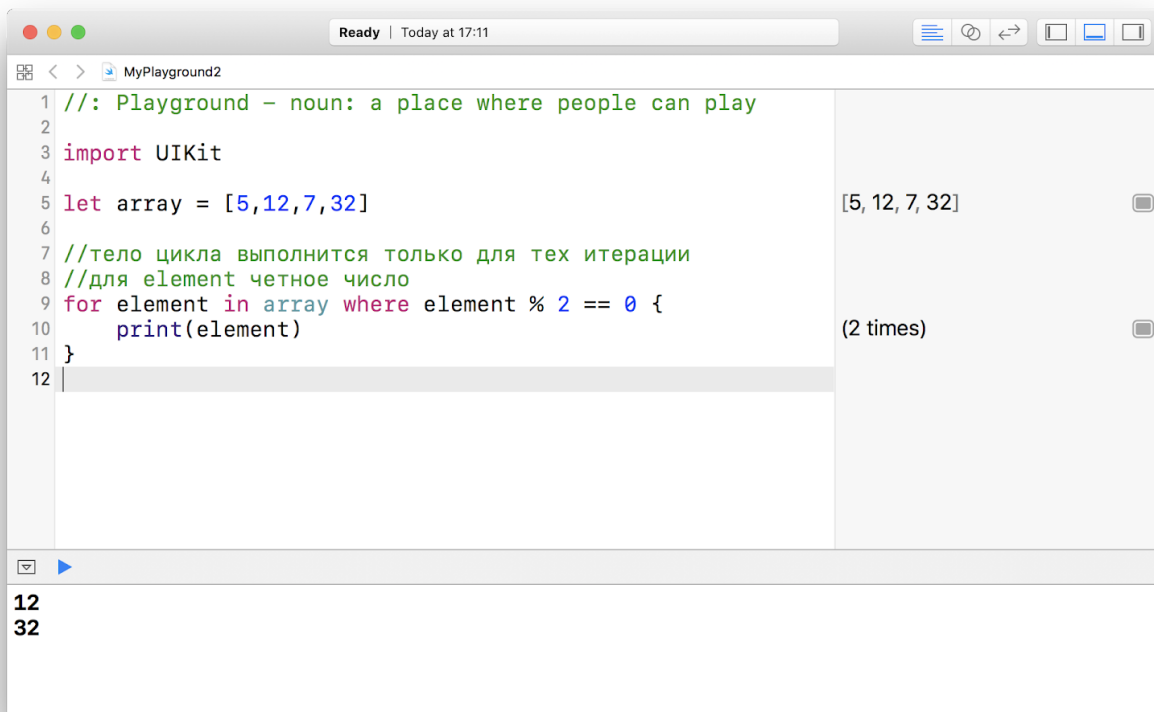
"=====\n"

(4 times)

0
2
4
=====
0
3
6
9

Особенность цикла «for» в языке Swift – возможность уточнить, для каких итераций стоит выполнять тело цикла, а для каких – нет.

```
for имя_элемента in коллекция where логическое_выражение { тело цикла }
```

