Новые подходы к работе с регулярными выражениями в **Swift.**

Просто о сложном.

Mobius, 2023. Газпромбанк
Спикер: Мирусин Илья (tg: @ilmsn)
Эксперт: Кочетков Андрей (tg: @andrey_ko4etkov)

Что обсудим

01 Тренды **Apple**

Что привело к изменению методов работы с регулярными выражениями

02 **NSRegularExpression**

Достоинства и недостатки.
Зачем менять устоявшийся подход?

03 **Regex, RegexBuilder**

Что меняется с появлением Regex, RegexBuilder: стало лучше?

04 Решение кейсов

Как одни и те же задачи решаются NSRegularExpression и RegexBuilder

05 Итоги

01. Что привело к изменению методов работы с регулярными выражениями

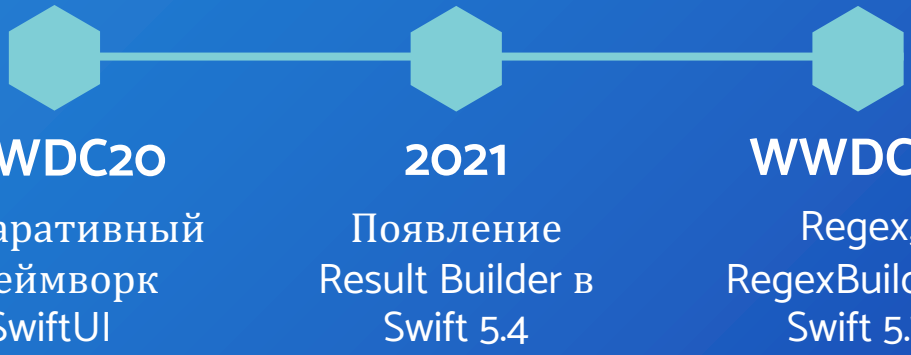
Как работали с **регулярными выражениями?**

- Метод NSString . range(of:options:range:locale:) или replacingOccurrences(of:with:options:) с опцией .regularExpression
  - ⚠ *имеет ограничения (нельзя использовать более 1 раза или в capture groups)*
- Строковые методы .split() + hardcode + .join()
  - ⚠ *мешает hardcode и отсутствие масштабируемости/гибкости*
- Класс **NSRegularExpression** на базе ICU regular expression
  - ⚠ *ICU работает с моделью строк, отличной от Swift, что **не позволяет** использовать преимущества строк Swift*
- Оба подхода используют сложный для чтения, записи и использования синтаксиса regex, вроде этого:

```
let pattern = #"^\S+@\S+\.\S+$"# // валидация e-mail
```

01. Что привело к изменению методов работы с регулярными выражениями

А, между тем, **как** развивается язык **Swift?**



02. Достоинства и недостатки **NSRegularExpression.** Зачем менять?

Почему мы привыкли к **NSRegularExpression** и регулярным выражениям?

- Кратко, точно и лаконично:

  \(\d{3}\) \d{3}-\d{4}, где:

  \(соответствует «(«
  \d есть однозначное число (0..9)
  {3} количество повторений
  \) означает «)»
  пробел
  - есть тире

... для извлечения номера телефона из строки
«Звоните (926) 082-1234 или (952) 123-4567»

02. Достоинства и недостатки NSRegularExpression. Зачем менять?

Но иногда бывает чуточку **сложнее**:

```
static let extnPattern = "(?:;ext=([0-90-9१-९]{1,7})|[
    \\t,]*(?:e?xt(?:ensi(?:ó?|ó))?n?|e?x t n?|[,xxX##~〜;]|int|anexo|i n t)[:\\.. ]?[
    \\t,-]*([0-90-9१-९]{1,7})#?|[- ]+([0-90-9१-९]{1,5})#)$"
```

... чтобы извлечь добавочный номер из строки «+7 612-345-678 ext.89»

02. Достоинства и недостатки NSRegularExpression. Зачем менять?

И все же. Почему мы **привыкли** к **NSRegularExpression?**

- Универсальный синтаксис регулярных выражений
- Обладает свойствами неизменяемости (immutable) и потокобезопасности (thread safe)
- Имеет throwing initializer, если regex pattern невалидный (проверяется в runtime)

02. Достоинства и недостатки NSRegularExpression. Зачем менять?

