2

УЧИМСЯ ПРОГРАММИРОВАТЬ В XCODE PLAYGROUND

Конечно, только что созданное вами приложение *Hello World* — уже большой успех. Но пришло время научиться серьезному программированию. В *Xcode* можно создавать особый тип документа — *Playground* («учебная площадка», или просто «площадка»). С его помощью мы научимся программировать в среде *Swift*. Если внутри площадки записать строки программы, станет видно, что происходит при их работе. Причем для этого не придется создавать приложение целиком, как мы делали в главе 1.

Откройте *Playground*. Запустите программу *Xcode* и выберите ***Get started with a playground*** («Начните с площадки») в диалоговом окне ***Welcome to Xcode***, как показано на рис. 2.1. Если это окно не открывается автоматически при запуске *Xcode*, выберите вариант ***Welcome to Xcode*** («Добро пожаловать в *Xcode*») в строке меню ***Window*** или нажмите одновременно клавиши ⌘**-shift-1**.

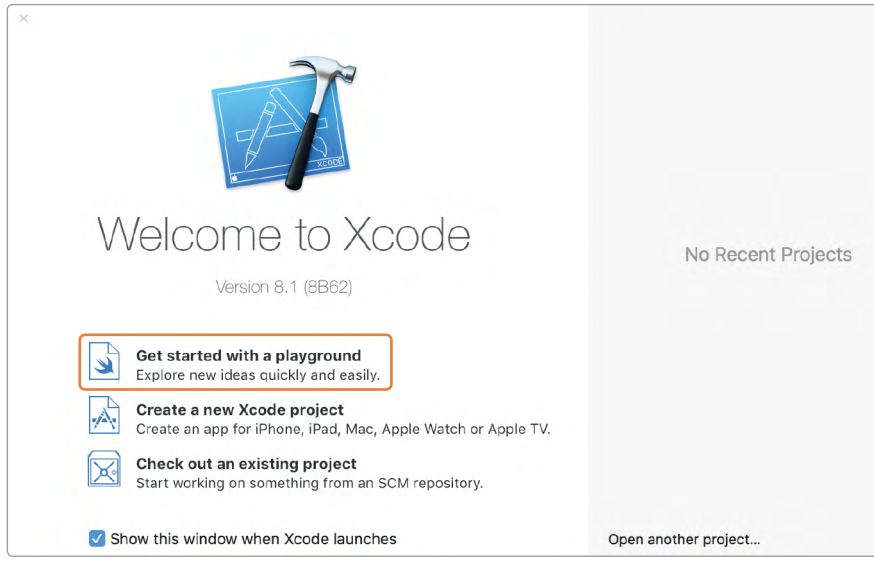


Рис. 2.1. Начало работы с площадкой

*В Xcode 9 появится окно выбора шаблона Playground. Выберите шаблон Blank

Компьютер попросит дать площадке название (рис. 2.2)*. В этом примере мы сохраняем данное по умолчанию название *MyPlayground,* но вы можете назвать свою площадку как хотите. Убедитесь в том, что выбрали *iOS* в качестве платформы для работы площадки.

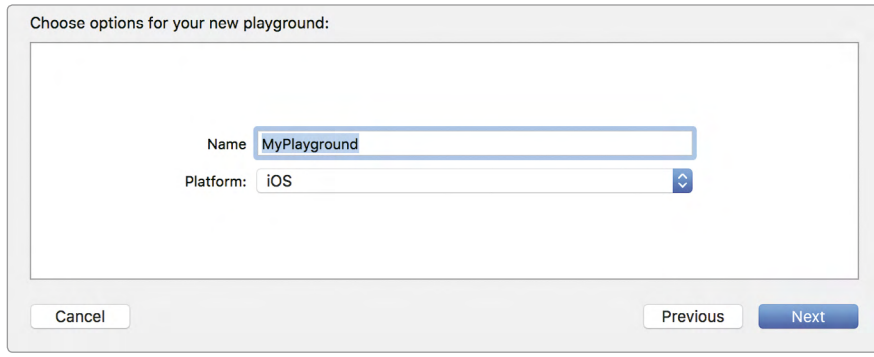


Рис. 2.2. Даем название площадке и выбираем платформу

Var — Сокр. от variable, переменный

Str — Сокр. от string, строка

При первом открытии площадки вы увидите в окне две панели, как на рис. 2.3. Слева — редактор площадки, в котором вы будете создавать программу. А справа — боковая панель результатов, где демонстрируются результаты работы вашей программы.

Строка `var str = "Hello, playground"` на рис. 2.3 создает переменную `(var)` с названием `str`. ***Переменную*** можно сравнить

с контейнером, в котором можно хранить обычные числа, последовательности чисел или составные объекты (о том, что это такое, мы поговорим позже).

Посмотрим, как работают переменные.

