ackee edu

BI-IOS

Lukáš Hromadník, Jakub Olejník, Igor Rosocha

Lekce 1 - Základy Swiftu

Základní typy

Swift obsahuje sadu základních typů. Mezi nejpoužívanější patří

- `String`
- `Int`, `Double`, `Float`
- `Bool`

Konstantní proměnná

Definovaná pomocí klíčového slova `let`. Po inicializaci hodnotou její hodnota nelze změnit.

```
let string: String = "nějaký text"
let bool = false
```

Pokud bychom během běhu programu chtěli změnit hodnotu jedné z výše zadefinovaných proměnných, dostaneme chybu během kompilace.

```
string = "nějaký jiný text"
```

```
// error: cannot assign to value: 'string' is a 'let' constant
// note: change 'let' to 'var' to make it mutable
```

Mutable variable

Pokud chceme za běhu programu měnit hodnotu uvnitřní proměnné, je potřeba použít mutable variantu.

Ta je definovaná pomocí klíčového slova `var` a lze dále v programu měnit její hodnotu.

```
var number = 0

number = 1 // No problem here
```

Řídící struktury

`If` - `elseif` - `else` určitě každý dobře zná.

Není potřeba psát kulaté závorky `(` `)` na ohraničení podmínky. Je možné je napsat, ale nedoporučujeme to.

```
if condition {
    // ...
} else if anotherCondition {
    // ...
} else {
    // ...
}
```

`guard`

Swift poskytuje i ”obrácený” `if` konstrukt, který vám dovoluje někde v kódu zkontrolovat, jestli daná podmínka platí.

Výhoda oproti `if` u je hlavně ta, že pomocí `guard` u se nezanoříte do nějaké větve `if` u, ale můžete vesele pokračovat dál bez odsazení.

```
let i = 0

guard i > 0 else { return }

// Tady vždy i > 0
```

Cykly

Ve Swiftu je zakázaný přístup k psaní "C"-čkového `for` cyklu. Nyní je dostupná pouze iterace přes danou kolekci.

V ukázce níže vidíte datový typ `Range`, který definuje rozsah mezi dvěma čísly. `0..<10` vytvoří `Range` obsahující čísla `0, ..., 9`. Pokud bychom chtěli udělat `Range` včetně pravé strany, změníme `..<` na `...`.

```
for i in 0..<10 {
    // Postupně pro i od 0 až do 9
}
```

Mezi další cykly patří `while` a `repeat-while`.

Rozdíl mezi nimi je v čase, kdy se kontroluje podmínka na pokračování v cyklu. U `while` se podmínka kontroluje na začátku, ale u `repeat-while` až na konci. U `repeat-while` tak dojde vždy minimálně k jednomu průchodu vnitřkem cyklu.

```
while condition {
    // ...
}

repeat {
    // ...
} while condition
```

Pattern matching

Swift má zabudovaný pattern matching – mechanismus ke kontrole elementů, zda-li odpovídají zadanému vzoru. Pomocí `where` specifikujete podmínku kladenou na jednotlivé elementy.

Lze ho například využít ve `for` cyklech.

```
for i in 0..<10 where i % 2 == 0 {
    print(i)
}
// prints 0
// prints 2
// prints 4
// prints 6
// prints 8
```

Funkce / metody

Definujeme pomocí klíčového slova `func`. Jednotlivé argumenty mají *vnitřní* a *vnější* pojmenování. Toto pojmenování je potřeba použít při volání funkce.