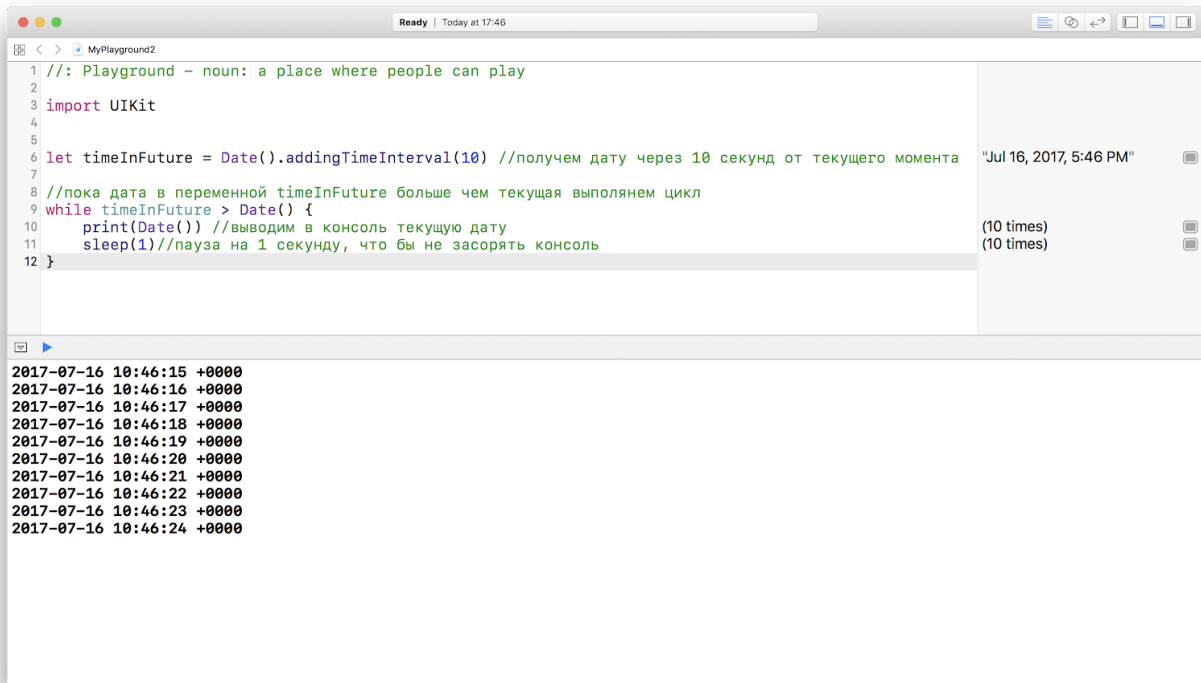


While

Цикл, созданный с помощью этого оператора, будет выполняться, пока указанное при создании условие истинно. Чаще всего используется в тех случаях, когда заранее неизвестно, сколько итераций должен выполнить цикл, но известно условие, при котором оно должно выполняться.

Важно понимать, что условие проверяется до выполнения тела цикла, и если в момент запуска цикла условие ложно, то цикл не выполнится ни разу.

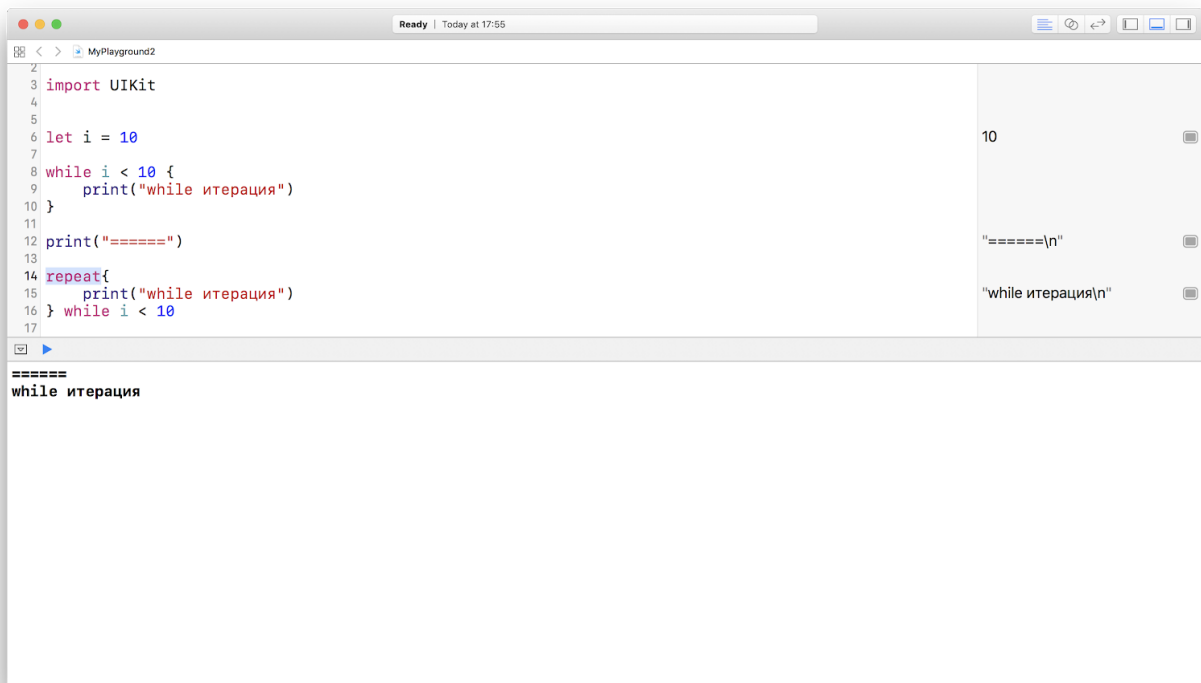
```
while условие { тело_цикла }
```

Repeat-while

Цикл, почти идентичный «while», но с одной оговоркой: он выполняет тело цикла **до** проверки условия, а не после. Поэтому если в момент запуска циклов условие ложно, то «while» не выполнится ни разу, а «repeat-while» выполнится один раз.

```
repeat {  
    тело_цикла  
} while условие
```



Операторы передачи управления

По умолчанию цикл выполняет все свои итерации. Иногда возникает необходимость не выполнять тело цикла до конца, а, пропустив оставшуюся часть, сразу перейти к следующей итерации, а возможно, и прервать выполнение цикла целиком. Для этого у нас есть два оператора передачи управления:

- «continue» – завершает выполнение тела цикла в месте вызова и переходит к следующей итерации;
- «break» – завершает выполнение тела цикла в месте вызова и останавливает выполнение цикла.

