Практическая работа № 12

Сабскрипты и Опциональная последовательность

Цель работы:

1. Научиться создавать и использовать опциональные последовательности.
2. Научиться создавать и использовать сабскрипты.
3. Приобрести навыки работы с сабскриптами и опциональными последовательностями.

Краткие теоретические сведения

Сабскрипты

Классы, структуры и перечисления могут определять сабскрипты (subscripts). Сабскрипты используются для доступа к элементам коллекции или последовательности. В ряде языков программирования есть похожая концепция - индексаторы. Сабскрипты позволяют обращаться с объектом класса или структуры как с отдельной коллекцией.

Для определения сабскрипта используется ключевое слово **subscript**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | subscript(параметры) ->тип\_возвращаемых\_объектов {        get{            // возвращаем значение      }      set(newValue) {            // устанавливаем новое значение newValue      }  } |

После ключевого слова subscript в скобках идут параметры, которые используются для получения элементов. Нередко параметры представляют числовой индекс, по которому надо получить объект. Далее указывается тип элементов, с которыми мы работаем.

Сабскрипт может состоять из двух блоков: get и set. Блок get возвращает элемент, а блок set устанавливает новое значение, которое передается через параметр newValue.

Например, создадим класс библиотеки. Упрощенно библиотека представляет некоторый набор книг. То есть мы можем представить класс библиотеки как коллекцию книг и использовать сабскрипты для получения книг по индексу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | classBook{     // класскниги        varname: String      init(name: String){            self.name = name      }  }  classLibrary{      // классбиблиотеки        varbooks: [Book] = [Book]()        init(){            books.append(Book(name: "Войнаимир"))          books.append(Book(name: "Отцыидети"))          books.append(Book(name: "Чайка"))      }        subscript(index: Int) -> Book{            get{              returnbooks[index]          }          set(newValue){              books[index] = newValue          }      }  }    varmyLibrary: Library = Library()  varfirstBook: Book = myLibrary[0]  // получаемэлементпоиндексу 0  print(firstBook.name)   // Войнаимир    myLibrary[2] = Book(name:"Мартин Иден")    // установка элемента по индексу 2  print(myLibrary[2].name)    // МартинИден |

Здесь сабскриптпредназначен для работы с типом Book. В блоке get происходит получение объекта Book по индексу из массива books. В блоке set устанавливаем объект Book в массиве books.

В итоге в программе мы сможем обращаться к библиотеке как к массиву по индексу для получения нужной книги:

|  |  |
| --- | --- |
| 1 | varfirstBook: Book = myLibrary[0] |

Есть два типа сабскриптов:

* Сабскрипты, которые поддерживают чтение и запись (то есть с блоками get и set, как в примере выше)
* Сабскрипты только для чтения (только с блоком get)

Изменим класс библиотеки, чтобы применить сабскрипт только для чтения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Class Library{        Var books: [Book] = [Book]()        init(){            books.append(Book(name: "Войнаимир"))          books.append(Book(name: "Отцыидети"))          books.append(Book(name: "Чайка"))      }        subscript(index: Int) -> Book{            returnbooks[index]      }  } |

Опциональная последовательность

Опциональная цепочка (optionalchaining) - процесс запросов и вызовов свойств, методов, сабскриптов (индексов) у опционала, который может быть nil. Если опционал содержит какое-либо значение, то вызов свойства, метода или сабскрипта успешен, и наоборот, если опционал равен nil, то вызов свойства, метода или сабскрипта возвращает nil. Множественные запросы могут быть соединены вместе, и вся цепочка этих запросов не срабатывает, если хотя бы один запрос равен nil.

Опциональная последовательность в Swift - аналог сообщению nil в Objective-C, но только она работает со всеми типами, и может быть проверена на успех или неудачу.