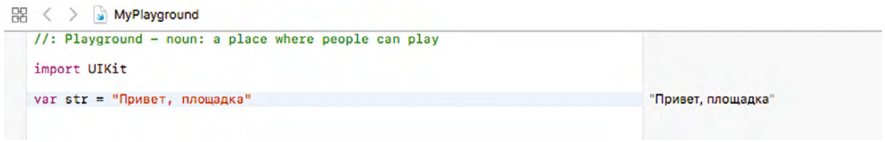


Рис. 2.3. Редактор площадки и боковая панель результатов

Playground — noun: a place where people can play — Площадка (сущ.) — место, где играют дети

Константы и переменные

Давайте вспомним, как выглядит строка программы из рис. 2.3:

```
var str = "Привет, площадка"
```

```
"Привет, площадка"
```

Для чего нужна эта строка? Во-первых, она создает переменную с названием `str`. Такое действие называется ***объявлением***, поскольку мы объявляем о том, что хотим создать переменную: печатаем слово `var` и название переменной — `str`.

Во-вторых, задает значение `"Привет, площадка"` для `str` с использованием оператора `=`. Такое действие называется ***присваиванием***, поскольку мы присваиваем значение нашей только что созданной переменной. Помните, мы сравнили переменную с контейнером? У нас появляется контейнер с названием `str` и значением `"Привет, площадка"`.

Эту строку программы можно читать как *«переменной str присвоить “Привет, площадка”»*. Как видите, программы в *Swift* довольно легко читать; эта строка программы говорит вам о происходящем почти нормальным языком.

Использовать переменные очень удобно. Если вы хотите напечатать слова «Привет, площадка», то все, что вам нужно сделать, — это использовать команду `print` («печатать») с аргументом `str`, как в приведенной ниже строке:

```
print(str)
```

```
"Привет, площадка\n"
```

Эта строка позволяет напечатать результат `"Привет, площадка\n"` в боковой панели. Символы `\n` автоматически добавляются к концу

любого текста, который вы печатаете. Их обычно называют ***символами новой строки***, они дают команду компьютеру перейти на следующую строку.

Чтобы увидеть точные результаты работы вашей программы, как если бы ее запустили по-настоящему, откройте область отладки, которая появляется ниже двух панелей, как показано на рис. 2.4. Для этого зайдите в меню ***View ▸ Debug Area ▸ Show Debug Area*** («Просмотр ▸ Область отладки ▸ Показывать область отладки») в *Xcode* или нажмите одновременно клавиши **⌘-shift-Y**.

Распечатав `str` в консоли в области отладки, вы увидите, что вокруг текста «Привет, площадка» появились кавычки, а символов новой строки нет. Именно так будет выглядеть `str` при нормальном, «официальном» запуске программы.

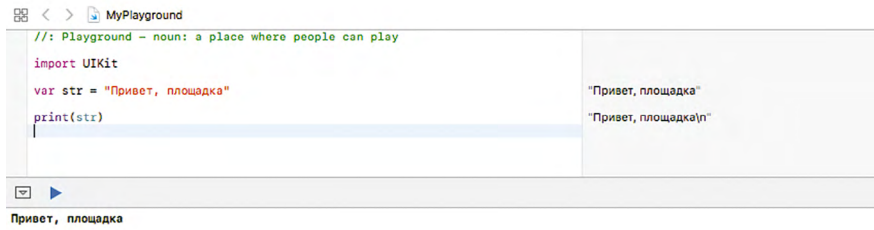


Рис. 2.4. Так будет выглядеть ваша программа при отображении в области отладки

Значения переменных можно менять. Попробуем это сделать. Добавьте в вашей площадке следующие строки:

```
❶ str = "Привет всем"                    "Привет всем"
  print(str)                             "Привет всем\n"
```

Введите название переменной и используйте оператор = для присвоения ей нового значения. Сделаем это в строке ❶, чтобы изменить значение `str` на `"Привет всем"`. Компьютер стирает все, что содержалось в `str` до этого, и говорит: «Так точно, шеф, теперь значение `str` равно `"Привет всем"`». Точнее, он мог бы так сказать, если бы умел говорить!

Обратите внимание: когда мы меняем значение `str`, то `var` еще раз не пишем! Компьютер помнит, что `str` уже существует. Мы просто хотим присвоить ей другое значение.

Так же можно объявлять ***константы***, которые, как и переменные, содержат значения. В отличие от переменной константа — величина постоянная, то есть никогда не меняет значение.

Объявление константы выглядит аналогично объявлению переменной, однако мы используем вместо слова `var` другое — `let` («пусть»):

```
let myName = "Глория"
```

```
"Глория"
```

В данном случае мы создаем константу с названием `myName` и присваиваем ей значение `"Глория"`.

После того как вы создадите константу и присвоите ей значение, она сохранит это значение навсегда. Константу можно себе представить в виде большого камня, на котором высекли символы. Если вы попытаетесь присвоить `myName` другое значение, например `"Мэтт"`, то получите примерно такую же ошибку, какая показана на рис. 2.5.