Пример использования **NSRegularExpression**

```
1  import Foundation
2
3  let testString = "contact me: ivan@ivanov.com"
4  let expression = "^[A-Za-zA-Яа-я0-9_
       \\!\\#\\$\\%\\&\\'\\*\\+\\-\\/\\=\\?\\^\\_\\{\\}\\|\\~\\@\\.\\(\\)\\:\\₽]*$"
5
6  do {
7      let regex = try NSRegularExpression(pattern: expression,
8                                          options: [.caseInsensitive])
9      let numberOfMatches = regex.numberOfMatches(in: testString,
10                                                 options: [],
11                                                 range: NSMakeRange(0, testString.count))
12     print("Number of matches: ", numberOfMatches)
13 } catch {
14     // Catching
15 }
```

02. Достоинства и недостатки NSRegularExpression. Зачем менять?

Теперь о **недостатках** NSRegularExpression

После создания экземпляра NSRegularExpression требуется обработка ошибок (error handling), которая не проверяется в compile time

Преобразует String в NSString (в дальнейшем используя NSRange, NSMutableString и т.д.), что не нативно для Swift (пережиток Objective-C)

Отсутствуют многие алгоритмы (разделение с помощью regex, как разделителя; удаление совпадений regex в строке; lazy matching, сопоставление n раз)

Алгоритмы NSRegularExpression не являются generic для String Protocol, что усложняет работу с substring

Страдает скорость обработки больших объемов данных

Проверка валидности паттерна только в runtime

02. Достоинства и недостатки NSRegularExpression. Зачем менять?

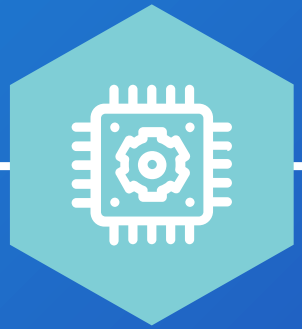
А **как менять** устоявшийся подход?

Regex Literals

Проверяемые в compile time

Путь Generic

Regex<Element>, Regex<Collection> or <R: Regex> where R.Element == Character



DSL Framework

Используя Result Builders по аналогии со SwiftUI

Extension к StringProtocol

Добавляя недостающие методы* работы с regex

* Напр., https://github.com/johnno1962/SwiftRegex5/blob/main/SwiftRegex5.playground/Sources/TupleRegex.swift#L114

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

Тип **Regex<Output>**

Regex Literals

RegexBuilder

Swift 5.7

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

С чего начались изменения в работе с регулярными выражениями?

В **Swift 5.7** реализованы новые типы (де-факто алгоритмы обработки строк)

- **Regex<Output>**
- **Regex<Output>.Match**

Если **Output** тип не определен используется **Regex<AnyRegexOutput>**

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

Особенности типа Regex<Output>:

- Регулярные выражения создаются в runtime
- Синтаксис схож с созданием регулярных выражений в compile time
- Output является Generic типом
- Так как создается динамически подходит для проверки user input

03. Что меняется с появлением Regex, RegexBuilder: стало лучше?

Пример использования Regex<Output>:

```swift
import Foundation

let passPattern =
    // 8 или более символов
    #"(?=.{8,})"# +
    // Как минимум 1 цифра
    #"(?=.*\d)"#

let regex = try! Regex(passPattern)

let inputExpression = "jUhdjudf2"

if let result = inputExpression.firstMatch(of: regex) {
    print("Password matches the patterns. And result is: ", result)
} else {
    print("Password not matches the pattern.")
}
```

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

Что под капотом типа Regex<Output>

- Строковые методы **prefixMatch(in:), firstMatch(in:), wholeMatch(in:),** возвращающие Regex<Output>.Match?
- Реализован доступ к результату выборки через subscript (напр. result.0, result.1 и т.д.)
- Методы:
  ignoresCase (игнор регистра),
  asciiOnlyDigits (только цифры ASCII),
  dotMatchesNewlines (начало и конец выражения совпадают с новой строкой),
  repetitionBehavior (квантификация),
  wordBoundaryKind (границы слова) и другие.