# ОП как альтернатива принудительному извлечению

Вы обозначаете опциональную последовательность, когда ставите вопросительный знак (?) опционального значения, свойство, метод или индекс которого вы хотите вызвать, если опционал не nil. Это очень похоже на установку восклицательного знака (!) после опционального значения для принудительного извлечения его значения. Основное отличие в том, что опциональная последовательность не исполняется, если опционал равен nil, в то время как принудительное извлечение приводит к runtime ошибке, когда опционал равен nil.

В частности, результат вызова опциональной последовательности того же типа, что и тип ожидаемого возвращаемого значения, только в завернутом в опционал виде. Свойство, которое обычно возвращало Int, вернет Int?, когда обращаются к нему через опциональную последовательность.

Первые два класса Person, Residence определены как:

classPerson {

var residence: Residence?

}

classResidence {

varnumberOfRooms = 1

}

Экземпляры Residence имеют единственное свойство numberOfRooms типа Int, со значением по умолчанию 1. Экземпляры Person имеют опциональное свойство residence типа Residence?.

Если вы создаете новый экземпляр Person, то его свойство residence по умолчанию имеет значение nil, в силу того, что оно является опционалом. В коде ниже john имеет свойство residence, значение которого nil:

letjohn = Person()

Если вы попытаетесь получить доступ к свойству numberOfRooms свойства residence экземпляра Person, поставив восклицательный знак после residence, для принудительного извлечения, то вы получите ошибку исполнения, потому что residence не имеет значения для извлечения:

letroomCount = john.residence!.numberOfRooms

// ошибка runtime

Код, представленный выше, срабатывает успешно, если john.residence имеет не nil значение и устанавливает корректное значение типа Int для roomCount. Однако этот код всегда будет выдавать ошибку исполнения, когда residence равен nil, что указано выше.

Опциональная последовательность предоставляет альтернативный способ получить доступ к значению numberOfRooms. Для использования опциональной последовательности используйте вопросительный знак, на месте восклицательного знака:

ifletroomCount = john.residence?.numberOfRooms {

print("John's residence has \(roomCount) room(s).")

} else {

print("Unable to retrieve the number of rooms.")

}

// Выведет "Unable to retrieve the number of rooms."

# Определение классовых моделей для ОП

Можно использовать опциональную последовательность для вызовов свойств, методов, сабскриптов, которые находятся более чем на один уровень глубже. Это позволяет вам пробираться через подсвойства, внутри сложных моделей вложенных типов, и проверять возможность доступа свойств, методов и сабскриптов этих подсвойств.

Фрагмент кода ниже определяет четыре модели классов для использования в нескольких следующих примерах, включая примеры с многоуровневой опциональной последовательностью. Эти классы расширяют модели Person, Residence приведенные ранее, добавляя классы Room, Address со свойствами, методами и сабскриптами.

Класс Person объявляется так же как и раньше:

classPerson{

var residence: Residence?

}

Класс Residence стал намного сложнее, чем был раньше. В этот раз класс Residence определяет переменное свойство rooms, которое инициализировано пустым массивом [Room]:

classResidence{

var rooms = [Room]()

varnumberOfRooms: Int {

returnrooms.count

}

subscript(i: Int) ->Room {

get {

return rooms[i]

}

set {

rooms[i] = newValue

}

}

funcprintNumberOfRooms() {

print("Общееколичествокомнатравно\(numberOfRooms)")

}

varaddress: Address?

}

Из-за того, что эта версия Residence хранит массив экземпляров Room, его свойство numberOfRooms реализовано как вычисляемое, а не как хранимое свойство. Вычисляемое свойство numberOfRooms просто возвращает значение свойства.

В качестве сокращенного варианта доступа к массиву rooms, эта версия класса Residence предлагает сабскрипт (доступный как для чтения, так и для записи), который предоставляет доступ к комнате под требуемым индексом в массиве rooms.

Эта версия Residence так же обеспечивает метод printNumberOfRooms, который просто выводит на экран количество комнат в резиденции.