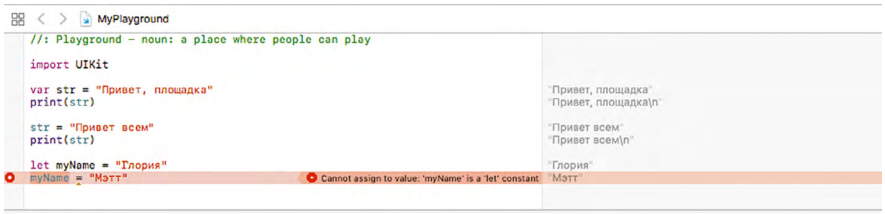


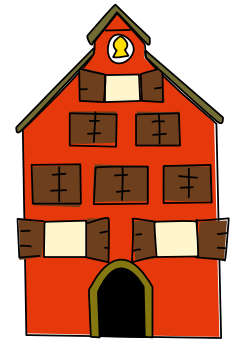
Рис. 2.5. Изменить значение константы не удастся

> *При работе в Playground символ возникающей ошибки — красный круг с маленьким белым кружком внутри. При нажатии на него вы увидите описание ошибки, а иногда и рекомендации по ее устранению.*

Когда использовать константы или переменные

Итак, вам удалось создать переменную и константу — отличная работа! Но когда нужно использовать переменную, а когда — константу? При работе в *Swift* лучше использовать константы вместо переменных, если вы не планируете менять значение. Константы помогают сделать программу «безопаснее». Если вы знаете, что что-то никогда не изменится, почему бы не выбить это на камне?

Предположим, вы хотите отслеживать общее количество окон в вашей комнате и их количество, открытых в тот или иной день. Общее количество окон в комнате не будет меняться, поэтому для хранения этого значения стоит использовать константу. Число открытых окон будет меняться в зависимости от погодных условий и времени суток, поэтому для хранения этого значения нужно использовать переменную.



Number of windows — Число окон

Number of windows open — Число открытых окон

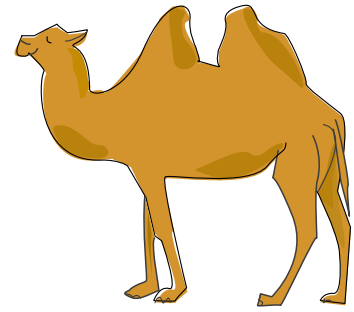
```
let numberOfWindows = 8
var numberOfWindowsOpen = 3
```

```
8
3
```

Объявляем `numberOfWindows` константой и присваиваем ей значение `8` (общее количество окон всегда одинаково). `numberOfWindowsOpen` объявляем переменной со значением `3` (оно может меняться, когда мы открываем или закрываем окна).

Используйте `var` для переменных и `let` для констант!

Как давать названия константам и переменным



Переменную или константу можно называть как угодно, но только не словами, которые используются самим *Swift*. К примеру, вы не можете назвать переменную словом `var`. Запись `var var` может привести в замешательство и вас, и компьютер. Если вы попытаетесь назвать переменную или константу словом, зарезервированным *Swift,* то у вас возникнет ошибка. Также в одном блоке программы «не уживутся» две переменные или константы с одним именем.

Существует еще целый ряд рекомендаций, которым стоит следовать при присвоении названий объектам в *Swift*. Они всегда начинаются со строчной буквы. Не бойтесь использовать длинные названия, избегайте сокращений. Так будет проще разобраться, зачем нужна переменная или константа. Если бы вы изучали чужую программу, что для вас было бы понятнее — `numKids` или `numberOfKidsInMyClass`? Первое название совсем загадочное, а второе переводится с английского как «Количество детей в моем классе».

Часто встречаются переменные и константы, названия которых состоят из нескольких слов, написанных с прописных букв, но слитно, как в предыдущем примере, — `numberOfKidsInMyClass`. Такой стиль называется ***верблюжьим***, потому что чередование строчных и прописных букв напоминает горбы на спине верблюда.

Типы данных

При работе *Swift* вы можете выбирать, какие виды (или ***типы***) данных хотите хранить в каждой переменной или константе. После того как вы скажете компьютеру, какому типу данных должна соответствовать переменная или константа, он не позволит присвоить этой переменной или константе значения другого типа. Вы же не станете наливать воду в корзину, в которой хранится картофель!

