Predavanje br. 3

Razvoj mobilnih aplikacija i servisa

dr. sci. Alma Šećerbegović, van. prof.

Sadržaj

- Kotlin
  - Ponavljanje prethodnog goriva
  - Liste, mape i filteri liste
  - Null sigurnost
  - Kompaktne funkcije
  - Lambda funkcije i funkcije višeg reda
  - Klase i objekti
  - Nasljeđivanje

Ponavljanje gradiva

Primjer 1

Napisati funkciju u Kotlinu koja provjerava da li je broj Armstrongov broj.

Armstrongov broj (poznat i kao Narcissusov broj) je broj koji je jednak zbiru svojih cifara podignutih na stepen jednak broju cifara u tom broju. Na primjer:

- **153** je Armstrongov broj jer $1^3+5^3+3^3= 1+125+27=153$.
- **9474** je Armstrongov broj jer $9^4+4^4+7^4+4^4=9474$.

Kolekcije - List, Map, Set

List

- Liste su uređena kolekcija elemenata.
- Dozvoljavaju ponovljene vrednosti.
- Svi elementi u listi kada deklarirane sa tipom, moraju imati isti tip.

```
val intList: List<Int> = listOf(1, 2, 3)
val doubleList: List<Double> = listOf(1.1, 2.2, 3.3)
val stringList: List<String> = listOf("Faris", "Amila", "Mujo")
```

- String.split() metoda vraća listu tipa List<String>

```
val words = "Kotlin je super".split(" ")   // ["Kotlin", "je", "super"]
```

- List<Any> može sadržati elemente različitih tipova
- List<List<Int>> (ugniježdene liste) omogućava rad sa dvodimenzionalnim kolekcijama

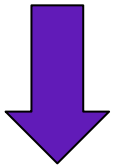
Liste

- Pristup elementima liste
  - Indeksiranje cijelim brojevima: počinje od nule
  - .size je broj elemenata
  - Koristite .get(i) ili [i] za pristup elementima po indeksu
    - .getOrNull(i) vraća null ako je i nevažeći
    - .getOrElse(i) {izraz} vraća izraz ako je i nevažeći
    - Koristite .last() za pristup poslednjem elementu
    - .indexOf() i .lastIndexOf() za pronalaženje indeksa elemenata
- Metodi
  - contains i containsAll

Filteri liste

Vrati dio liste koji odgovara određenom uslovu

red	red-orange	dark red	orange	bright orange	saffron



Primjeni `filter()` na listi

Uslov: element liste sadrži riječ “red”

red	red-orange	dark red

Iteracija kroz liste

Ako funkcija kao argument uzima samo jedan parametar, možete izostaviti njegovu deklaraciju i strelicu "->". Parametar je implicitno deklarisan pod imenom it

```
val ints = listOf(1, 2, 3)
ints.filter { it > 0 }
```

Filteri prolazi kroz listu, gdje `it` je vrijednost elementa u listi.
Ovo je isto kao i:

```
ints.filter { n: Int -> n > 0 }   OR   ints.filter { n -> n > 0 }
```

Filteri liste

Uslov filtera u zagradama `{}` provjerava svaki element liste. Ako izraz u zagradama vrati `true`, taj element se uključuje.

```
val books = listOf("nature", "biology", "birds")

println(books.filter { it[0] == 'b' })
```

```
⇒ [biology, birds]
```

Eager i lazy filteri

Evaluacija izraza u listama:

- **Eager:** dešava bez obzira da li će se rezultat ikad koristiti
- **Lazy:** dešava se samo ako je neophodno pri izvršavanju koda

Lazy evaluacija listi je korisna kada vam nije potreban čitav rezultat ili kada je lista izuzetno velika i više kopija ne bi stale u RAM.

Eager filteri

Filters su eager po default-u. Nova lista se kreira svaki put kad se koristi eager filter.

```
val instruments = listOf("viola", "cello", "violin")
val eager = instruments.filter { it [0] == 'v' }
println("eager: " + eager)
```

```
⇒ eager: [viola, violin]
```

Lazy filteri

Sekvence (Sequences) su strukture podataka koje koriste odloženu evaluaciju (lazy evaluation) i mogu se koristiti sa funkcijama za filtriranje kako bi se filtriranje izvršavalo na odložen način.

```
val instruments = listOf("viola", "cello", "violin")
val filtered = instruments.asSequence().filter { it[0] == 'v'}
println("filtered: " + filtered)
```

```
⇒ filtered: kotlin.sequences.FilteringSequence@386cc1c4
```

Sequences -> lists

Sequence se mogu pretvoriti u liste korištenjem `toList()`.

```
val filtered = instruments.asSequence().filter { it[0] == 'v'}
val newList = filtered.toList()
println("new list: " + newList)
```

```
⇒ new list: [viola, violin]
```

Ostale transformacije liste

- `map()` funkcija primenjuje istu transformaciju na svaki element i vraća listu sa transformisanim elementima.

```
val numbers = setOf(1, 2, 3)
println(numbers.map { it * 3 })
=> [3, 6, 9]
```

