Tvorba mobilních aplikací

3. seminář

Radek Janoštík

Univerzita Palackého v Olomouci

1. 3. 2022

Nullable vs. non-nullable (1 / 2)

- V kotlinu není dovoleno přiřadit **null** do referenčního datového typu
  val str : **String** = **null** `//copile-time chyba`
- Každý *non-nullable* datový typ má *nullable* protějšek
  val str2 : **String**? = **null**
  - Otazník nás bude neustále upozorňovat, že se v proměnné může objevit **null**

Nullable vs. non-nullable (1 / 2)

- V kotlinu není dovoleno přiřadit **null** do referenčního datového typu
  val str : **String** = **null** `//copile-time chyba`
- Každý *non-nullable* datový typ má *nullable* protějšek
  val str2 : **String**? = **null**
  - Otazník nás bude neustále upozorňovat, že se v proměnné může objevit **null**
- K dispozici *safe-call* operátor (známe ze C#)
  val substr: **String**? = str2?.substringBefore('x')
- Je ekvivalentní s:
  val substr: **String**? = **if** (str2!=**null**) str2.substringBefore('x') **else null**

Nullable vs. non-nullable (2 / 2)

- Safe-call operátor v případě přítomnosti **null** vrátí vždy **null**
- Často je potřeba vrátit něco jiného (výchozí hodnotu) → *Elvis operátor*
  return str2?.substringBefore('x') ?: `"default value"`
- Pro potlačení kontroly kompilátor na **null** máme *Joker operátor*
  var jokerStr = str3 !!. substring(1..4)
  - Nedoporučuje se používat – přijdeme o kontrolu kompilátorem
  - Může nám tedy program spadnout za běhu

Výjimky

- Systém umožňující reagovat výjimečné situace
- V Kotlinu nejsme nuceni výjimky ošetřovat – „chytat“
- **try** – **catch** – **finally** blok stejný jako v Javě/C# **ALE** je *výraz*

  ```
  val a: Int? = try { input.toInt() } catch (e: NumberFormatException) { null }
  ```

- Výsledkem bude poslední řádek z **try** nebo z **catch** (nebude z **finally**)
- Nemáme *checked* a *unchecked* výjimky – nemusíme nic psát k hlavičkám funkcí a metod
- „Vyhození“ pomocí výrazu **throw** např **throw** IllegalArgumentException(message)
- Všechny výjimky dědí z `Throwable`

OOP v Kotlinu

- OOP v Kotlinu není vynucováno, můžeme psát klidně čistě procedurálně (top-level funkce) i funkcionálně
- Snaha o jednoduchost, nepsat „zbytečnosti“
  - O co se může postarat kompilátor, nechť se postará
- Výrazně odlišný přístup ke statickým metodám
- Rozšiřující metody (*extension methods*)

Anonymní objekty I

- „Když programátor chce objekt, ať jej dostane bez omáčky okolo“
  - *Výraz*: klíčové slovo **object** následované složenými závorkami `{}` vytvoří anonymní objekt

```
fun writePerson() {
    val somePerson = object {
        val bNumber = "123456789/8765"
        var name = "Radek"
        var surname = "Janostik"
        fun changeName(newName: String) {
            name = newName
        }
    }
    println ("${somePerson.bNumber} - ${somePerson.name}")
    somePerson.changeName("Darek")
    println ("${somePerson.bNumber} - ${somePerson.name}")
}
```

Anonymní objekty II

- Limity anonymních objektů
  - Nemohou být jako návratový typ metod a funkcí
  - Nemohou být jako typy parametrů metod a funkcí
  - Jsou-li jako vlastnost nějaké třídy, jsou vždy typu **Any** (není přístup k metodám)
- Časté použití pro rychlou implementaci nějakého rozhraní

```
fun createRunnable(): Runnable {
    val result = object: Runnable {
        override fun run() {
            println ("Spoustim slozity vypocet pomoci anonymniho rozhrani")
        }
    }
    return result // ted vratit muzu (proc?)
}
```

- Zkráceně: `fun createRunnableShort(): Runnable = Runnable { println("spoustim!")}`

Singleton – Jedináček

- Známe návrhový vzor? K čemu se používá?

Singleton – Jedináček

- Známe návrhový vzor? K čemu se používá?
  - Zajištění, že bude existovat jediná instance dané třídy
  - Většinou pro kontrolu nad zdroji (databáze, síť, . . .)