Объявление типов данных

Сообщить компьютеру, для какого типа данных предназначена переменная или константа, можно при ее создании. В примере с окнами переменная всегда будет *целым числом,* `Int` (окно ведь не делится на части). Скажем об этом компьютеру:

```
var numberOfWindowsOpen: Int = 3
```

```
3
```

Двоеточие означает «*имеет тип*». Эта строка программы говорит: «Значение переменной `numberOfWindowsOpen`, представляющей собой целое число, равно 3». Иными словами, она создает переменную с именем, сообщает компьютеру, какой тип данных она будет содержать, а затем присваивает ей значение. И все это смогла сделать одна-единственная строчка программы! Но мы уже говорили, что *Swift* — очень *ясный* язык! В некоторых других языках программирования для того же потребуется написать несколько строк. *Swift* сконструирован так, что вы можете задавать в одной строке сразу несколько вещей.

Тип данных достаточно объявить один раз. Если попытаться это сделать повторно, *Xcode* выдаст ошибку. Как только вы объявили тип данных, переменная или константа сохранит его навсегда.

Обратите внимание: переменная или константа не может содержать значения, не принадлежащие к определенному типу. Например,

если вы попытаетесь присвоить десятичную дробь переменной `numberOfWindowsOpen`, то получите сообщение об ошибке, как показано на рис. 2.6.

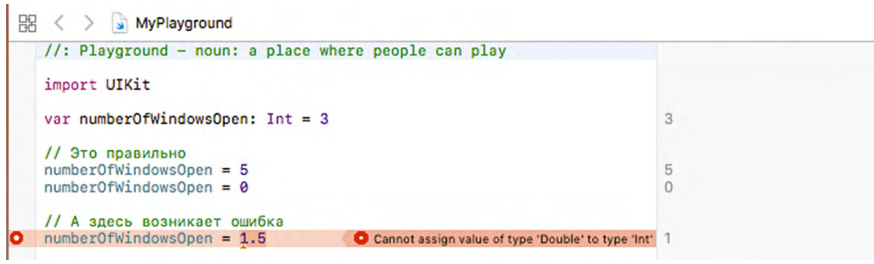


Рис. 2.6. Вы не можете присвоить десятичную дробь переменной типа Int

Вполне логично использовать значения `numberOfWindowsOpen = 5` и `numberOfWindowsOpen = 0`. Но вы не можете установить значение `numberOfWindowsOpen = 1,5`*.

*В Swift дробную часть нужно писать после точки, а не после запятой, как в русском языке

Распространенные типы данных

Мы разобрались, что тип данных позволяет компьютеру узнавать, с какими именно *видами* данных он будет работать и как хранить их в памяти. Но какими бывают эти типы данных? Самые распространенные — `Int`, `Double`, `Float`, `Bool` и `String`.

Рассмотрим каждый из них.

Int (целые числа)

Поговорим о целых числах чуть подробнее. Целое число, обозначаемое в *Swift* как `Int` (сокр. от *integer* — «целое число»), — это число, у которого нет дробной части. Это числа, которые мы обычно используем для счета. Целые числа имеют *знак*, то есть могут быть положительными и отрицательными (или равными нулю).

Double и Float (числа с дробной частью)

Для записи дробных чисел, таких как 3,14, в *Swift* есть два типа данных: `Double` (сокр. от *double precision* — «двойная точность») и `Float` (сокр. от *floating-point number* — «число с плавающей точкой»)**. Тип данных `Double` более распространен в *Swift*, поскольку может описывать числа большего размера, поэтому давайте сосредоточимся на нем.

**Целое число типа Double или Float можно записать с нулем после точки: 3.0

У числа, принадлежащего типу `Double`, обязательно должна быть хотя бы одна цифра слева от разделительного знака. В противном случае вы получите ошибку в процессе работе программы. Предположим, что цена одного банана 19 центов:

```
❶ var bananaPrice: Double = .19 // Ошибка
❷ var bananaPrice: Double = 0.19 // Правильно
```

```
0.19
```

Banana price — Цена банана

В строке ❶ программы возникнет ошибка, поскольку у введенного значения нет цифры слева от разделительного знака. В строке ❷ этого не произойдет, поскольку перед разделительным знаком стоит ноль. (Фразы «`// Ошибка`» и «`// Правильно`» представляют собой *комментарии,* то есть заметки внутри программы, которые игнорируются компьютером. См. раздел «Небольшой комментарий к комментариям» на с. 51.)

Bool (булев тип, или значения True/False)

Так называемые *булевы значения* бывают двух видов: истинные и ложные.

```
let swiftIsFun = true
var iAmSleeping = false
```

```
true
false
```

Swift is fun — Swift классный

I am sleeping — Я сплю

True — Истинный

False — Ложный

Такие значения часто используются в выражениях `if-else` («если-иначе») для того, чтобы сообщить компьютеру, по какому пути должна двигаться программа (об операторах типа `Bool` и выражениях `if-else` мы детально расскажем в главе 3).

String

Тип данных `String` («Строка») используется для хранения слов и фраз. *Строка* — это набор символов, заключенных в кавычки. Например, `"Привет, площадка"` — строка.