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

Недостатки типа Regex<Output>:

- Требуется экранировать используемый паттерн регулярного выражения
- Нет валидации синтаксиса в compile time, узнаем об ошибке только в runtime, если try выбрасывает ошибку

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

После были реализованы Regex Literals со своими особенностями:

- Проверяется в compile time
- Подсвечивание синтаксиса, не требуется error handling (try)
- Capture type известен в compile time и представляет собой Substring

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

... продолжение особенностей Regex Literals

- Можно присваивать имена переменным с выборки

```
let regex = /(?<identifier>[[:alpha:]]\w*) = (?<hex>[0-9A-F]+)/

if let match = inputExpression.wholeMatch(of: regex) {
    print(match.identifier, match.hex)
}
```

- Результат выборки формирует кортеж (tuple) из Substring:

```
if let match = inputExpression.firstMatch(of: regexUsername) {
    let (matched, name, lastName) = match.output
    print(matched, name, lastName)
}
```

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

Пример использования Regex Literals:

```
// 1–2 цифры, за которыми следует – или /
// плюс 1–2 цифры, за которым следует – или /
// в конце 4 цифры
let regexDate = /(?<date>\d{1,2}[\/-]\d{1,2}[\/-]\d{4})/

let inputExpression = "10/05/2023"

if let match = inputExpression.wholeMatch(of: regexDate) {
    print("Captured name: ", match.date)
}
```

```
Captured name:  10/05/2023
```

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

А затем появился RegexBuilder:

- Использующий Domain-Specific Language (DSL)

- В основе которого механизм **Result Builder** (реализован в Swift 5.4), позволяющий «собирать» результат выполнения функции из последовательности компонентов, например вот так:

```swift
import RegexBuilder

let regex = Regex {
    OneOrMore(.digit)
    "@"
    OneOrMore(.word)
}
```

- Преследующий принципы выразительности, понятности, безопасности типов и удобства использования.

03. Что меняется с появлением Regex, RegexBuilder: стало лучше?

Реализуя RegexBuilder, мы отступаем от классического подхода к regex, когда регулярные выражения:

- Просто отвечали на вопрос «да/нет», при этом механизм выполнения значения не имел
- Не делали различий между лексическим анализом и парсингом., то есть не затрагивали вопросы обработки и устранения ошибок, дебага, и т.д., что позволяют современные парсеры

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

На WWDC22 механизм RegexBuilder характеризовали так:

Использование RegexBuilder приводит к понятным литералам, сами билдеры обеспечивают структурирование конструкций regex

Можно использовать уже готовые парсеры, имеющиеся в Foundation, как часть выражения regex (напр., date formatter)

Имеется поддержка Unicode

Предсказуемое исполнение, контролируемое управление

Проверка и валидация в compile time

Простота, понятность и экономия времени

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

Можно смело сказать, что RegexBuilder решает следующие проблемы текстовой записи regex:

- Символы синтаксиса \, (, [, { расположены в одну строку и сложны к прочтению
- Меньшие паттерны (subpattern) встраиваются в общий паттерн выражения
- Отсутствует code completion при составлении выражения
- Capturing group дают необработанные данные (range/substring) и преобразовываются только после
- Наличие inline комментариев вида (?#...) затрудняет читаемость

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

… и решает эти проблемы regex следующим образом:

БЫЛО	СТАЛО
Символы синтаксиса \, (, [, { расположены в одну строку и сложны к прочтению	Capture groups и квантификаторы читаемы и выглядят как API calls
Меньшие паттерны (subpattern) встраиваются в общий паттерн выражения	Scoping + отступы
Отсутствует code completion при составлении выражения	Есть code completion
Capturing group дают необработанные данные (range/substring) и преобразовываются только после	Преобразовываются в точке вызова
Наличие inline комментариев затрудняет читаемость	Обычные комментарии

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

RegexBuilder предлагает следующие инструменты под капотом:

01. Captures

Capture
Сохраняет substring (или преобразованные данные) после matching с регулярным выражением

TryCapture
Выполняет преобразование (в замыкании transform) над сохраненной substring; в случае неуспеха - возвращает результат без transform. Strongly typed capture!

