Seminář 2: Pokročilejší konstrukce jazyka Swift

Tvorba mobilních aplikací

Roman Vyjídáček

Obsah semináře

- Pokročilé koncepty jazyka Swift

Pokročilé koncepty jazyka Swift

Výjimky a jejich zpracování

- Typ výjimky definujeme jako `enum` implementující protokol `Error`
- Funkci která může vyvolat výjimku označíme jako `throws`
- Pomocí `throw` vyvoláme výjimku
- V `do-catch` zachytíme chybu a umožníme její zpracování
- Můžeme však výjimku ignorovat pomocí `try?`

```
enum PasswordError: Error {
    case tooShort
}

func checkPassword(_ password: String) throws
{
    if password.count < 6 {
        throw PasswordError.tooShort
    }
}

do {
    try checkPassword("123")
} catch {
    print("Chyba: heslo je příliš krátké.")
}

// Vrátí `nil` při chybě
let result = try? checkPassword("123")
```

Protokol

- Definuje rozhraní, které musí třída/ struktura splnit
- Nevlastní žádnou implementaci – pouze požadavky
- Pomocí extension (viz dále) ale můžeme vytvořit defaultní implementaci

```
protocol Drivable {
    var speed: Int { get set }
    func drive()
}

class Car: Drivable {
    var speed = 100

    func drive() {
        print("Auto jede rychlostí
              \(speed) km/h.")
    }
}
```

Extensions

- Extensions přidávají nové metody k existujícím typům bez nutnosti dědičnosti
- Je možné v extension definovat i nové konstruktory

```
extension Int {

    func squared() -> Int {
        return self * self
    }
}

let number: Int = 4
print(number.squared())
```

Closures

- Anonymní funkce bez názvu
- Mohou být přiřazeny do proměnné nebo konstanty
- Lze je předávat jako parametr funkcím
- Mají vstupní parametry i návratovou hodnotu
- Mohou zachytávat hodnoty z okolního kontextu
- Podporují zkrácený zápis (closure syntax)

```
let greet = { (name: String) -> String in
    "Ahoj, \(name)"
}

print(greet("Petr"))

func perform(action: () -> Void) {
    action()
}

perform {
    print("Provádím akci")
}

var counter = 0

let increment = { counter += 1 }

increment()
increment()
```

Funkce vyšších řádů: map

- Pracuje s kolekcemi (Array)
- Transformuje každý prvek na novou hodnotu
- Výsledkem je nové pole
- Původní kolekce zůstává nezměněna

```
let numbers = [1, 2, 3, 4]

let squared = numbers.map { number in
    number * number
}

print(squared) // [1, 4, 9, 16]
```

Funkce vyšších řádů: compactMap

- Transformuje prvky kolekce
- Odstraní nil hodnoty
- Používá se při práci s Optional

```
let values = ["1", "2", "x", "3"]

let numbers = values.compactMap { value in
    Int(value)
}

print(numbers) // [1, 2, 3]
```

Funkce vyšších řádů: flatMap

- Transformuje a zplošťuje kolekce
- Odstraňuje vnořené struktury
- Používá se pro práci s kolekcemi kolekcí

```
let nested = [[1, 2], [3, 4], [5]]

let flattened = nested.flatMap { array in
    array
}

print(flattened) // [1, 2, 3, 4, 5]
```

Funkce vyšších řádů: filter

- Pracuje s kolekcemi (Array)
- Vybere prvky splňující podmínku
- Vrací nové pole
- Původní kolekce zůstává nezměněna

```
let numbers = [1, 2, 3, 4, 5, 6]

let evenNumbers = numbers.filter { number in
    number % 2 == 0
}

print(evenNumbers) // [2, 4, 6]
```

Funkce vyšších řádů: reduce

- Slouží ke sloučení prvků kolekce do jedné hodnoty
- Používá počáteční hodnotu akumulátoru
- Postupně zpracovává všechny prvky kolekce

```
let numbers = [1, 2, 3, 4]

let sum = numbers.reduce(0) { result,
number in
    result + number
}

print(sum) // 10
```

Generika

- Umožňují psát znovupoužitelný kód
- Nezávisí na konkrétním datovém typu
- Zvyšují typovou bezpečnost
- Používají se u funkcí, struktur i tříd

```
func swapValues<T>(_ a: inout T, _ b: inout T) {
    let temp = a
    a = b
    b = temp
}

var x = 10
var y = 20

swapValues(&x, &y)
print(x, y)

var first = "A"
var second = "B"

swapValues(&first, &second)
print(first, second)
```

Generika s omezením

- Umožňují omezit generický typ pomocí protokolu
- Zajišťují dostupnost konkrétní funkcionality
- Zvyšují bezpečnost a čitelnost kódu

```
func isEqual<T: Equatable>(_ a: T, _ b: T) ->
Bool {
    a == b
}

print(isEqual(5, 5))
print(isEqual("A", "B"))
```