Строки могут состоять из разных символов: букв, чисел, знаков препинания и так далее. Кавычки очень важны, поскольку они сообщают компьютеру, что любая информация, заключенная в них, — часть формируемой строки.

Используя строки, можно создавать целые предложения, записывая их одну за другой. Этот процесс называется «конкатенация». Посмотрим, как он работает.

Morning greeting — Утреннее приветствие

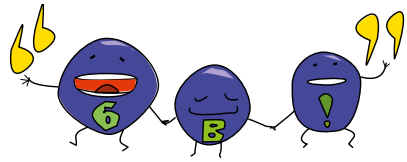
Friend — Друг

Special greeting — Особое приветствие

```
let morningGreeting = "Доброе утро"
let friend = "Джуд"
let specialGreeting = morningGreeting + ", " +
friend
```

```
"Доброе утро"
"Джуд"
"Доброе утро, Джуд"
```

При соединении строк с помощью знака «плюс» (+) программа создает переменную `specialGreeting` со значением `"Доброе утро, Джуд"`. Обратите внимание: если между `morningGreeting` и именем друга вы не добавите строку, содержащую запятую и пробел (`", "`), `specialGreeting` будет выглядеть как `"Доброе утроДжуд"`.



Вывод типа

Возможно, вы заметили, что иногда при объявлении переменной мы добавляем тип данных:

```
var numberOfWindowsOpen: Int = 3
```

```
3
```

А иногда мы этого не делаем:

```
var numberOfWindowsOpen = 3
```

```
3
```

В чем разница? Компьютер достаточно умен для того, чтобы понять, к какому типу относятся данные. Это действие называется ***выводом типа*** *(type inference),* потому что компьютер будет делать вывод, то есть догадываться о том, какой тип данных мы используем, на основании наших подсказок. Когда вы создаете переменную и присваиваете ей значение, это здорово помогает компьютеру. Вот несколько примеров:

- если вы зададите число без десятичного знака, например 3, компьютер отнесет его к типу `Int`;
- с десятичным знаком, например 3.14, — к типу `Double`;
- слова `true` или `false` (без кавычек) — к типу `Bool`;
- один или несколько символов в кавычках — к типу `String`.

Введенный тип данных присваивается переменной или константе. Вы можете задавать тип данных каждый раз при объявлении новой константы или переменной, и это не будет ошибкой. Но почему бы не позволить компьютеру самому это сделать?

Изменение типов данных с помощью приведения

Приведение, или превращение типов, — временное изменение типа данных для переменной или константы. Вы можете думать об этом как о временном волшебном превращении, после которого значение какое-то время ведет себя как иной тип данных. Чтобы сделать приведение типов, укажите новый тип данных, а затем переменную в скобках.

Обратите внимание: это не приведет к *реальному изменению типа данных*, а лишь даст временное значение для единственной строчки программы.

Несколько примеров приведения между `Int` и `Double`. Вот как это выглядит в боковой панели результатов:

```
let months = 12                                  12
print(months)                                    "12\n"
❶ let doubleMonths = Double(months)              12
print(doubleMonths)                              "12.0\n"
```

Months — Месяцы

В строке ❶ приводим константы переменной `months` из формата `Int` в формат `Double` и сохраняем получившееся значение в новой константе `doubleMonths`. Появляется десятичный знак, а результат приведения равен `12.0`. Так же вы можете произвести приведение из `Double` в `Int`:

```
let days = 365.25                                365.25
❶ Int(days)                                      365
```

Days — Дни

В строке ❶ производим приведение значения `days` в формате `Double` к формату `Int`. Исчезает и разделительный знак, и все цифры после него: наше число превращается в 365. Это происходит потому, что тип данных `Int` не позволяет сохранять десятичные дроби, он может хранить только целые числа.

Отметим, что приведение не меняет типа данных. В нашем примере даже после приведения `days` все еще относится к типу `Double`. Это можно проверить, если распечатать его значение:

```
print(days)
```
```
"365.25\n"
```

В боковой панели результатов показано, что величина `days` все еще равна `365,25`. В следующем разделе рассмотрим на примерах, где и когда использовать приведение. Так что, если вам пока непонятно, зачем это нужно, наберитесь терпения!

Операторы

Для математических вычислений в *Swift* используются арифметические операторы: присваивания (=) , сложения (+), вычитания (–), умножения (*) и деления (/).

Их можно применять с типами данных `Int`, `Float` и `Double`. Числа, над которыми проводятся действия, называются ***операндами***. Поэкспериментируйте с ними в своей площадке, например введите следующий код:

```
6.2 + 1.4
3 * 5
16-2
9 / 3
```
```
7.6
15
14
3
```

Результаты каждого математического выражения отобразятся в боковой панели. Форма математических выражений в программе не отличается от их обычной записи. Например, 16 минус 2 записывается как `16-2`.