Singleton – Jedináček

- Známe návrhový vzor? K čemu se používá?
  - Zajištění, že bude existovat jediná instance dané třídy
  - Většinou pro kontrolu nad zdroji (databáze, síť, . . .)
- Pojmenujeme-li anonymní objekt vznikne *příkaz*, který vytvoří anonymní interní třídu
- Máme ihned zajištěno, že bude existovat pouze jedna jediná instance

```
object Database {
    val db = mutableListOf<Int>()
    fun printDb()  = db.forEach() { println ( it ) }
}
fun main(args: Array<String>) {
     Database.db.add(5)
}
```

Singleton – Jedináček

- Známe návrhový vzor? K čemu se používá?
  - Zajištění, že bude existovat jediná instance dané třídy
  - Většinou pro kontrolu nad zdroji (databáze, síť, . . .)
- Pojmenujeme-li anonymní objekt vznikne *příkaz*, který vytvoří anonymní interní třídu
- Máme ihned zajištěno, že bude existovat pouze jedna jediná instance

```
object Database {
    val db = mutableListOf<Int>()
    fun printDb() = db.forEach() { println ( it ) }
}
fun main(args: Array<String>) {
     Database.db.add(5)
}
```

- Jak udělat singleton v Javě a C#?

Třídy – definice

- Vytvoření třídy co nic neumí: **class** SmallestClassEver
  - Automaticky se vytváří konstruktor bez argumentů

Třídy – definice

- Vytvoření třídy co nic neumí: **class** SmallestClassEver
  - Automaticky se vytváří konstruktor bez argumentů
- Vytvoření třídy s jednou read-only *vlastností*(*property*): **class** Person(**val** id: **String**)
  - Kompilátor vytvoří primární konstruktor
  - Definuje *slot*, vytvoří *getter*
  - Přístup ke všem členům tříd je **public** jako výchozí

Třídy – definice

- Vytvoření třídy co nic neumí: **class** SmallestClassEver
  - Automaticky se vytváří konstruktor bez argumentů
- Vytvoření třídy s jednou read-only *vlastností*(*property*): **class** Person(**val** id: **String**)
  - Kompilátor vytvoří primární konstruktor
  - Definuje *slot*, vytvoří *getter*
  - Přístup ke všem členům tříd je **public** jako výchozí
- Předchozí je zkratka za
  public class Person **public constructor**(**public val** id: **String**)
- **var** a **val** určuje, zda bude vlastnost jen pro čtení, či ne

```
class Person2(val id: String, var address: String)
val p: Person2 = Person2("123456789/8740", "17. listopadu, 12")
println (p.address)
```

- Ověření, co se děje na pozadí: `$ javap -P Person2.class`

Třídy – práce s vlastnostmi

- Bývá potřeba mít kontrolu nad settery, můžeme dodefinovat vlastní

```
class Person3(val id: String, addressParam: String) {
    var isAlive = true n
    var address = addressParam
        set(value) {
            if (value.isBlank()) {
                throw RuntimeException("Address should not be empty")
            }
            field = value
        }
}
```

- addressParam je obyčejným parametrem – chybí **var** i **val** → nevznikne vlastnost
- Automaticky se vytvoří vlastnost **var** isAlive typu Boolean s getterem a setterem
- Definujeme vlastní setter s validací – klíčové slovo **field**

Třídy – modifikátory přístupu

- V Kotlinu máme *(naštěstí)* jen 4 modifikátory přístupu ke členům tříd
- **public** – přístup není omezen, může přistupovat/volat každý
- **private** – přístup maximálně omezen – pouze objekt samotný

Třídy – modifikátory přístupu

- V Kotlinu máme *(naštěstí)* jen 4 modifikátory přístupu ke členům tříd
- **public** – přístup není omezen, může přistupovat/volat každý
- **private** – přístup maximálně omezen – pouze objekt samotný
- Tedy stejně jako v Javě

Třídy – modifikátory přístupu

- V Kotlinu máme *(naštěstí)* jen 4 modifikátory přístupu ke členům tříd
- **public** – přístup není omezen, může přistupovat/volat každý
- **private** – přístup maximálně omezen – pouze objekt samotný
- Tedy stejně jako v Javě
- **protected** – přístup mají pouze odvozené třídy *(potomci)*
- **internal** – přístup odkudkoliv ze stejného *balíčku*
  - Všechny zdrojové kódy, kterou se kompilují spolu
  - Nemá oporu v bytekódu, kompilátor přidává interní konvence pojmenování
- Getter má stejnou úroveň přístupu jako vlastnost, setter můžeme modifikovat