Функции

Мы пока не писали больших программ – все они умещались на один экран, но в реальной жизни приложения состоят из миллионов строк кода, разбросанного по разным файлам. В этом случае часто возникает ситуация, когда одно и то же действие необходимо выполнить в нескольких местах программы. Иногда это одна строка кода, иногда бывает 100 и более.

Представьте, что перед вами встала такая задача. Скорее всего, вы просто скопируете нужный код и вставите его в необходимое место. Выглядит несложно, но у этого подхода есть ряд минусов. Во-первых, объем вашей программы растет. Во-вторых, если этот код понадобится изменить, вам придется повторять свои изменения во всех местах, где он встречается.

Специально для решения этой проблемы были созданы функции. Функция – это блок кода, который можно переиспользовать в любом месте программы сколько угодно раз.

Использование функции называется «вызов». Другими словами, если вам необходимо где-то выполнить блок кода из функции, вы ее вызываете. Функция имеет имя, по которому мы отличаем ее от других и можем вызывать.



На самом деле вы уже работали с функциями, но все они были из стандартного набора: «row», «sqrt», «print» и т.д.

Параметры

Параметры могут быть входными – те, что передаются функции во время вызова и используются внутри функции. А могут быть выходными – те, что можно получить от функции и использовать в месте вызова. Так, функция «sqrt» принимает число, вычисляет его корень и возвращает результат своих вычислений. Значение, которое возвращает функция, можно использовать в выражении, а можно присвоить его переменной. Возвращаемое значение также называют результатом функции.

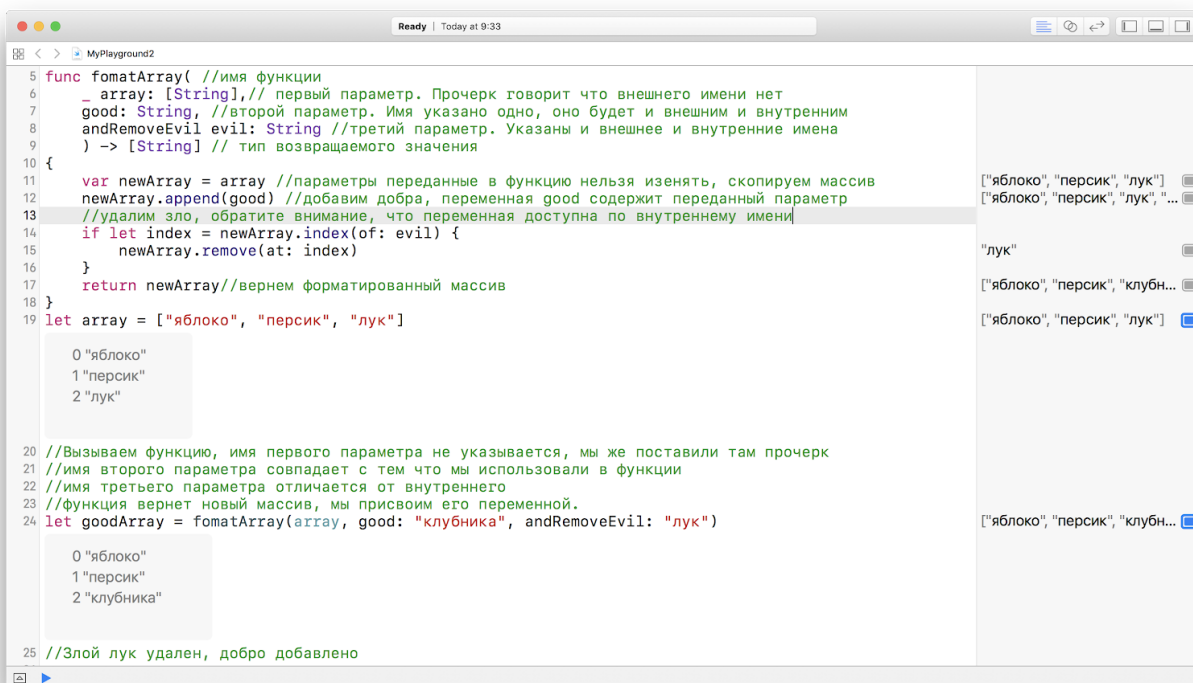
<code>row(4)</code>	<i>// мы передали функции параметр «4», она вернула новое значение «2», которое мы никак не использовали</i>	2
<code>let x = row(4)</code>	<i>// здесь мы присваиваем возвращаемое функцией значение переменной «x»</i>	2
<code>let x = 7 + row(4)</code>	<i>// используем результат функции в выражении</i>	9

Будет ли функция принимать или возвращать параметры, описывается при ее определении. Количество входных параметров не ограничено, а выходной может быть только один. Конечно, вы можете вернуть один кортеж с несколькими значениями.