И наконец, Residence определяет опциональное свойство address типа Address?. Тип класса Address для этого свойства определен ниже.

Класс Room используется для массива rooms, в качестве простого класса с одним свойством name и инициализатором, в котором устанавливается значение свойства name как подходящее имя комнаты:

classRoom{

let name: String

init(name: String) {

self.name = name

}

}

Последний класс в этой модели Address. Этот класс имеет три опциональных свойства типа String?. Первые два свойства buildingName, buildingNumber являются альтернативным вариантом определения конкретного здания как части адреса. Третье свойство street используется для названия улицы, для этого адреса:

classAddress{

varbuildingName: String?

varbuildingNumber: String?

var street: String?

funcbuildingIdentifier() ->String? {

ifletbuildingNumber = buildingNumber, let street = street {

return"\(buildingNumber)\(street)"

} elseifbuildingName != nil {

returnbuildingName

} else {

returnnil

}

}

}

Класс Address так же предоставляет метод buildingIdentifier(), который возвращает String?. Этот метод проверяет значения свойств экземпляра и возвращает buildingName, если у него есть значение или возвращает buildingNumber, если у него есть значение или nil, если ни у одного из свойств нет значения.

# Доступ к свойствам через ОП

Как было показано в разделе [ОП как альтернатива принудительному извлечению](https://swiftbook.ru/content/languageguide/optional-chaininig/#optional-chaining-as-an-alternative-to-forced-unwrapping), вы можете использовать опциональную последовательность для доступа к свойству опционального значения и проверить результат доступа к этому свойству на успешность

Используйте классы, определенные ранее, для создания нового экземпляра Person и попробуйте получить доступ к свойству numberOfRooms, как вы делали ранее:

let john = Person()

ifletroomCount = john.residence?.numberOfRooms {

print("John's residence has \(roomCount) room(s).")

} else {

print("Unable to retrieve the number of rooms.")

}

// Выведет "Unable to retrieve the number of rooms."

Так как john.residence равен nil, этот вызов опциональной последовательности не будет успешен как и ранее.

Вы можете попробовать установить значение свойства через опциональную последовательность:

letsomeAddress = Address()

someAddress.buildingNumber = "29"

someAddress.street = "Acacia Road"

john.residence?.address = someAddress

В этом примере попытка установить значение свойству address опциональному свойству residence? провалится, так как john.residence все еще nil.

Присваивание является частью опциональной цепочки, что означает, что никакой код с правой стороны не вычисляется. В предыдущем примере не так просто заметить, что someAddress никогда не вычисляется, потому что доступ к самой константе не имеет никаких побочных эффектов. Пример ниже делает тоже самое присваивание, но он использует функцию для того, чтобы создать адрес. Функция выводит "Functionwascalled" до того, как вернется значение, что позволяет вам увидеть была ли вычислена правая часть от оператора присваивания.

funccreateAddress() ->Address {

print("Function was called.")

letsomeAddress = Address()

someAddress.buildingNumber = "29"

someAddress.street = "Acacia Road"

returnsomeAddress

}

john.residence?.address = createAddress()

Можно заметить, что функция createAddress() не вызывается, так как ничего в консоли у нас не выводится.

# Вызов методов через ОП

Вы можете использовать опциональную последовательность для вызова метода опциональной величины, и проверить сам вызов метода на успешность. Вы можете сделать это, даже если этот метод не возвращает значения.

Метод printNumberOfRooms класса Residence выводит текущее значение numberOfRooms. Вот как выглядит этот метод:

funcprintNumberOfRooms() {

print("Общее количество комнат равно \(numberOfRooms)")

}

Этот метод не определяет возвращаемого значения. Однако функции и методы без возвращаемого значения имеют неявный возвращаемый тип Void. Если вы вызовите этот метод на опциональном значении в опциональной последовательности, то он вернет тип не Void, а Void?, потому что возвращаемые значения всегда опционального типа, когда они вызываются через опциональную последовательность.if john.residence?.printNumberOfRooms() != nil { print("Есть возможность вывести общее количество комнат.") } else { print("Нет возможности вывести общее количество комнат.") } // Выведет "Нет возможности вывести общее количество комнат."