Pokud použijeme pouze jeden název pro argument, použije se tento název jak pro vnější tak vnitřní pojmenování.

```
func multiply(number: Int, by multiplier: Int) -> Int {
    return number * multiplier
}

multiply(number: 2, by: 4) // Returns 8
```

Pokud nechceme mít argument pojmenovaný z volání funkce, na místo pro vnější jméno dáme podtržítko `_`. Vnitřní pojmenování musíme zvolit.

```
func multiplyNumber(_ number: Int, by multiplier: Int) -> Int {
    return number * multiplier
}

multiplyNumber(2, by: 4) // Returns 8
```

Toto je preferovaný způsob pojmenování funkce než předchozí verze.

```
multiplyNumber(2, by: 4)   // Preferred
multiply(number: 2, by: 4) // Not preferred
```

Defaultní hodnotu pro argument přidáme přiřazením hodnoty, stejně jako u proměnné. Díky pojmenování argumentů můžeme defaultní hodnotu nastavit libovolnému argumentu.
Z volání funkce potom bude jasné, u kterých argumentů se má použít defaultní hodnota a které mají hodnotu předanou z volání.

Správné pojmenování funkce a jejich argmentů pomáhá dokumentaci kódu. Z volání funkce lze pouze jejím přečtením zjistit, co daná funkce dělá.

Implicit return

```
func foo(_ bar: Int) -> Bool {
    return barInner > 0
}

func foo(_ bar: Int) -> Bool {
    barInner > 0
}
```

Optionals

```
var optionalValue: String? = "x"
// ...
optionalValue = nil
```

Optional proměnná buď obsahuje hodnotu a je rovna `x` anebo vůbec nemá hodnotu přiřazenou.

Jediný datový typ, který má toto chování. Zbytek typů má garantovanou hodnotu.

Implicit init

```
var value: String? // no need to specify the right side
```

Použití v kódu

```
let optionalCount: Int? = optionalValue?.count
```

```
if let value = optionalValue {
    // here value: String
}
```

```
guard let value = optionalValue else { return }
// here value: String
```

Unwrapping

```
let nonOptionalCount: Int = optionalValue!.count
```

Nil-Coalescing operator

```
let nonOptionalCount: Int = optionalValue?.count ?? 0
```

Implicitly unwrapped optionals

```
var futureValue: Int!

print(futureValue) // will crash

futureValue = 24
print(futureValue) // no need to unwrap
```

Tuples

Zapouzdření několika hodnot do jednoho složeného typu

```
let tuple = (1, "ahoj")
print(tuple.0) // prints 1
print(tuple.1) // prints ahoj
```

```
let cleverTuple = (id: 1, name: "ahoj")
print(cleverTuple.id) // prints 1
print(cleverTuple.name) // prints ahoj
```

Pole

```
let array: [String] = []
let array = [String]()
let array = ["jak", "se", "mas"]

print(array[2]) // prints mas

for element in array {
    print(element)
}
// jak
// se
// mas
```

Dictionary

```
let dict: [String: Int] = [:]
let dict = [String: Int]()
let dict = ["prvni": 1, "druhy": 2, "treti": 3]

print(dict["treti"]) // prints 3
```

Enum

```
enum MyType {
    case first
    case second
    case third
}

let value = MyType.first
let value: MyType = .first

let value = MyType() // not possible
```

```
let value: MyType = .first

switch value {
case .first:
    fallthrough

case .second:
    break

case default:
    print("third")
}
```

Při použití `case default` přijdete o compile check, že jste použili všechny případy

Struct

```
struct Person {
    let name: String
    let surname: String

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

let person = Person(name: "Lukáš", surname: "Hromadník")
```

Mají kompilátorem generovaný `init`, pokud neexistuje `private` proměnná.

```
struct Person {
    let name: String
    let surname: String
    let age: Int
}

let person = Person(
    name: "Lukáš",
    surname: "Hromadník",
    age: 42
)
```

Jedná se o `value-type`, tedy při práci s nimi dochází ke kopírování a vytváření nových instancí.

```
var a: Int = 42
var b: Int = a

b = 0

print(a, b)
```

Mutace

```
struct Person {
    let name: String
}

let p1 = Person(name: "Jan")
p1.name = "Honza" // Nope, not possible
```

Mutace

```
struct Person {
    var name: String
}

let p1 = Person(name: "Jan")
p1.name = "Honza" // Nope, still not possible
```

Mutace

```
struct Person {
    var name: String
}

var p1 = Person(name: "Jan")
p1.name = "Honza" // Good
```

