Урок 3

Условные инструкции

if

Самым распространенным условным ветвлением кода во всех языках программирования является инструкция `if` и **Swift** в этом случае не исключение. В самой простой форме данная инструкция имеет следующий синтаксис:

```
if condition {
    some code
}
```

После ключевого слова `if`, которое переводится, как «если», необходимо задать условие. Далее после пробела открывается фигурная скобка и на следующей строке выполняется какое-то действие, если условие истинно. Ветвление заканчивается закрывающейся фигурной скобкой на следующей строке после кода.

else if

```
if condition1 {
some code
} else if condition2 {
some code
}
```

Инструкция `else if` используется для проверки следующего условия, в случае если первое условие `if` оказалось ложным. Если условие `else if` истинно, то выполняется код помещенный внутри фигурных скобок.

Как только компилятор обнаружит условие, которое является истинным, проверка последующих условий прекращается и код в них не будет выполнен, даже если эти условия истины.

else

```
if condition1 {
some code
} else if condition2 {
some code
} else {
    some code
}
```

Данное ветвление содержит финальную инструкцию `else`, которая не содержит в себе ни каких условий и будет выполнена только в том случае, если все предыдущие условия были ложными.

Тернарный оператор

Тернарный условный оператор — это оператор который состоит из трех частей. Синтаксис такого оператора имеет следующий вид: выражение `?` действие`1` `:` действие`2`

Switch statement

Обычно мы используем инструкцию `if`, если наше условие достаточно про- стое и предусматривает всего несколько вариантов. А вот инструкция `switch` подходит для более сложных условий, с множественными перестановками, и

очень полезна в ситуациях, где по найденному совпадению с условием выбирается соответствующая ветка кода для исполнения.

Инструкция `switch` подразумевает наличие какого-то значения, которое сравнивается с несколькими возможными шаблонами. После того как значение совпало с каким-либо шаблоном, выполняется код, соответствующий ответвлению этого шаблона, и больше сравнения уже не происходит. `switch` представляет собой альтернативу инструкции `if`, отвечающей нескольким потенциальным значениям.

В самой простой форме в инструкции `switch` значение сравнивается с одним или более значений того же типа:

```
switch значение для сопоставления {
 case значение 1:
  инструкция для значения 1
 case значение 2, значение 3:
  инструкция для значения 2 или значения 3
 default:
  инструкция,  если совпадений с шаблонами не найдено
}
```

После ключевого слова `switch` вписываем значение для сопоставления. Это значение в отличии от инструкции `if` не должно возвращать булево значение. Т.е. тут мы указываем конкретное значение…

Далее открываем тело ветвления при помощи фигурной скобки и прописываем каждый возможный случай в кейсах, каждый из которых начинается с ключевого слова `case`

Каждый отдельный блок `case` в `switch` — это отдельная ветка исполнительного кода. Инструкция switch определяет какое ветвление должно быть выбрано

После того, как перечислены возможные варианты, последним блоком кода идет условие, которое срабатывает, если не один из кейсов выше не со-

ответствует значению для сопоставления. Такой случай по умолчанию называется `default`, и он всегда идет после всех остальных случаев.

По умолчанию после выполнения кейса в switch управление переходит на инструкции после switch, чтобы провалиться в следующий кейс нужно написать в конце case ключевое слово **fallthrough**.

Перечисления

Описание перечисления

Перечисление — это набор значений определенного типа данных, позволяющий взаимодействовать с этими значениями.

Перечисление — это очень мощная концепция, позволяющая избежать минорных ошибок при разработке приложений, она делает код чище и безопаснее. По своей сути перечисление позволяют вам создавать списки возможных значений, который в дальнейшем могут быть использованы вами в различных ситуациях.

Синтаксис перечислений

```
enum EnumirationName {
    case enumiration
}
```

После ключевого слова **enum** (от слова *enumiration*) с большой буквы пишем имя перечисления. В фигурных скобка после ключевого слова **case** пишем значение.

Также как и при работе с классами, создавая перечисление, вы создаете новый тип данных.