# Доступ к сабскриптамчерез ОП

Когда вы получаете доступ к опциональному значению через опциональную последовательность, вы размещаете вопросительный знак до скобок сабскрипта (индекса), а не после. Вопросительный знак опциональной последовательности следует сразу после части выражения, которая является опционалом.

Из-за того, что john.residence является nil, то вызов сабскрипта проваливается:

ifletfirstRoomName = john.residence?[0].name {

print("Название первой комнаты \(firstRoomName).")

} else {

print("Никак не получить название первой комнаты.")

}

// Выведет "Никак не получить название первой комнаты."

Вызов вопросительного знака опциональной последовательности в этом сабскрипте идет сразу после john.residence, но до скобок сабскрипта, потому что john.residence является опциональным значением, на которое применяется опциональная последовательность.

Аналогично вы можете попробовать установить новое значение через сабскрипт с помощью опциональной последовательности:

john.residence?[0] = Room(name: "Bathroom")

Это попытка установки значения через сабскрипт так же не срабатывает, так как residence все еще nil.

Если вы создадите и присвоите действительное значение экземпляру Residence, при помощи одного или нескольких экземпляров Room в массиве rooms, то вы сможете использовать сабскрипт на экземпляре residence для того, чтобы получить доступ к массиву rooms через опциональную последовательность:

letjohnsHouse = Residence()

johnsHouse.rooms.append(Room(name: "Гостиная"))

johnsHouse.rooms.append(Room(name: "Кухня"))

john.residence = johnsHouse

ifletfirstRoomName = john.residence?[0].name {

print("Название первой комнаты \(firstRoomName).")

} else {

print("Никак не получить название первой комнаты.")

}

// Выведет "Название первой комнаты Гостиная."

## Получение доступа к сабскрипту (индексу) опционального типа

Если сабскрипт возвращает значение опционального типа, например ключ словаря типа Dictionary в Swift, то мы должны поставить вопросительный знак после закрывающей скобки сабскрипта, для присоединения его опционального возвращаемого значения:

vartestScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]

testScores["Dave"]?[0] = 91

testScores["Bev"]?[0] += 1

testScores["Brian"]?[0] = 72

// массив "Dave" теперьимеетвид [91, 82, 84], массив "Bev" - [80, 94, 81]

Пример выше определяет словарь testScores, который содержит две пары ключ/значение, которые соединяют ключ типа String со значением типа [Int]. Пример использует опциональную последовательность для установки значения первого элемента ключа "Dave" равным 91, для увеличения первого элемента массива под ключом "Bev" на 1 и для попытки установить первое значение несуществующего массива, соответствующего ключу "Brian" равным 72. Первые два вызова завершились успешно, потому что их ключи находятся в testScores. Третий вызов завершился неудачей, так как такого ключа как "Brian" в словаре не оказалось.

# Соединение нескольких уровней ОП

Вы можете соединить несколько уровней опциональных последовательностей вместе для того, чтобы пробраться до свойств, методов, сабскриптов, которые находятся глубже в модели. Однако многоуровневые опциональные последовательности не добавляют новых уровней опциональности к возвращаемым значениям:

Скажем другими словами:

* Если тип, который вы пытаетесь получить не опциональный, то он станет опциональным из-за опциональной последовательности.
* Если тип, который вы пытаетесь получить, уже опциональный, то более опциональным он уже не станет, даже по причине опциональной последовательности.

Таким образом:

* Если вы пытаетесь получить значение типа Int через опциональную последовательность, то получите Int?, и это не будет зависеть от того, сколько уровней в опциональной последовательности задействовано.
* Аналогично, если вы попытаетесь получить значение типа Int? через опциональную последовательность, то вы получите Int?, что опять таки не зависит от количества уровней, которые задействованы в опциональной последовательности.