Reference
Представляет собой создаваемую до скоупа Regex ссылку на регулярное выражение (Capture/TryCapture)

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

Capture. **Пример** работы

```
1  import RegexBuilder
2
3  let regex = Regex {
4    Capture {
5        OneOrMore(.whitespace.inverted)
6    }
7    "@"
8    Capture {
9        OneOrMore(.whitespace.inverted)
10       "."
11       OneOrMore(.whitespace.inverted)
12   }
13 }
14
15 let inputExpression = "User with the email ivanov@google.com has logged into system."
16
17 if let matchValue = inputExpression.firstMatch(of: regex) {
18   let username = matchValue.1 // ivanov
19   let emailServer = matchValue.2 // google.com
20 }
```

03. Что меняется с появлением Regex, RegexBuilder: стало лучше?

TryCapture. Пример работы

```
import RegexBuilder

let regex = Regex {
  TryCapture {
      OneOrMore(.whitespace.inverted)
  } transform: { String($0) }
  "@"
  Capture {
      OneOrMore(.whitespace.inverted)
      "."
      OneOrMore(.whitespace.inverted)
  }
}

let inputExpression = "User with the email ivanov@google.com has logged into system."

if let matchValue = inputExpression.firstMatch(of: regex) {
    let userName = matchValue.1
    print(type(of: userName)) // String
    let emailServer = matchValue.2
    print(type(of: emailServer)) // Substring
}
```

03. Что меняется с появлением Regex, RegexBuilder: стало лучше?

Reference. Пример работы

```swift
import RegexBuilder

let username = Reference(Substring.self) // Инициализирована ссылка username
let emailServer = Reference(Substring.self) // Инициализирована ссылка emailServer

let regex = Regex {
  // Используем Reference username
  Capture(as: username) {
      OneOrMore(.whitespace.inverted)
  }
  "@"
  // Используем Reference emailServer
  Capture(as: emailServer) {
      OneOrMore(.whitespace.inverted)
      "."
      OneOrMore(.whitespace.inverted)
  }
}

let inputExpression = "User with the email ivanov@google.com has logged into system."

if let matchValue = inputExpression.firstMatch(of: regex) {
    print("Username:", matchValue[username]) // Username: ivanov
    print("EmailServer:", matchValue[emailServer]) // EmailServer: google.com
}
```

03. Что меняется с появлением Regex, RegexBuilder: стало лучше?

RegexBuilder предлагает следующие инструменты под капотом:

02. Quantifiers

- **One:** единожды
- **Optionally:** 0 или 1 раз
- **ZeroOrMore:** 0 или больше раз
- **OneOrMore:** 1 или больше раз
- **Repeat:** указанное n раз
- **Local:** создает **atomic group** с отказом от **backtracking** в случае успеха

03. Что меняется с появлением **Regex, RegexBuilder:** стало лучше?

Quantifiers. Local. **Пример** работы

```
1  import RegexBuilder
2
3  let regex = Regex {
4    TryCapture {
5        OneOrMore(.whitespace.inverted)
6    } transform: { String($0) }
7    "@"
8    Capture {
9        OneOrMore(.whitespace.inverted)
10   }
11   "."
12   Capture {
13     Local {
14         ChoiceOf {
15             "net.org"
16             "net"
17         }
18     }
19     ".org"
20   }
21 }
22
23 let inputExpression = "User with the email ivanov@company.net.org has logged into system." // Not matching.
24
25 if let matchValue = inputExpression.firstMatch(of: regex) { // nil
26     let userName = matchValue.1
27     let emailServer = matchValue.2
28     let emailDomain = matchValue.3
29 }
```

RegexEngine обладает механизмом **backtracking** «из коробки»

В случае с использованием **Local, backtracking** прекращается, когда мы получаем **matching** результат

03. Что меняется с появлением Regex, RegexBuilder: стало лучше?