Входные параметры отделены друг от друга запятой. Они имеют два имени. Внешнее имя используется в месте вызова функции, внутреннее – внутри самой функции. Можно объединить внешние и внутренние имена, указав только одно имя параметра. Имена внешних аргументов необходимы, чтобы при вызове функции было понятно, какие данные следует ей передать. Если вы

считаете, что все ясно без лишних слов, можете вообще убрать внешнее имя, поставив вместо него прочерк, как сделано в функции «row».

```
func имя_функции ( внешнее_имя_параметра внутреннее_имя_параметра: тип_параметра
) -> тип_возвращаемого_значения
func имя_функции ( внешнее_и_внутреннее_имя_параметра: тип_параметра ) ->
тип_возвращаемого_значения
func имя_функции ( _ внутреннее_имя_параметра: тип_параметра ) ->
тип_возвращаемого_значения
```



```
5 func fomatArray( //имя функции
6   _ array: [String], // первый параметр. Прочерк говорит что внешнего имени нет
7   good: String, //второй параметр. Имя указано одно, оно будет и внешним и внутренним
8   andRemoveEvil evil: String //третий параметр. Указаны и внешнее и внутренние имена
9 ) -> [String] // тип возвращаемого значения
10 {
11   var newArray = array //параметры переданные в функцию нельзя изменять, скопируем массив
12   newArray.append(good) //добавим добра, переменная good содержит переданный параметр
13   //удалим зло, обратите внимание, что переменная доступна по внутреннему имени
14   if let index = newArray.index(of: evil) {
15     newArray.remove(at: index)
16   }
17   return newArray //вернем форматированный массив
18 }
19 let array = ["яблоко", "персик", "лук"]

20 //Вызываем функцию, имя первого параметра не указывается, мы же поставили там прочерк
21 //имя второго параметра совпадает с тем что мы использовали в функции
22 //имя третьего параметра отличается от внутреннего
23 //функция вернет новый массив, мы присвоим его переменной.
24 let goodArray = fomatArray(array, good: "клубника", andRemoveEvil: "лук")

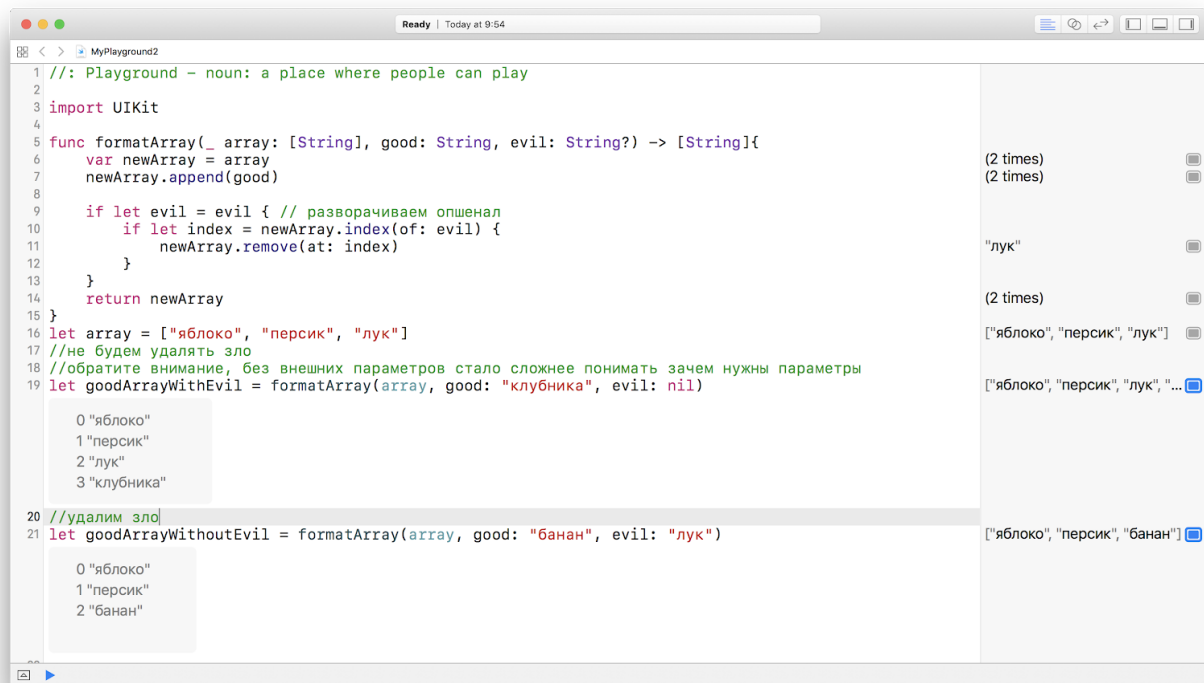
25 //Злой лук удален, добро добавлено
```