Пример ниже пробует получить доступ к свойству street свойства address свойства residence экземпляра john. Здесь задействовано два уровня опциональной последовательности для того, чтобы соединить свойства residence и address, оба из которых опционального типа:

ifletjohnsStreet = john.residence?.address?.street {

print("John's street name is \(johnsStreet).")

} else {

print("Unable to retrieve the address.")

}

// Выведет "Unable to retrieve the address."

Значение john.residence на данный момент содержит корректный экземпляр класса Residence. Однако значение john.residence.address равно nil. Из-за этого вызов john.residence?.address?.street проваливается.

Обратите внимание, что в примере выше вы пытаетесь получить значение свойства street. Тип этого свойства String?. Возвращаемое значение john.residence?.address?.street так же String?, даже если два уровня опциональной последовательности применены в дополнение к опциональному типу самого свойства.

Если вы установите фактический экземпляр класса Address как значение для john.residence.address и установите фактическое значение для свойства street, то вы можете получить доступ к значению свойства street через многоуровневую опциональную последовательность (цепочку):

letjohnsAddress = Address()

johnsAddress.buildingName = "The Larches"

johnsAddress.street = "Laurel Street"

john.residence?.address = johnsAddress

ifletjohnsStreet = john.residence?.address?.street {

print("John's street name is \(johnsStreet).")

} else {

print("Unable to retrieve the address.")

}

// Выведет "John's street name is Laurel Street."

В этом примере попытка установить свойство address свойства john.residence будет успешной, потому что значение john.residence в настоящее время содержит действующий экземпляр Address.

## Дженерики в Swift

Дженерики (универсальные шаблоны) позволяют вам писать гибкие, многократно используемые функции и типы, которые могут работать с любым типом. Вы можете написать код, который избегает дублирования и выражает свое предназначение в ясной, абстрактной манере.  
  
Типы Array и Dictionary в Swift являются универсальными коллекциями (дженериками).  
В приведенном ниже коде универсальная функция для свапа двух значений используется для строки и целого числа. Это пример кода многократного использования.

funcswapTwoValues<T>(\_ a: inout T, \_ b: inout T) {

lettemporaryA = a

a = b

b = temporaryA

}

var num1 = 4

var num2 = 5

var str1 = “a”

var str2 = “b”

swapTwoValues(&num1,&num2)

swapTwoValues(&str1,&str2)

print (“num1:”, num1) *//output: 5*

print (“num2:”, num2) *//output: 4*

print (“str1:”, str1) *//output: b*

print (“str2:”, str2) *//output: a*

## Когда их следует использовать опциональные типы?

Опциональным (Optional, “опционал”) в Swift является тип, в котором значение может быть, а может и не быть. Опционалы обозначаются путем добавления «?» к любому типу.  
  
*Варианты использования опционала:*

1. Фрагменты кода, которые могут претерпеть неудачу (я чего-то ожидал, но ничего не получил).
2. Объекты, которые в данный момент являются пустыми, но могут стать чем-то позже (и наоборот).

*Хороший пример опционала:*  
  
***Свойство, которое может присутствовать или отсутствовать****, например отчество или муж/жена в классе Person.*  
***Метод, который может возвращать либо значение, либо ничего****, например, поиск соответствия в массиве.*  
***Метод, который может вернуть либо результат, либо получить ошибку и ничего не вернуть****, например, пытаться прочитать содержимое файла (в результате чего обычно будут возвращены данные файла), но файл не существует.*  
***Свойства-делегаты****, которые не всегда должны быть установлены и обычно устанавливаются после инициализации.****Как слабые ссылки в классах****. То, на что они указывают, может быть установлено в nil в любое время.  
Если вам нужен способ узнать, когда установлено значение (данные еще не загружены> данные) вместо использования отдельного логической переменной dataLoaded.*

## Что такое опциональная последовательность (optionalchaining) в Swift?

***Процессы запроса, вызова свойств, сабскриптов и методов для опционала, который может иметь значение «nil», определяется как опциональная последовательность (опциональная цепочка)*.**  
  
Опциональная последовательность возвращает два значения —

* если опционал содержит “значение”, то при вызове связанных с ним свойств, методов и сабскриптов возвращается значение
* если опционал содержит “nil”, все связанные с ним свойства, методы и сабскрипты возвращают nil

Опциональная последовательность — это альтернатива принудительной распаковки.