RegexBuilder предлагает следующие инструменты под капотом:

03. Components

- **CharacterClass**
  Матчинг с отдельным символом или группой символов
- **Anchor**
  Поиск совпадений на определенной позиции
- **Lookahead**
  Опережающая проверка
- **NegativeLookahead**
  Негативная опережающая проверка (ретро)
- **ChoiceOf**
  Выбирает один из составляющих его компонентов

03. Что меняется с появлением Regex, RegexBuilder: стало лучше?

Lookahead. Пример работы

```
1  import RegexBuilder
2
3  let regex = Regex {
4      Capture {
5          OneOrMore(.digit)
6          ","
7          OneOrMore(.digit)
8          Lookahead {
9              "RUB"
10         }
11     }
12 }
13
14 let inputExpression = "MasterCard **0001 покупка Перекресток 140,32RUB"
15
16 if let matchValue = inputExpression.firstMatch(of: regex) {
17     let purchaseSum = matchValue.1 // 140,32
18 }
```

03. Что меняется с появлением Regex, RegexBuilder: стало лучше?

RegexBuilder предлагает следующие инструменты под капотом:

04. Builders

RegexComponentBuilder

Позволяет связывать несколько регулярных выражений в одно (механизм Concatenation)

AlternationBuilder

Обеспечивает последовательный перебор каждого из нескольких регулярных выражений в своем составе. Alternation может быть создана использованием компонента ChoiceOf

03. Что меняется с появлением Regex, RegexBuilder: стало лучше?

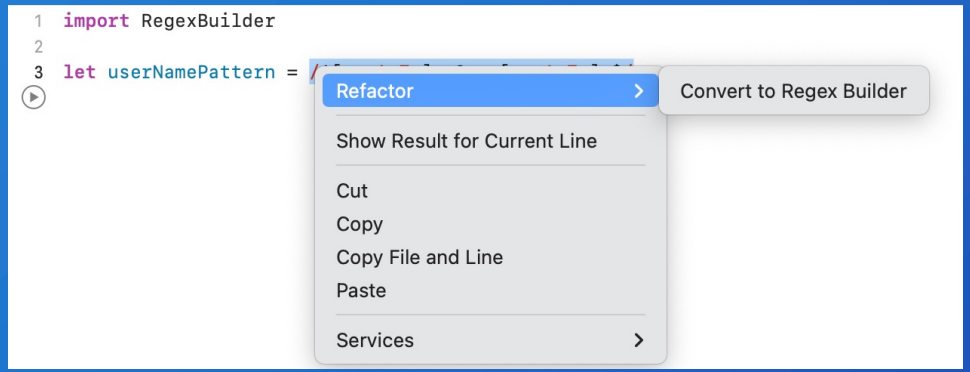
Минутка комфорта.

А что, если можно сделать рефакторинг regex в два клика?

01

```
1  import RegexBuilder
2
3  let userNamePattern = /^[a-zA-Z-]+ ?.* [a-zA-Z-]+$/
```

02



03. Что меняется с появлением Regex, RegexBuilder: стало лучше?

А что, если можно сделать рефакторинг regex в два клика?

03

```
import RegexBuilder

let userNamePattern = Regex {
    Anchor.startOfLine
    OneOrMore {
        CharacterClass(
            .anyOf("-"),
            ("a"..."z"),
            ("A"..."Z")
        )
    }
    Optionally {
        " "
    }
    ZeroOrMore(.any)
    " "
    OneOrMore {
        CharacterClass(
            .anyOf("-"),
            ("a"..."z"),
            ("A"..."Z")
        )
    }
    Anchor.endOfLine
}
```

03. Что меняется с появлением Regex, RegexBuilder: стало лучше?

Тем не менее, у RegexBuilder есть и противники такого подхода, которые считают, что:

- Фреймворк усложняет работу с регулярными выражениями, особенно для тех, кто знаком и давно работает с «чистыми» регулярными выражениями (проще написать регулярное выражение в одну строку)
- Он не является универсальным для других языков, «заточен» только под Swift
- Развернутый синтаксис регулярного выражения занимает значительное число строк на экране и не улучшает, а ухудшает читаемость

04. Как одни и те же задачи решаются NSRegularExpression и RegexBuilder

A.