Если сохранить результат в переменной или константе, вы сможете использовать его в других частях программы. Чтобы понять, как это работает, введите следующие строки в свою площадку:

Sum — Сумма

Three times five — Трижды пять

```
var sum = 6.2 + 1.4
❶ print(sum)
let threeTimesFive = 3 * 5
```
```
7.6
"7.6\n"
15
```

Распечатав `sum` в строке ❶, вы увидите в боковой панели значение `7,6`. До сих пор мы использовали в своих математических

выражениях лишь числа, однако математические операторы также работают с переменными и константами.

Добавьте в свою площадку следующую программу:

```
let three = 3
let five = 5
let half = 0.5
let quarter = 0.25
var luckyNumber = 7
three * luckyNumber
five + three
half + quarter
```

```
3
5
0.5
0.25
7
21
8
0.75
```

Three, five, half,quater — 3, 5, половина, четверть

Lucky number — Счастливое число

Как видим, математические операторы в отношении переменных и констант используются точно так же, как числа.

О важности пробелов

При работе в *Swift* очень важно не забывать о пробелах. Они ставятся с обеих сторон математического оператора либо не ставятся нигде. Если пробел есть только с одной стороны, это приведет к возникновению ошибки. Посмотрите на рис. 2.7.

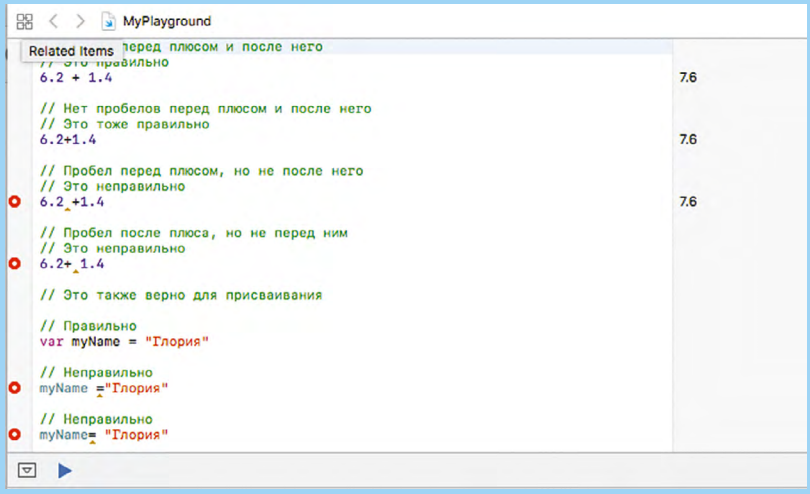


Рис. 2.7. Убедитесь, что количество пробелов с обеих сторон ваших операторов одинаково

Обратите внимание: математический оператор используется в отношении переменных или констант *одного и того же* типа данных. В описанной выше программе `three` и `five` относятся к типу данных `Int`, константы `half` и `quarter` — к `Double`, поскольку имеют десятичный знак. Если вы попытаетесь сложить или перемножить типы данных `Int` и `Double`, возникнет такая ошибка, как показано на рис. 2.8.

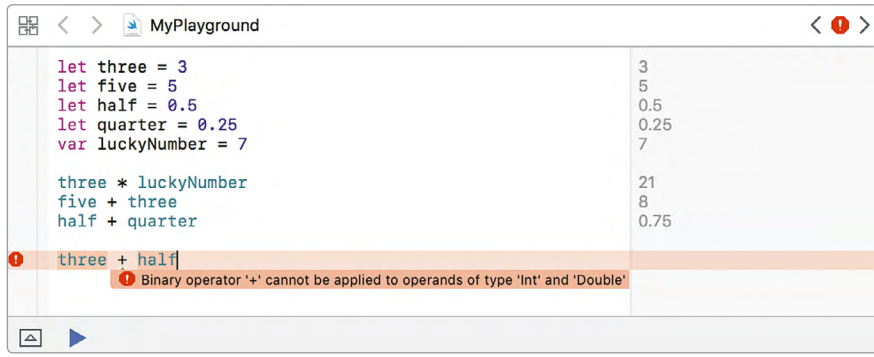


Рис. 2.8. При работе в Swift вы не можете производить математические операции над смешанными типами данных

Но что делать, если вам нужно произвести математические вычисления с данными различных типов? Предположим, вы хотите рассчитать 1/10 от своего возраста:

My age — Мой возраст

Multiplier — Множитель

One tenth my age — Одна десятая от моего возраста

```
var myAge = 11 // это тип Int
let multiplier = 0.1 // это тип Double
var oneTenthMyAge = myAge * multiplier
```

```
11
0.1
```

Выполнение последней строки команды приведет к появлению ошибки, поскольку мы пытаемся умножить `Int` на `Double`. Но не беспокойтесь! Есть пара вариантов, позволяющих привести операнды к одному типу данных. Первый вариант: объявлять тип данных `myAge` как `Double` следующим образом:

```
var myAge = 11.0 // это тип Double
let multiplier = 0.1 // это тип Double
var oneTenthMyAge = myAge * multiplier
```

```
11.0
0.1
1.1
```

Эта программа будет работать, ведь мы перемножаем типы данных `Double`.