Значения перечислений можно перечислять через запятую, тогда вид перечисления в целом будет иметь более компактный вид:

```
enum Weekday {

    case monday, tuesday, wednesday,
    thursday, friday, saturday, sunday

}
```

В этом примере значения внутри перечисления имеют тип **Weekday**. Доступ к значениям перечисления происходит через точку. Инициализацию значения можно выполнить двумя способами Можно присвоить переменной тип данных, в качестве которого выступает имя перечисления.

```
var weekday: Weekday
weekday = Weekday.thursday
```

Так же можно сразу присвоить конкретный кейс из перечисления.

```
var weekday = Weekday.thursday
```

В том случае, если тип переменной соответствует перечислению (выше мы присвоили переменной тип **Weekday**), то присваивая этой переменной какой либо значения из перечисления, можно опустить само название перечисления и просто выбрать нужный кейс поставив после знака присваивания точку.

```
weekday = .monday
```

Возможности перечислений

```
switch weekday {
    case .monday:
        print("To set the alarm for 8 am")
    case .tuesday:
        print("To set the alarm for 8:30 am")
    case .wednesday:
        print("To set the alarm for 8:30 am")
    case .thursday:
        print("To set the alarm for 7:30 am")
    case .friday:
        print("Yay! The last day of the week!")
    default:
        print("Weekend. Alarm not set")
    }
```

Исходные значения

Перечислениям может быть присвоен определенный тип данных (**String**, **Character**, **Int**). В этом случае членам перечисления можно присвоить уникальные исходные значения определенного типа, при этом все члены перечисле- ния будут иметь один общий тип данных:

```
enum Weekday: String {

    case monday = "To set the alarm for 8 am"
    case tuesday = "To set the alarm for 8:30 am"
    case wednesday = "To set the alarm for 9 am"
    case thursday = "To set the alarm for 7:30 am"
    case friday = "Yay! The last day of the week!"
    case saturday
    case sunday
}

var weekday = Weekday.monday
```

Для доступа к исходному значению члена перечисления существует специальное свойство:

```
print(weekday.rawValue)
```

Когда вы работаете с перечислениями, которые хранят целочисленные или строковые исходные значения, можно не присваивать исходные значения явно для каждого конкретного кейса. Swift автоматически сделает это за вас.

```
weekday = .saturday
print(weekday.rawValue)
```

Weekday.saturday имеет неявное текстовое значение **saturday**. При этом если в качестве исходных значений используются целые числа, неявное значение для каждого кейса будет на единицу больше, чем в предыдущем кейсе. Если первый кейс не имеет заданного значения, его значение равно 0.

```
enum Planet: Int {
    case mercury = 1, venus, earth, mars, jupiter, saturn,
uranus, neptune
}

var somePlanet = Planet.earth
print("Earth is the \(somePlanet.rawValue) planet from the sun")
```

Если объявить перечисление вместе с типом исходного значения, то перечисление автоматически получает инициализатор, который берет значение типа исходного члена перечисления (как параметр **rawValue**) и возвращает либо член перечисления либо **nil**. Вы можете использовать этот инициализатор, чтобы попытаться создать новый экземпляр перечисления.

В этом примере **Uranus** инициализируется через его исходное значение **7**:

Инициализация

```
let possiblePlanet = Planet(rawValue: 7)
print("The seventh planet is \(possiblePlanet!)")
```

Типы данных так же можно присваивать каждому члену перечисления по отдельности. Члены перечисления могут хранить связанные значения любого необходимого типа. При этом набор связанных значений для каждого члена перечисления может быть произвольным.

Связанные значения (ассоциированные параметры)

```
enum Weekday {

    case workday(String, Int)
    case weekend(String)
}
```

Члены перечисления **Weekday** не имеют ни каких значений типа **Int** или **String**.

Они лишь определяют типы связанных значений, которые переменные с типом **Weekday** могут содержать.

```
var weekday = Weekday.workday("Set alarm to", 8)

switch weekday {
    case .workday(let message, let time):
        print("\(message), \(time)")
    case .weekend(let weekendMessage):
        print("\(weekendMessage)")
    }
```

Если все связанные значения для членов перечисления извлекаются как константы или переменные, то для краткости вы можете разместить одиночное let или var перед именем члена:

```
switch weekday {
    case let .workday(message, time):
        print("\(message), \(time)")
    case .weekend(let weekendMessage):
        print("\(weekendMessage)")
    }
```

Опциональные типы

Опциональные типы данных или просто **опционалы** — это еще одна особенность языка Swift. Это такой особый тип данных, который говорит о том, что переменная или константа либо имеет значение определенного типа, либо **nil**.