Generika s omezením pomocí where

- Umožňuje omezit generický typ pomocí podmínky
- Používá se u struktur, tříd i extensions
- Zpřesňuje chování pro konkrétní typy

```
struct Stack<T> {
    var items: [T] = []

    mutating func push(_ item: T) {
        items.append(item)
    }

    mutating func pop() -> T? {
        items.popLast()
    }
}

extension Stack where T: Equatable {
    func contains(_ item: T) -> Bool {
        items.contains(item)
    }
}

var intStack = Stack(items: [1, 2, 3])
print(intStack.contains(2)) // true
```

Associated Types

- Používají se v protokolech místo konkrétního typu
- Umožňují definovat závislý typ
- Často se kombinují s generikou

```
protocol Container {
    associatedtype Item
    var items: [Item] { get set }
}

struct IntContainer: Container {
    var items: [Int]
}

struct StringContainer: Container {
    var items: [String]
}
```

Tuples

- Umožňují seskupit více hodnot dohromady
- Jednotlivé hodnoty mohou mít různé typy
- Hodnoty lze pojmenovat
- Často se používají pro návrat více hodnot z funkce

```
let user = (0, "Bob", false)

print(user.0) // id
print(user.1) // name

let userNamed = (id: 1,
                 name: "Alice",
                 active: true)

print(userNamed.id)
print(userNamed.name)
```

Klíčové slovo lazy

- Inicializují se až při prvním použití
- Používají se pro optimalizaci výkonu
- Jsou dostupné pouze u var

```
class ConfigurationManager {

    lazy var config: [String: String] = {
        print("Inicializuji konfiguraci")

        var values: [String: String] = [:]
        values["environment"] = "production"
        values["apiUrl"] = "https://api.example.com"

        return values
    }()
}

let manager = ConfigurationManager()
print(manager.config)
print(manager.config)
```

Klíčové slovo defer

- Kód se provede při opuštění scope
- Používá se pro úklidové operace
- Provádí se vždy, i při návratu z funkce

```
enum ParseError: Error {
    case empty
    case invalid
}

func parse(_ text: String) throws -> Int {
    print("Start")

    defer { print("Defer: úklid") }

    if text.isEmpty {
        print("Větev: empty")
        throw ParseError.empty
    }

    guard let value = Int(text) else {
        print("Větev: invalid")
        return -1
    }

    print("Větev: success")
    return value
}

do {
    print(try parse("42"))
    print(try parse("x"))
    print(try parse(""))
} catch {
    print("Chyba:", error)
}
```

KeyPath

- Umožňuje bezpečný a typově kontrolovaný přístup k vlastnostem
- Lze je předávat jako hodnotu
- Používají se v generickém a funkcionálním kódu

```
struct User {
    let name: String
    let age: Int
}

let users = [
    User(name: "Alice", age: 30),
    User(name: "Bob", age: 25),
    User(name: "Charlie", age: 40)
]

// Použití s map
let names = users.map(\.name)

// Dynamický přístup pomocí KeyPath
func values<T, V>(from items: [T],
                  keyPath: KeyPath<T, V>) -> [V] {
    items.map { $0[keyPath: keyPath] }
}

let ages = values(from: users, keyPath: \.age)

print(names)
print(ages)
```

WritableKeyPath

- Umožňuje zápis do vlastnosti pomocí KeyPath
- Používá se s var vlastnostmi
- Rozšiřuje KeyPath o možnost změny hodnoty

```swift
struct User {
    var name: String
    var age: Int
}

var users = [
    User(name: "Alice", age: 30),
    User(name: "Bob", age: 25)
]

// Změna jedné vlastnosti pomocí WritableKeyPath
func update<T, V>(
    _ items: inout [T],
    keyPath: WritableKeyPath<T, V>,
    to newValue: V
) {
    for index in items.indices {
        items[index][keyPath: keyPath] = newValue
    }
}

update(&users, keyPath: \.age, to: 18)

print(users)
```

ReferenceWritableKeyPath

- Umožňuje zápis do vlastnosti referenčního typu
- Používá se pouze s class
- Změna se projeví na všech referencích

```
class User {
    var name: String
    var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

let user1 = User(name: "Alice", age: 30)
let user2 = user1

func update<T, V>(
    _ object: T,
    keyPath: ReferenceWritableKeyPath<T, V>,
    to newValue: V
) {
    object[keyPath: keyPath] = newValue
}

update(user1, keyPath: \.age, to: 40)

print(user1.age) // 40
print(user2.age) // 40
```

Přetěžování operátorů

- Umožňuje definovat vlastní chování operátorů
- Operátory jsou běžné funkce
- Lze je definovat globálně nebo jako statické metody
- Používá se pro práci s vlastními typy

```
struct Vector2D {
    let x: Double
    let y: Double
}

func + (lhs: Vector2D, rhs: Vector2D)
-> Vector2D {
    Vector2D(
        x: lhs.x + rhs.x,
        y: lhs.y + rhs.y
    )
}

let a = Vector2D(x: 1, y: 2)
let b = Vector2D(x: 3, y: 4)

let c = a + b
print(c)
```

Úkoly

https://vyjidacek.cz/tmai/seminar2.php