Třídy – inicializace

- Potřebujme-li při inicializaci provést nějaký kódu máme **init** blok
- Může jich být více (nedoporučuje se, odshora dolů), vidí pouze vlastnosti definované „nad nimi“

```
init {
if (address.isBlank()) throw RuntimeException("Address should not be
    blank")
}
```

- Konstruktorů může být více, volání „konstruktoru z konstruktoru“ pomocí dvojtečky

```
constructor(id: String, addressParam: String, isAlive: Boolean) : this(id, addressParam) {
        this.isAlive = isAlive
    }
```

Třídy – metody

- Metody se definují pomocí klíčového slova **fun**
- Úplně stejně jako funkce, akorát jsou součástí třídy
- Stejné modifikátory přístupu jako výše

```
fun firstPartOfId () = id.substringBefore ('/')
private fun secondPartOfId() = id.substringAfter ('/')
```

Třídy – statičtí členové

- Statické metody/sloty jsou spjaty se třídou, nikoliv s konkrétní instancí
- V Kotlinu není obdoba `static`

Třídy – statičtí členové

- Statické metody/sloty jsou spjaty se třídou, nikoliv s konkrétní instancí
- V Kotlinu není obdoba `static`
- K dispozici *companion* objekt – „singleton pro třídu“
  - Mohou implementovat rozhraní
  - Mohou dědit
  - Může být pojmenován

```
companion object { // lze pojmenovat
        var numberOfInstances = 0
            private set
        val greet = "Hello Person!"
        fun someStaticFunction(arg1: Int) = arg1*3.1415
    }
```

- Vhodný také pro implementaci návrhového vzoru *továrna(factory*

Třídy – generika

- Motivace: Závisí funkcionalita seznamu na datovém typu jeho prvků?

Třídy – generika

- Motivace: Závisí funkcionalita seznamu na datovém typu jeho prvků?
- $\Rightarrow$ NE – až na typy argumentů a návratových hodnot

Třídy – generika

- Motivace: Závisí funkcionalita seznamu na datovém typu jeho prvků?
- $\Rightarrow$ NE – až na typy argumentů a návratových hodnot
- Generické třídy/datové typy $\rightarrow$ podobné jako v Javě a C#

```
class ListNode<T>(val value: T) where T: Comparable<T>, T : Runnable {
    var next: ListNode<T>? = null
}
```

- Je-li omezení jen jedno, může být rovnou v ostrých závorkách:

```
class ListNode2<T: Comparable<T>> (val value: T) {
    var next: ListNode2<T>? = null
}
```

Datové třídy

- Speciální třídy spíše jako nosiče dat než funkcionality
- Konstruktor musí definovat alespoň jednu vlastnost, může obsahovat parametr
- Automatické vygenerování metod equals(), hashCode(), toString()
- Vytvoření metody copy() pro kopii objektu(všechny vlastnosti) se změnou některých vlastností
  - Nekopíruje „do hloubky“
- Pro vlastnosti z **primárního konstruktoru** vytvořeny metody component1() ...componentN()
  - Vhodné pro destruktualizaci/dekonstrukci

```
val log1 = MeteoLog(10.2, 64.7, 3.5)
println(log1)
val log2 = log1.copy(temperature = 12.6)
println(log2.component2())
val (temp, _, wind) = log1 // dekonstrukce
println(temp)
```

Rozhraní

- Stěžejní koncept v OOP, dnes se často „programuje proti rozhraní“
  - Je pro nás důležité, jak se objekt chová, co umí
  - Snadná záměna výsledné implementace (testování, mockování, . . .)
- Velmi podobné jako v Javě
  - Mohou implementovat metody (bez klíč. slova `default`)
  - Mohou mít statické metody (přes companion objekt!)

```
interface Stack<T> {
    fun size() : Int
    fun pop(): T?
    fun peek(): T?
    fun push(number: T)
    fun print () : String
}
```

Rozhraní – implementace

```
class MyStack<T> : Stack<T> {
    val data = mutableListOf<T>()
    var members = -1

    private fun isEmpty() = members < 0

    override fun size(): Int = members + 1

    override fun pop(): T? = if (!isEmpty()) data[members--] else null

    override fun peek(): T? = if (!isEmpty()) data[members] else null

    override fun push(value: T) {
        data.add(value)
        members++
    }
```