0 "яблоко"
1 "персик"
2 "лук"

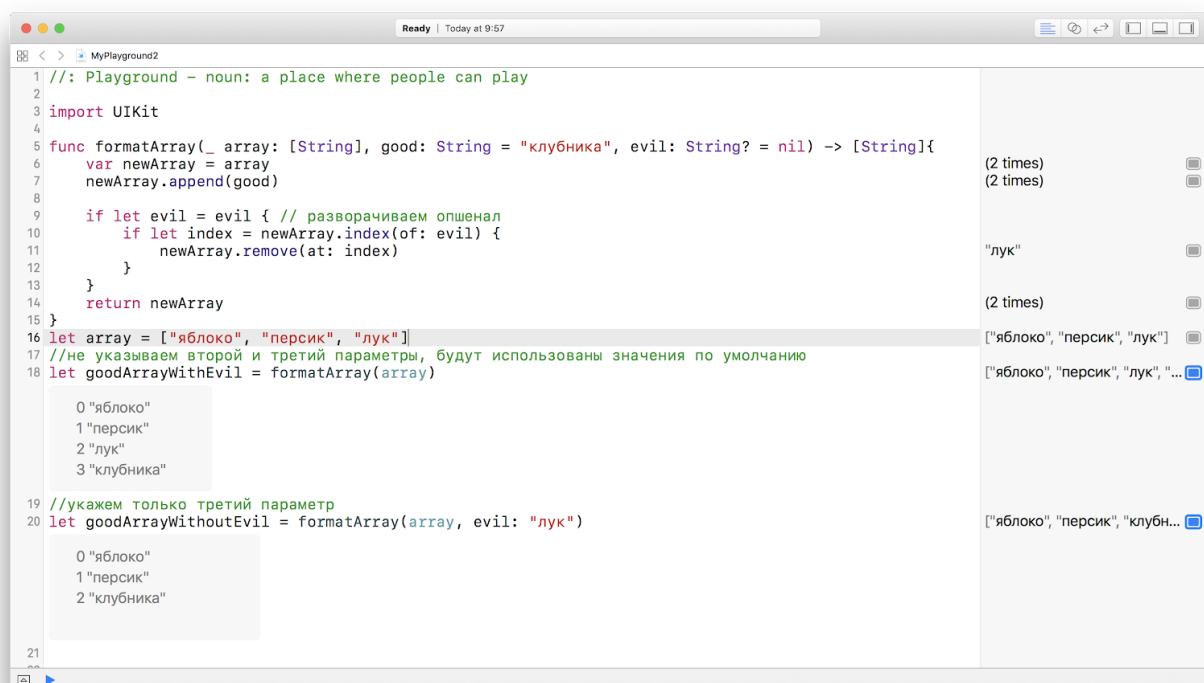
0 "яблоко", "персик", "лук"
1 "яблоко", "персик", "лук", "...
"лук"
2 "яблоко", "персик", "клубн...
3 "яблоко", "персик", "лук"

0 "яблоко", "персик", "клубн...