Задача: валидировать вводимый пользователем пароль.
NSRegularExpression

```swift
import Foundation

func validatePassword(input password: String) -> Bool {
    // At least 8 characters, one capital letter, one lowercase letter, one digit, one special character
    let passwordPattern = #"(?=.{8,})(?=.*[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[ !$%&?._-])"#

    let passwordRegex = try! NSRegularExpression(
        pattern: passwordPattern,
        options: []
    )

    let result = passwordRegex.matches(
        in: password,
        options: [],
        range: NSRange(
            password.startIndex..<password.endIndex,
            in: password
        )
    )

    return !result.isEmpty
}

validatePassword(input: "z") // false. not validated
validatePassword(input: "1234567Gg$") // true. validated
```

04. Как одни и те же задачи решаются NSRegularExpression и RegexBuilder

A.

Задача: валидировать вводимый пользователем пароль.

RegexBuilder

```swift
import RegexBuilder

func validatePassword(input password: String) -> Bool {
    // At least 8 characters, one capital letter, one lowercase letter, one digit, one special character
    let passwordPattern = Regex {
        Lookahead {
            Repeat(8...) {.any}
        }
        Lookahead {
            Regex {
                ZeroOrMore(.any)
                ("A"..."Z")
            }
        }
        Lookahead {
            Regex {
                ZeroOrMore(.any)
                ("a"..."z")
            }
        }
        Lookahead {
            Regex {
                ZeroOrMore(.any)
                One(.digit)
            }
        }
        Lookahead {
            Regex {
                ZeroOrMore(.any)
                One(.anyOf("!$%&?._-"))
            }
        }
    }

    guard let _ = password.firstMatch(of: passwordPattern) else {
        return false
    }

    return true
}
validatePassword(input: "z") // false. not validated
validatePassword(input: "1234567Gg$") // true. validated
validatePassword(input: "AbCdBdFt") // false. not validated
validatePassword(input: "$AOOOOO1") // false. not validated
```

04. Как одни и те же задачи решаются NSRegularExpression и RegexBuilder

B.

Задача: валидация **url** в приложении (например, если **url** приходит с **backend**).

NSPredicate

```
import Foundation
import RegexBuilder

extension URL {

    var isValid: Bool {

        let regEx = "((https|http)://)([(w|W)]{3}+\\.)?+(.)+([.]|[/])+(.)+"
        let predicate = NSPredicate(format: "SELF MATCHES %@", argumentArray: [regEx])

        return predicate.evaluate(with: self.absoluteString)
    }

}

let ftpUrl = URL(string: "ftp://google.com")!
let httpsUrlWithoutDomain = URL(string: "https://www.google/")!
let ordinaryUrl = URL(string: "https://google.com")!

ftpUrl.isValid // false
httpsUrlWithoutDomain.isValid // false
ordinaryUrl.isValid // true
```

04. Как одни и те же задачи решаются NSRegularExpression и RegexBuilder

B.

Задача: валидация **url** в приложении (например, если **url** приходит с **backend**).
RegexBuilder.

Способ ПЕРВЫЙ.
Простой.

```
import Foundation
import RegexBuilder

extension URL {

    var isValid: Bool {
        let emailRegEx = Regex {
            Capture(.url())
        }

        if let _ = self.absoluteString.firstMatch(of: emailRegEx) {
            return true
        }

        return false
    }
}

let ftpUrl = URL(string: "ftp://google.com")!
let notUrl = URL(string: "google.com")!

ftpUrl.isValid // true
notUrl.isValid // false
```

04. Как одни и те же задачи решаются NSRegularExpression и RegexBuilder

B.

Задача: валидация **url** в приложении (напр. если приходит с **backend**).
RegexBuilder.