Abstraktní třídy

- Nemohou mít konkrétní instanci – slouží jako *základní* třídy pro hierarchii objektů
- Mohou implementovat některé metody.
- Abstraktní metody(=bez implementace) musí být označeny **abstract**

```
abstract class Worker(val name: String, var salary: Double) {
    abstract fun whatCanDo(): String
    fun nameToSalary() = name to salary
}
class DemolitionMan(name: String, salary:Double, val nickName: String) : Worker(name,
     salary) {
    override fun whatCanDo(): String = "Destroy your building"
    fun sayMyNameIs() = "My name is $name, please call me $nickName"
}
```

- Vlastnosti abstraktních třídy mají „na pozadí“ slot, rozhraní nikoliv
- Třída může implementovat více rozhraní, ale dědit pouze z jedné třídy

Dědičnost

- Ve výchozím stavu jsou všechny třídy *final* (= nelze z nich odvozovat třídy)
  - Pro možnost dědičnosti potřeba označit třídu jako **open**
  - Každou metodu, která bude moc být *přepsána* také
  - $\Rightarrow$ větší bezpečnost – nikdy se nestane, že by někdo přepsal metodu, kterou autor základní třídy nepovolil přepsat
- V potomcích je možné přepisovat i vlastnosti
  - **val** vlastnost může být přepsána **val** i **var** vlastností v potomkovi
  - **var** vlastnost může být přepsána pouze **var** vlastností v potomkovi

```
open class Agent(id:String, addressParam:String, var salary: Double, var agency: String)
    : Person5(id, addressParam) {
    final override fun toString() = "Employee $id with salary $salary" //
        zadny potomek uz neprepise
    fun workHard() {
        println ("I'am working hard")
    }

    override fun firstPartOfId () = "Blunt, James Blunt"
}
```

Výčtové typy

- Datový typ, který může nabývat pouze určitých hodnot

  enum class Position {WORKER, FOREMAN, MANAGER, OWNER}

- Získání hodnoty z řetězce: **val** pos = Position.valueOf("MANAGER")
- Iterace přes hodnoty

```
for (pos in Position.values()) {
        println ("${pos.name} ${pos.ordinal}")
}
```

- Je to třída, může mít vlastnosti i metody

```
enum class Position2(val greet: String, val bonus: Double) {
    WORKER("Hi, my hardworker", 0.0),
    FOREMAN("Mr., how are they working?", 1.1),
    MANAGER("Sir, is company prospering?", 1.25),
    OWNER("Good morning, our lord", Math.PI) {
        override fun greetWithPosition() = "I will not greet!"
    }; // povinny strednik!
```

Rozšiřující metody (extension methods)

- V Kotlinu můžeme rozšířit jakoukoliv třídu o metodu, aniž bychom z ní dědili
  fun Agent.killedAgents() = Random.nextInt(0,20)
- Metodu můžeme volat jako klasickou metodu: println (agent.killedAgents())
- Rozšiřující metody nemohou přistupovat k **private** ani k **protected** členům
- Při shodě jmen má vždy přednost člen třídy
- *Viditelnost* rozšiřující metody je stejná, jako viditelnost její definice

```
fun String.underscoreSameLength() : String {
    val sb = StringBuilder(this.length)
    for (ch in this) {
        sb.append("_")
    }
    return sb.toString()
}
println(agent.id.underscoreSameLength())
```

Úkol (1 / 2)

1. Rozšiřte datový typ **String** o metodu toggle()
   - Vrátí nový řetězec, kde bude prohozena velikost písmen (z malých udělá velká a naopak)
   - Mezery nahradí hvězdičkou

2. Navrhněte `enum` reprezentující planety Sluneční soustavy
   - Kromě jména planety přidejte i vlastnosti pro hmotnost, vzdálenost od Slunce a dobu oběhu kolem Slunce
   - Implementujte metodu **fun** kmByDays(days: **Double**), která vypočítá, kolik kilometrů urazí planeta za daný počet (pozemských)dnů

Úkol (2 / 2)

3. Navrhněte rozhraní pro hashovací tabulku obsahující libovolné objekty stejného typu
   - přidání, odebrání, test přítomnosti, ... (ještě něco?)

4. Toto rozhraní implementujte vlastní hashovací tabulkou, kolize řešte řetězením