Второй вариант: использовать приведение, поскольку мы не планируем менять тип данных для `myAge` на `Double`, а лишь хотим произвести с ним математические вычисления, как если бы он имел тип `Double`.

Рассмотрим пример:

```
var myAge = 11 // это тип Int
let multiplier = 0.1 // это тип Double
❶ var oneTenthMyAge = Double(myAge) * multiplier
❷ oneTenthMyAge = myAge * multiplier
```

```
11
0.1
1.1
```

В строке ❶ приводим `myAge` к `Double`. В строке ❷ возникает ошибка, потому что `myAge` все еще принадлежит к `Int`. Приведение его к типу `Double` в строке ❶ постоянного изменения не повлекло.

Можем ли мы привести значение `multiplier` к типу `Int`? Конечно! Однако это приведет к менее точным расчетам, поскольку при приведении переменной из `Double` в `Int` компьютер убирает все цифры после десятичного знака, превращая число в целое. Значение `multiplier`, равное `0,1`, усечется в типе `Int` до `0`.

Давайте поэкспериментируем с приведением нескольких переменных внутри площадки и посмотрим, что получится:

```
❶ Int(multiplier)
❷ Int(1.9)
```

```
0
1
```

В строке ❶ приведение `multiplier` из типа `Double` в тип `Int` дает `0`. Это значение оказывается иным после приведения, поскольку у нас исчезает десятичный знак и `0.1` превращается в `0`. И если мы не учтем этого в нашей программе, то может случиться всякое.

К процессу приведения нужно относиться очень осторожно и проверять, не привело ли это действие к неожиданному изменению значений. В строке ❷ содержится еще один пример приведения `Double` в `Int`, в котором `1.9` не округляется до `2`, а превращается в `1` из-за того, что цифры после десятичного знака просто исключаются.

В *Swift* существует еще один математический оператор — *оператор нахождения остатка* (*modulo*, он же *modulus*, %), который позволяет получить остаток после деления. Например, при делении 7 на 2 образуется остаток, равный 1 (7 % 2 = 1).

Попробуйте поэкспериментировать с оператором остатка в площадке:

```
10 % 3                          1
12 % 4                          0
34 % 5                          4
var evenNumber = 864            864
❶ evenNumber % 2                0
var oddNumber = 571             571
❷ oddNumber % 2                 1
```

Как видим, оператор нахождения остатка может использоваться для определения четных (`evenNumber % 2` равен `0` в строке ❶) и нечетных (`oddNumber % 2` равен `1` в строке ❷) чисел.

Порядок действий

До сих пор мы совершали одно математическое действие в каждой строке программы, но *Swift* позволяет включать в одну строку сразу несколько. Рассмотрим пример.

Допустим, у вас 3 купюры по 5 долларов и 2 купюры по 1 доллару. Сколько всего у вас денег? Сделаем расчет в одной строке:

My money —
Мои деньги

```
var myMoney = 5 * 3 + 2          17
```

В `myMoney` оказывается значение `17`. Компьютер умножает 5 на 3, а затем прибавляет 2. Обратите внимание: он не просто выполняет действия слева направо, а знает, что сначала нужно перемножить числа `5` и `3`, а лишь *затем* прибавить `2`. Посмотрите на следующее выражение:

```
myMoney = 2 + 5 * 3              17
```

Мы поменяли операции местами, но результат по-прежнему равен `17`. Если бы компьютер просто выполнял программу слева направо и сложил бы 2 и 5, получилось бы 7. Затем он умножил бы этот результат, равный 7, на 3, и получилось бы 21. Компьютер знаком с порядком действий и всегда сначала производит умножение и деление, затем — сложение и вычитание.

Задание порядка с помощью скобок

Как вы знаете из школьной программы, чтобы изменить порядок действий, используют скобки. Те же правила работают и в программировании. Поставив скобки вокруг какого-то выражения, вы сообщаете компьютеру, что этот шаг нужно сделать первым:

```
❶ myMoney = 2 + (5 * 3)
❷ myMoney = (2 + 5) * 3
```

```
17
21
```

В строке ❶ скобки приказывают компьютеру сначала умножить 5 на 3, а затем прибавить 2. В результате вы получите 17. В строке ❷ скобки приказывают компьютеру сначала сложить 2 и 5, а затем умножить результат на 3, что дает нам 21. Вы можете усложнять свою программу, помещая одни скобки внутри других. Компьютер сначала произведет действие во внутренних скобках, потом во внешних. Потренируемся на примере:

```
myMoney = 1 + ((2 + 3) * 4)
```

```
21
```

Сначала компьютер сложит 2 и 3 во внутренних скобках, затем умножит результат на 4, поскольку это действие находится внутри внешних скобок. И напоследок прибавит к результату 1, так как это действие — за пределами обеих пар скобок. Результат равен 21.