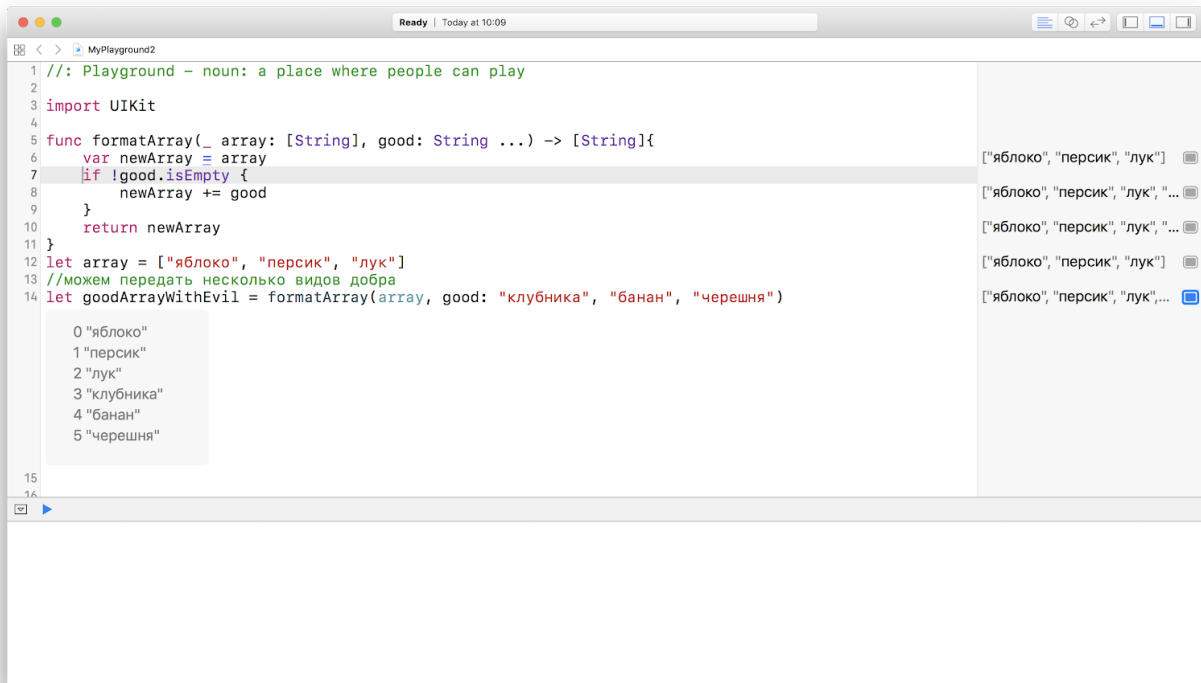
Иногда, описывая функцию, вы понимаете, что один из параметров нужен не всегда. В таком случае вы можете указать его опциональным.



Некоторым параметрам вы можете установить значение по умолчанию и при вызове решать, следует ли указать особое значение или достаточно значения по умолчанию.

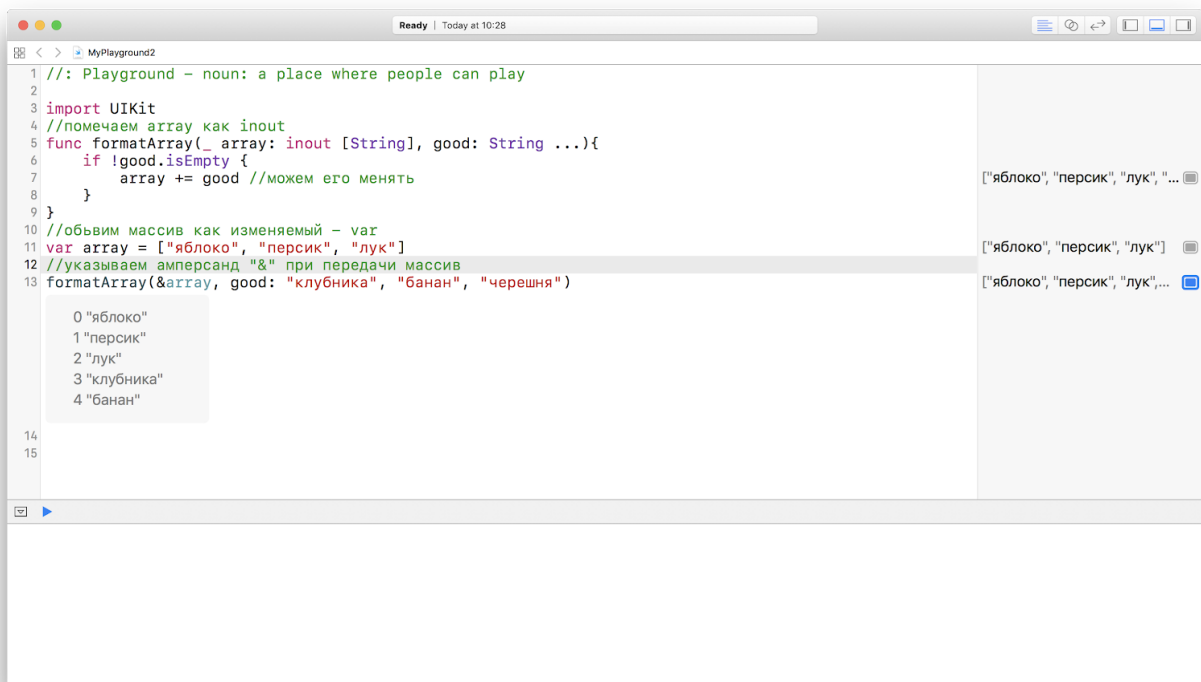


Как вы могли заметить, функция может принимать столько параметров, сколько мы описали. Но есть один трюк, чтобы она принимала любое количество параметров, – добавить вариативный параметр. Он может быть только один и всегда указывается последним. Внутри функции переменная вариативного параметра представляет собой массив.