Способ **ВТОРОЙ.**
Правильный.

```
let ftpUrl = URL(string: "ftp://google.com")!
let httpsWithoutColon = URL(string: "https//google.net.org/")!
let ordinaryHttpsUrl = URL(string: "https://www.google.com/?searchId=123")!
let ordinaryHttpUrl = URL(string: "http://yahoo.com/")!

ftpUrl.isValid // false
httpsWithoutColon.isValid // false
ordinaryHttpsUrl.isValid // true
ordinaryHttpUrl.isValid // true
```

```
import Foundation
import RegexBuilder

extension URL {

    var isValid: Bool {

        let regEx = Regex {
            ChoiceOf {
                "http"
                "https"
            }
            "://"
            Optionally {
                "\\w"
                Repeat(count: 3) {
                  One(.anyOf("(w|W)"))
                }
            }
            OneOrMore(.any)
            OneOrMore {
                ChoiceOf {
                  One(.anyOf("."))
                  One(.anyOf("/"))
                }
            }
            OneOrMore(.any)
        }

        if let _ = self.absoluteString.firstMatch(of: regEx) {
            return true
        }

        return false
    }

}
```

04. Как одни и те же задачи решаются NSRegularExpression и RegexBuilder

C.

Задача: валидация текста **push** сообщения для получения кода его группы.
NSRegularExpression

1. Получаем код группы и регулярное выражение для отнесения текста **push** к нему

2. Для всех списаний с банковской карты используется следующее **regex:**

#"^*[0-9]{4}\s(Оплата|Перевод|Снятие)"#

3. Далее ищем соответствия с помощью метода **isMessage(message:pattern)**

```
private func isMessage(_ message: String, matchesFilterExpression pattern: String) -> Bool {
    do {
        let filterExpression = try NSRegularExpression(pattern: pattern, options: [.caseInsensitive])
        let numberOfMatches = filterExpression.numberOfMatches(
            in: message,
            options: [],
            range: NSMakeRange(0, message.count)
        )

        return numberOfMatches > 0
    } catch {
        return false
    }
}
```

04. Как одни и те же задачи решаются NSRegularExpression и RegexBuilder

C.

Задача: валидация текста **push** сообщения для получения кода его группы.
С использованием **Regex Literals**

```swift
import Foundation
import RegexBuilder

private func isMessage(_ message: String, matchesFilterExpression pattern:
    Regex<(Substring, Substring)>) -> Bool {

    if let _ = message.firstMatch(of: pattern) {
        return true
    }

    return false
}

let regularExpression = /^\*[0-9]{4}\s(Оплата|Перевод|Снятие)/

isMessage("*0001 Перевод 1000р SBP C2C SPISANIE Доступно 500р",
          matchesFilterExpression: regularExpression) // true
isMessage("*9992 Снятие 200р GAZPROMBANK Доступно 0р",
          matchesFilterExpression: regularExpression) // true
isMessage("*0001 Зачисление зарплаты 10р Доступно 12р",
          matchesFilterExpression: regularExpression) // false
```

04. Как одни и те же задачи решаются NSRegularExpression и RegexBuilder

C.

Задача: валидация текста **push** сообщения для получения кода его группы.
С использованием **RegexBuilder**

```swift
import Foundation
import RegexBuilder

private func isMessageHasDebitCode(_ message: String) -> Bool {

    let regEx = Regex {
        /^/
        "*"
        Repeat(count: 4) {
          ("0"..."9")
        }
        One(.whitespace)
        Capture {
          ChoiceOf {
            "Оплата"
            "Перевод"
            "Снятие"
          }
        }
    }

    if let _ = message.firstMatch(of: regEx) {
        return true
    }

    return false
}

isMessageHasDebitCode("*0001 Перевод 1000р SBP C2C SPISANIE Доступно 500р") // true
isMessageHasDebitCode("*9992 Снятие 200р GAZPROMBANK Доступно 0р") // true
isMessageHasDebitCode("*0001 Зачисление зарплаты 10р Доступно 12р") // false
```

04. Как одни и те же задачи решаются NSRegularExpression и RegexBuilder

D.

Задача: валидация маски карты и названия при выполнении перевода.

NSRegularExpression

1. Имеем **regex** для валидации маски и названия карты:

```
#"(?<number>.\d{4})\s(?<title>.+)"#
```

04. Как одни и те же задачи решаются NSRegularExpression и RegexBuilder

D.

Задача: валидация маски карты и названия при выполнении перевода.