## Что такое принудительная распаковка (forcedunwrapping)?

**Принудительная распаковка** — это способ **извлечения** значения, содержащегося в опционале. Эта операция опасна, потому что вы, по сути, говорите компилятору: я уверен, что этот опционал содержит реальное значение, извлеки его!

let value: Int? = 1

letnewValue: Int = value! *// ТеперьnewValueсодержит 1*

letanotherOptionalInt: Int? = nil

letanotherInt = anotherOptionalInt! *// Output:fatal error: внезапный nil прираспаковкеопциональногозначения.*

## Что такое неявная распаковка (implicitunwrapping)?

**Неявная распаковка**: когда мы определяем неявно распакованный опционал, мы определяем контейнер, который будет автоматически выполнять принудительную распаковку каждый раз, когда мы его считываем.

varname: String! = “Virat”

letstudent = name*// В данный момент мы считываем текст*

name = nil

letplayer = name*//Output:fatalerror: внезапныйnil при распаковке опционального значения.*

*Если неявно распакованный опционал равен nil и вы пытаетесь получить доступ к его упакованному значению, вы вызовете ошибку времени выполнения. Результат точно такой же, как если бы вы поместили восклицательный знак после обычного опционала, который не содержит значения.*

## Что такое опциональныйбиндинг (Optionalbinding)?

Вы можете распаковывать опционалы как «безопасным», так и «небезопасным» способом. Безопасный способ — использовать опциональный биндинг.  
  
Опциональныйбиндинг используется для выяснения, содержит ли опционал значение, и если да, то мы сделаем это значение доступным в качестве временной константы или переменной. Таким образом нет необходимости использовать суффикс! для доступа к его значению.

letpossibleString: String? = "Hello"

ifletactualString = possibleString {

*//actualString - обычное (не опциональное) строковое значение*

*//равное значению, хранимому в possibleString*

print(actualString)

}

else {

*//possibleString не содержат значения, обработайте эту*

*//ситуацию*

}

Задание для самостоятельного выполнения

1. Создать класс по варианту, сделать сабскрипты для чтения и изменения его полей.
2. Создать новый класс на основе исходного с учетом варианта задания (автопарк, библиотека, каталог, команда, группа, реестр фирм, склад компьютеров, спортклуб, автостоянка, каталог журнальных статей), использовать пример со стр 2.
3. Заполнить информацию и вывести на экран
4. Создать класс ошибок (код, сообщение), возвращаемых сервером. На его основе создайте учет ошибок за текущие сутки, добавьте время в качестве, ключа. Используя сабскрипты, верните ответ сервера в заданном временном интервале.

Варианты

1. Автобус (номер маршрута, название начальной и конечной остановки, количество остановок по маршруту)
2. Книга (название книги, автор, издательство, количество страниц)
3. Файл (имя, время создания, дата создания, размер)
4. Спортсмен (ФИО, вид спорта, сколько лет занимается)
5. Студент (ФИО, группа, средний балл зачетки)
6. Фирма (название, вид услуг, годовой оборот)
7. Компьютер (производитель, частота процессора, количество ядер, объем памяти)
8. Спорт (вид спорта, количество занимающихся в городе, средняя стоимость комплекта инвентаря)
9. Автомобиль (марка, фирма-производитель, год выпуска, пробег)
10. Статья (название, автор, название журнала, год и месяц издания)