Обратите внимание, что функция возвращает новый массив, а старый остается неизменным. В большинстве случаев такое поведение предпочтительно, но иногда необходимо изменить именно переданный параметр, а не вернуть новый. Если вы попытаетесь это проверить, получите ошибку, ведь параметры передаются внутрь функции как константы. Но есть способ это исправить.

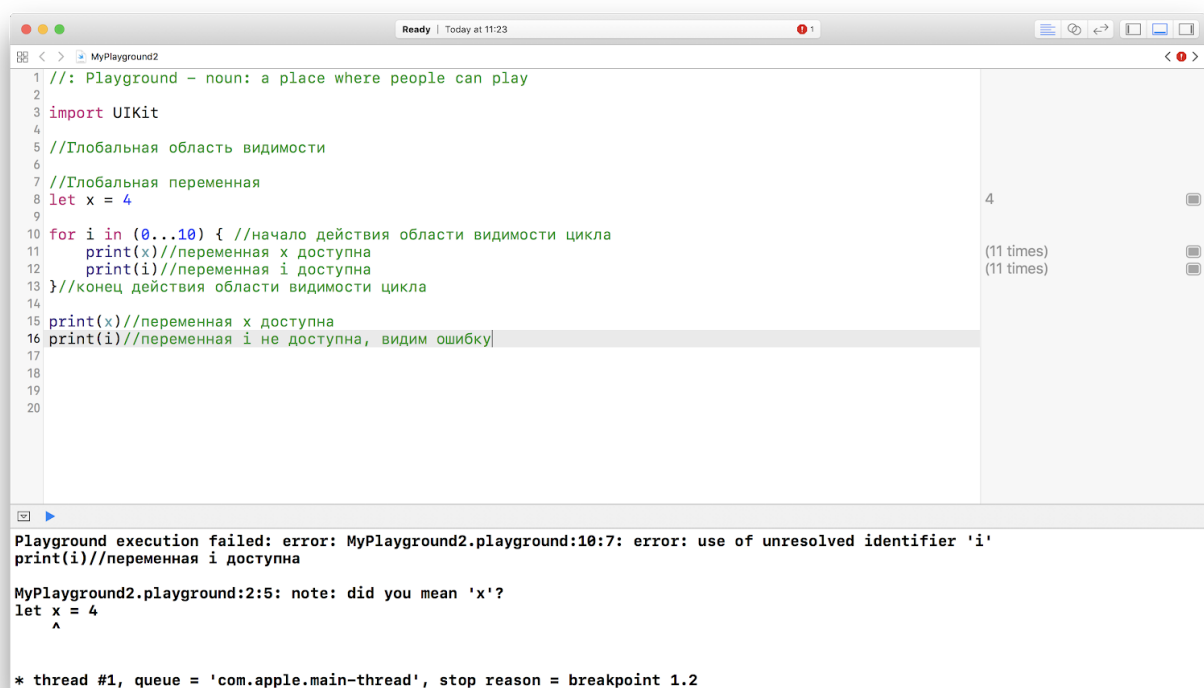
Любой параметр можно сделать передаваемым по ссылке, а не по назначению. Другими словами, он будет не копироваться внутрь функции, а передаваться как есть, и его можно будет менять. Для этого параметр надо пометить ключевым словом `inout`, а передаваемую переменную – символом `&`.



Область видимости

На данном уроке мы использовали огромное количество операторов, создающих **область видимости**. Областью видимости можно считать блок (зону) между двумя фигурными скобками «{}». Области видимости находятся одна в другой, как матрешки. Самая верная, «глобальная» область – это ваши файлы до каких-либо скобок.

Область видимости определяет доступность и время жизни переменных. Переменные, объявленные внутри какой-либо области, доступны внутри нее и всем вложенным в нее областям, но недоступны за пределами этой области. Фактически переменные уничтожаются, как только выполнение программы выходит за пределы области видимости. Переменные, объявленные в глобальной области видимости, называют **глобальными**. Они доступны в любом месте программы и уничтожаются, только когда закрывается приложение.



```
1 //: Playground - noun: a place where people can play
2
3 import UIKit
4
5 //Глобальная область видимости
6
7 //Глобальная переменная
8 let x = 4
9
10 for i in (0...10) { //начало действия области видимости цикла
11     print(x)//переменная x доступна
12     print(i)//переменная i доступна
13 } //конец действия области видимости цикла
14
15 print(x)//переменная x доступна
16 print(i)//переменная i не доступна, видим ошибку
17
18
19
20
```