NSRegularExpression

2. Далее ищем соответствия с помощью метода **getTitleAndCardNumber(from:with):**

```swift
import Foundation

private func getTitleAndCardNumber(
    from label: String,
    with pattern: String
) -> (title: String, cardNumber: String)? {
    let regex: NSRegularExpression
    do {
        regex = try NSRegularExpression(pattern: pattern, options: [])
    } catch {
        assertionFailure("Некорректное регулярное выражение: \(pattern)")
        return nil
    }
    let range = NSRange(label.startIndex..<label.endIndex, in: label)
    let matches = regex.matches(in: label, options: [], range: range)
    guard let match = matches.first else {
        return nil
    }
    guard let numberRange = Range(match.range(withName: "number"), in: label),
          let titleRange = Range(match.range(withName: "title"), in: label)
    else {
        return nil
    }

    let number = String(label[numberRange]).replacingOccurrences(of: " ", with: "• ")
    let title = String(label[titleRange])

    return (title, number)
}

let regularExpression = #"(?<number>.\d{4})\s(?<title>.+)"#

getTitleAndCardNumber(from: "*4444 название_карты",
                      with: regularExpression) // (title: "название_карты", cardNumber: "*4444")
getTitleAndCardNumber(from: " 2121 Кредитная MasterCard",
                      with: regularExpression) // (title: "Кредитная MasterCard", cardNumber: "• 2121")
getTitleAndCardNumber(from: "0001ДебетоваяUnionPay",
                      with: regularExpression) // nil
```

04. Как одни и те же задачи решаются NSRegularExpression и RegexBuilder

D.

Задача: валидация маски карты и названия при выполнении перевода.
С использованием **RegexLiteral**

```swift
import Foundation
import RegexBuilder

private func getTitleAndCardNumber(
    from label: String,
    with pattern: Regex<(Substring, number: Substring, title: Substring)>
) -> (title: String, cardNumber: String)? {

    guard let match = label.firstMatch(of: pattern) else {
        return nil
    }

    let number = String(match.number).replacingOccurrences(of: " ", with: "• ")
    let title = String(match.title)

    return (title, number)
}

let regularExpression = /(?<number>.\d{4})\s(?<title>.+)/

getTitleAndCardNumber(from: "*4444 название_карты",
                      with: regularExpression) // (title: "название_карты", cardNumber: "*4444")
getTitleAndCardNumber(from: " 2121 Кредитная MasterCard",
                      with: regularExpression) // (title: "Кредитная MasterCard", cardNumber: "• 2121")
getTitleAndCardNumber(from: "0001ДебетоваяUnionPay",
                      with: regularExpression) // nil
```

04. Как одни и те же задачи решаются NSRegularExpression и RegexBuilder

D.

Задача: валидация маски карты и названия при выполнении перевода.
С использованием **RegexBuilder**

```swift
import Foundation
import RegexBuilder

private func getTitleAndCardNumber(from label: String) -> (title: String, cardNumber: String)? {

    let number = Reference(Substring.self)
    let title = Reference(Substring.self)

    let regEx = Regex {
      Capture(as: number) {
        Regex {
          /./
          Repeat(count: 4) {
            One(.digit)
          }
        }
      }
      One(.whitespace)
      Capture(as: title) {
        OneOrMore {
          /./
        }
      }
    }

    guard let match = label.firstMatch(of: regEx) else {
        return nil
    }

    return (
        String(match[title]),
        String(match[number]).replacingOccurrences(of: " ", with: "• ")
    )
}

getTitleAndCardNumber(from: "*4444 название_карты") // (title: "название_карты", cardNumber: "*4444")
getTitleAndCardNumber(from: " 2121 Кредитная MasterCard") // (title: "Кредитная MasterCard", cardNumber: "• 2121")
getTitleAndCardNumber(from: "0001ДебетоваяUnionPay") // nil
```

05. Итоги

- Отчетливо прослеживается курс Apple на реализацию декларативного синтаксиса в языке Swift
- Фреймворк RegexBuilder удачно вписывается в общую стратегию развития и представляет собой инструмент, упрощающий работу с регулярными выражениями
- NSRegularExpression как класс из библиотеки Foundation, а также возможность работы с типом Regex<Output> и литералами должны сохраниться, как альтернатива использованию DSL

СПАСИБО ЗА ВНИМАНИЕ