Составные операторы присваивания

Еще одна категория операторов, которую вы будете использовать, называется *составными операторами присваивания*. Это своего рода «короткий путь», совмещающий математический оператор с оператором присваивания (=). Рассмотрим пример:

```
a = a + b
```

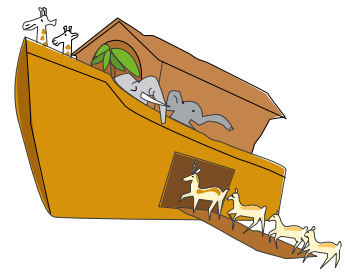
превращается в

```
a += b
```

Эти операторы нужны для обновления значения переменной после проведения над ней какого-либо действия. Выражение `a += b` говорит нам: «прибавьте значение `b` к `a` и сохраните новое значение в `a`». В табл. 2.1 приведены математические выражения с составными операторами присваивания и те же выражения в длинной форме.

Таблица 2.1. Операторы присваивания в короткой и длинной форме

Короткая форма	Длинная форма
`a += b`	`a = a + b`
`a -= b`	`a = a – b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`



Animals on ark — Животные на ковчеге

Посмотрим, как работает оператор +=. Допустим, вы хотите написать программу для расчета количества животных на Ноевом ковчеге. Сначала создаем переменную с названием `animalsOnArk` и устанавливаем для нее исходное значение, равное `0`, поскольку на ковчеге пока нет никого. По мере заполнения ковчега увеличиваем значение `animalsOnArk` и таким образом считаем животных. На ковчег поднимаются два жирафа — к значению `animalsOnArk` нужно добавить 2. За ними идут два слона — опять прибавляем 2. Еще две пары антилоп — увеличиваем значение `animalsOnArk` на 4.

Number of giraffes — Число жирафов

Number of elephants — Число слонов

Number of antelopes — Число антилоп

```
var animalsOnArk = 0                                    // 0
let numberOfGiraffes = 2
animalsOnArk += numberOfGiraffes                        // 2
let numberOfElephants = 2
animalsOnArk += numberOfElephants                       // 4
let numberOfAntelopes = 4
animalsOnArk += numberOfAntelopes                       // 8
```

Итак, на ковчеге два жирафа, два слона и четыре антилопы. Значение `animalsOnArk` — `8`. Настоящий зоопарк!

Небольшой комментарий к комментариям

Языки программирования, как правило, позволяют создавать комментарии рядом с программным кодом. Это заметки, которые пишутся для себя, а не для компьютера. Они игнорируются компьютером и предназначены лишь для того, чтобы помочь людям, изучающим программу, понять, о чем идет речь. Хотя программа будет работать и без комментариев, полезно включить их для частей программы, которые могут смутить будущих читателей. И даже если вы не собираетесь показывать вашу программу кому-то еще, комментарии помогут вспомнить, о чем вы думали, когда ее писали, почему использовали тот или иной код. Часто бывает так, что вы возвращаетесь к коду, который написали несколько месяцев или лет назад, и не помните ход своих мыслей.

Добавить комментарий в *Swift* можно двумя способами. Первый — поставить две косые черты, наклоненные вправо (//), перед текстом комментария. Его можно поместить на отдельной строке, например, так:

```
// Мои любимые вещи
```

Или расположить в той же строке кода:

```
var myFavoriteAnimal = "Лошадь" // Не обязательно одомашненное
                                   животное
```

My favorite animal — Мое любимое животное

Второй способ годится для длинных или многострочных комментариев. Начало и конец комментария отмечаются знаками `/*` и `*/`.

```
/*
В этой части программы мы считаем животных, входящих на ковчег.
*/
{
var animalsOnArk = 0
let numberOfGiraffes = 2
animalsOnArk += numberOfGiraffes
--snip--
}
```

Snip — Пропуск. Мы будем использовать это слово, когда пропускаем несколько строк кода для экономии места

Многострочные комментарии полезны при отладке программы. Например, вы пытаетесь найти ошибку в программе и не хотите, чтобы какие-то ее части запускались. Но вы не хотите и удалять их. Тогда вы можете временно заключить эти части в многострочные комментарии. Компьютер проигнорирует код, который вы сделаете в виде комментария. А когда найдете ошибку, можете раскомментировать обратно.

Что вы узнали

Из этой главы вы узнали, как создавать программы в площадке *Swift* так, чтобы сразу же видеть результаты. Создали переменные и константы, а также научились использовать основные типы данных и операторы.

Прочитав главу 3, вы научитесь управлять компьютером с помощью условных выражений.

издательство
МАНН, ИВАНОВ И ФЕРБЕР
Максимально полезные книги