Самый простой пример: вам необходимо использовать в своем приложении какие то данные полученные из сети. Для этого вы их сначала получаете, а потом используете по назначению. Но по ряду различных причин эти данные не всегда могут быть получены, или данные могут иметь не тот тип, который мы ожидаем. Если мы обратимся к этим данным, ожидая получить от них значения определенного типа, то программа упадет с ошибкой в рантайме. Что бы приложение не падало по независимым от нас причинам, необходимо использовать опциональные типы. В этом случае, обратившись к опциональной переменно, мы либо получим значение с нужным нам типом, либо **nil**.

Опциональные типы имеют в конце знак вопроса, например `Int?`. Тип `Int` и `Int?` — это не одно и то же. Это два разных типа. Тип `Int?` это опциональный интерджер, который основан на типе `Int`. Знак вопроса в конце типа означа- ет, что это опциональный тип и он может принимать значения, как основного типа (в нашем случае целочисленное число), так и `nil`. `nil` это специальное значение, которое говорит, что объект не имеет ни какого значения.

Опциональную переменную можно установить в состояния отсутствия значения, присвоив ей `nil`. `nil` можно присвоить только опциональным объек- там. Если вы считаете, что значение переменной или константы может отсутствовать, всегда присваивайте таким объектам опциональные типы.

Опциональные переменные, если им не присвоить ни какого значения автоматически устанавливаются в **nil**

Извлечение опциональных типов

Когда вы работаете с объектами опционального типа, вы можете не знать содержит ли объект значение или нет. Поэтому прежде чем обратится к такому объекту надо проверить, есть ли в нем значение. Для этих целей можно использовать инструкцию `if` сравнивая опционал с `nil`.

```
if someOptionalValue == nil {
    print("someOptionalValue does not contains some value")
}
```

Если условие проверки на содержание значения в переменной истинно, то мы можем получить доступ к его значению, добавив восклицательный знак (!) в конце имени опционала. Восклицательный знак фактически говорит: «Я знаю точно, что этот опционал содержит значение, пожалуйста, используй его». Это выражение известно как Принудительное извлечение значения опционала:

```
if someOptionalValue != nil {
    print("someOptionalValue has some value of \
(someOptionalValue!).")
}
```

Если применить принудительное извлечение **!** к несуществующему опциональному значению — это вызовет рантайм ошибку. Поэтому всегда прежде чем его использовать, необходимо выполнить проверку на содержание значения.

Привязка опционалов

Что бы постоянно не использовать восклицательный знак при обращении к опционалу, можно из опционала извлечь значение и присвоить его временной

переменной или константе, а далее работать с этой переменной. Такой способ называется *Привязкой опционалов*. Прежде чем присвоить значение опционалу переменной, необходимо проверить его на содержание этого значения.

```
if let currentValue = someOptionalValue {
    print(currentValue)
} else {
    print("Error")
}
```

При извлечении функционала таким способом, мы присваиваем значение это- го функционала другому объекту и в дальнейшем работаем уже с новым объектом, который имеет явный тип. Поэтому использовать **!** тут не нужно.

Неявно извлеченные опционалы

Иногда, сразу понятно из структуры программы, что опционал всегда будет иметь значение, после того как это значение впервые было установлено. В этих случаях, очень полезно избавиться от проверки и извлечения значения опционала каждый раз при обращении к нему, потому что можно с уверенностью утверждать, что он постоянно имеет значение.

Эти виды опционалов называются неявно извлеченные опционалы. Их можно писать, используя восклицательный знак (`String!`), вместо вопросительного знака (`String?`), после типа, который вы хотите сделать опциональным.

Неявно извлеченные опционалы полезны, когда известно, что значение опционала существует непосредственно после первого объявления опционала, и точно будет существовать после этого.

Вместо размещения восклицательного знака после имени опционала каждый раз, когда вы его используете, ставьте восклицательный знак после типа опционала вовремя его объявления.

Не используйте неявно извлечённый опционал, если существует вероятность, что в будущем переменная может стать `nil`. Всегда используйте нормальный тип опционала, если вам нужно проверять на `nil` значение в течение срока службы переменной.