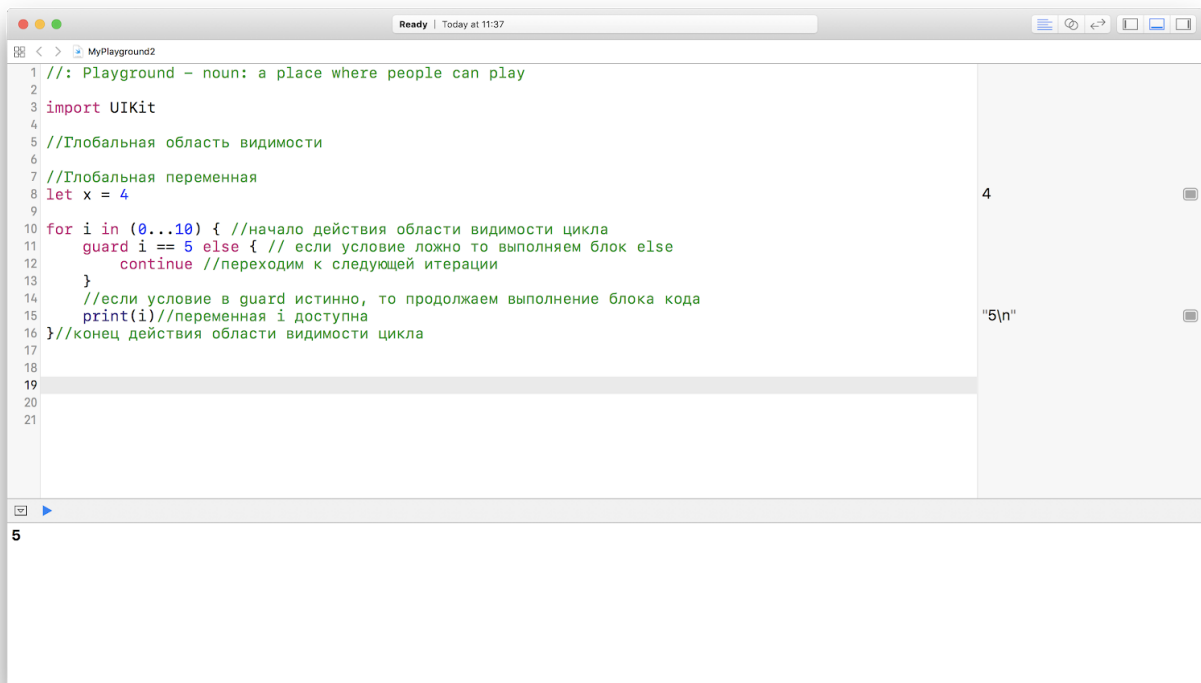
Playground execution failed: error: MyPlayground2.playground:10:7: error: use of unresolved identifier 'i'
print(i)//переменная i доступна

MyPlayground2.playground:2:5: note: did you mean 'x'?
let x = 4
^

* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.2

Еще одной особенностью области видимости является то, что ее можно покинуть. При этом выполнение кода внутри области прервется, а программа продолжит выполняться в месте выхода из этой области. Таких оператора четыре:

- «break» – используется в циклах, немедленно прерывает выполнение цикла.
- «continue» – используется в циклах, немедленно переходит к следующей итерации цикла.
- «return» – используется внутри методов и функции, возвращает значение и покидает область видимости.
- «guard» – проверяет условие; если условие ложно, покидает область видимости. Важно понимать, что сам по себе «guard» область видимости покинуть не может, вы должны сами реализовать способ выхода в блоке «else», указав одну из подходящих конструкций, перечисленных выше.



Домашнее задание

Формат файла ДР: «2I_ФИ.playground».

1. Написать функцию, которая определяет, четное число или нет.
2. Написать функцию, которая определяет, делится ли число без остатка на 3.
3. Создать возрастающий массив из 100 чисел.
4. Удалить из этого массива все четные числа и все числа, которые не делятся на 3.

Практика

Задача 1. Создать массив из 10 элементов:

Решение:

```

var testArray: [Int] = []
for i in 0...9 {
    testArray.append(i)
}

```

Задача 2. Сделать все элементы этого массива четными.

Решение:


```
var countI = 0
for (index, value) in testArray.enumerated() {
    if (value % 2) > 0 {
        testArray[index] += 1
    }
}
```

Задача 3

Удалить из исходного массива все нечетные элементы.

Решение:

```
for (index, value) in testArray.enumerated() {
    if (value % 2) > 0 {
        testArray.remove(at: testArray.index(of: value)!)
    }
}
```

Дополнительные материалы

1. https://developer.apple.com/library/prerelease/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309