Class

Reference-type a nemá implicitní `init`, lze ho nechat vygenerovat

```
class Person {
    var name: String
    let surname: String
    lazy var fullName = name + " " + surname

    init(name: String, surname: String) {
        self.name = name
        self.surname = surname
    }
}
```

Mutace

```
class Person {
    let name: String
    let surname: String

    // Init přeskočen pro jednoduchost
}

let person = Person(name: "Lukáš", surname: "Hromadník")
person.name = "Jan" // Nope
```

Mutace

```
class Person {
    var name: String
    let surname: String

    // Init přeskočen pro jednoduchost
}

let person = Person(name: "Lukáš", surname: "Hromadník")
person.name = "Jan" // Good

person = Person(name: "Jan", surname: "Fit") // Nope
```

Dědičnost

```
class Car {
    var model: String

    init() {
        self.model = "Škoda"
    }
}

class BmwCar: Car {
    override init() {
        super.init()

        self.model = "BMW"
    }
}
```

Protokoly

Definice interfacu pro nějaký datový typ

```
protocol Animal {
    var sound: String { get }
    var name: String { get set }

    func makeSound()
}
```

```
protocol Animal {
    var sound: String { get }
    func makeSound()
}


struct Dog: Animal {
    let breed: String
    let sound: String

    func makeSound() {
        print(sound)
    }
}

let someAnimal: Animal = Dog(breed: "Husky", sound: "Woof")
print(someAnimal.sound)
someAnimal.makeSound()

print(someAnimal.breed) // Nope, not possible
```

Extensions

```
protocol Animal {
    func makeSound()
}

struct Dog {
    let breed: String
}

extension Dog: Animal {
    func makeSound() { print("Woof") }
}
```

Protocol extensions

```
protocol Animal {
    var sound: String { get }
}

extension Animal {
    func makeSound() { print(sound) }
}

extension Dog: Animal {
    var sound: String {
        "Woof"
    }
}
```

Access control

```
struct Person {
    // Cannot be use anywhere else but right here
    // in the `struct` scope
    private let id: Int

    // Can be use outside the `struct` scope
    // but only within current file
    fileprivate let id2: Int
```

```
    // Visible in the current module
    // Very similar to `protected` in other languages
    // Also this is default
    internal let name: String

    // Visible outside of the module, e.g. imported framework
    public private(set) var surname: String

    // Can be overriden outside the module
    open let age: Int
}
```

```swift
public struct Person {
    private let id: Int
    public let name: String

    // Not generated by default
    public init(id: Int, name: String) {
        self.id = id
        self.name = name
    }
}

// Outside the module
let person = Person(id: 1, name: "User")
print(person.id) // Not visible here
```

Access control in Extensions

```
public extension Person {
    func makeSound() {
        print("I am " + name)
    }
}

// Outside the module
let person = Person(id: 1, name: "User")
person.makeSound() // print I am User
```

Generika

```
class Stack<Element> {
    private var items: [Element] = []

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

    func pop() -> Element {
        items.removeLast()
    }
}

let stack = Stack<Int>()
```

Closures

Blok kódu, který může zachytit hodnoty proměnných ve jeho scope

```
let closure: (Int) -> String = { number -> String in
    return String(number)
}
```

Closures

```
let closure: (Int) -> String = { number -> String in
    return String(number)
}

let closure: (Int) -> String = { number in
    String(number)
}

let closure: (Int) -> String = { String($0) }

let closure: (Int) -> String = String.init
```

Trailing closure syntax

```
func travel(action: () -> Void) {
    print("I'm getting ready to go.")
    action()
    print("I arrived!")
}

travel(action: { print("On vacation") })

travel() {
    print("On vacation")
}

travel {
    print("On vacation")
}
```

Multiple trailing closures

```
struct Button<Content: View> {
    init(
        action: () -> Void,
        label: () -> Content
    )
}

let button = Button(
    action: { },
    label: { }
)

let button = Button {
    // action
} label: {
    // closure pro nastavení label
}
```