- `flatten()` vraća jednu listu sa svim elementima iz ugnježdenih kolekcija.

```
val numberSets = listOf(setOf(1, 2, 3), setOf(4, 5), setOf(1, 2))
println(numberSets.flatten())
=> [1, 2, 3, 4, 5, 1, 2]
```

Set

- Set je neuređena kolekcija jedinstvenih elemenata.
- Ne dozvoljava ponavljanje elemenata.
- Koristi se kada vam je potrebna kolekcija bez duplikata.

```
val numberSet: Set<Int> = setOf(1, 2, 3, 2)  // [1, 2, 3] – duplikati se ignorišu
```

- MutableSet omogućava dodavanje i uklanjanje elemenata:

```
val mutableSet: MutableSet<String> = mutableSetOf("Apple", "Banana")

mutableSet.add("Orange")

mutableSet.remove("Banana")
```

Map

- Mapa je kolekcija parova ključ-vrijednost.
- Svaki ključ je jedinstven, ali vrijednosti mogu biti iste.

```
fun main() {
    val numMap = mapOf("one" to 1, "two" to 2, "zero" to 0)
    val another = mapOf("zero" to 0, "one" to 1, "two" to 2)

    println("numMap: $numMap")
    println("another: $another")
    println("numMap == another: ${numMap == another}")
    println("string representations are equal: ${"$numMap" == "$another"}")
}
```

- Get metodi
  - .get(key) i [key] vraćaju null ako .keys ne sadrži dati ključ.
  - .getValue(key) baca izuzetak (Exception) ako mapa ne sadrži dati ključ.
  - getOrElse(key) {izraz} vraća vrijednost izraza ako mapa ne sadrži dati ključ.
  - getOrDefault(key, defaultValue) vraća defaultValue ako mapa ne sadrži dati ključ.

Primjer

U prodavnici su svi proizvodi sačuvani u mapi Map<String, Int>, koja sadrži parove naziv proizvoda - cijena.

Kupac dolazi sa spiskom za kupovinu i želi da zna koja će biti ukupna cijena proizvoda sa tog spiska. Imajte na umu da neki proizvodi možda nisu dostupni u prodavnici.

Dati su vam Map<String, Int> (naziv proizvoda i cijena) i MutableList<String> (spisak za kupovinu) kao parametri funkcije. Vratite ukupnu cijenu proizvoda koje kupac želi da kupi.

Null sigurnost

Null sigurnost

- U Kotlinu varijable ne mogu biti null po default-u

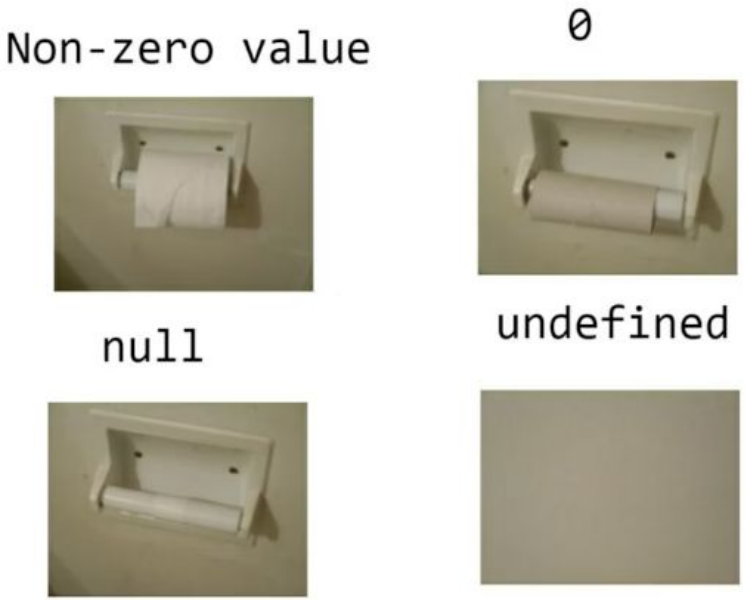
```
var name: String? = null
```

```
var age: Int? = null
```

- Kako pristupiti ovakvoj varijabli?

```
if (name != null) {
    print(name.length)
}
```

```
print(name?.length)
```



Sigurni pozivi (safe calls) - ?.

- Dat je sljedeći kod. Koji je izlaz?

```
val name: String? = "Kotlin"
val length = name.length
```

- Rješenje: null provjera

```
val name: String? = "Kotlin"
val length = if (name != null) name.length else null
```

- Operator ?.

```
val name: String? = "Kotlin"
val length = name?.length // length je null ako je name null
```

```
val street = city?.address?.street // isto kao i izraz ispod

val street = if (city != null && city.address != null)
    city.address.street else null
```

Siguran Poziv (Safe Call)

`someThing?.otherThing` neće izazvati `NullPointerException` (NPE) ako je `someThing` `null`.

Na primjer, zaposleni može biti dodjeljen nekom odjeljenju (ili ne). To odjeljenje može imati šefa, koji može, ali i ne mora, imati ime koje želimo ispisati:

```
fun printDepartmentHead(employee: Employee) {
        println(employee.department?.head?.name)
}
```

Ako želite ispisati samo ne-null vrijednosti, koristite siguran poziv zajedno sa funkcijom `let`:

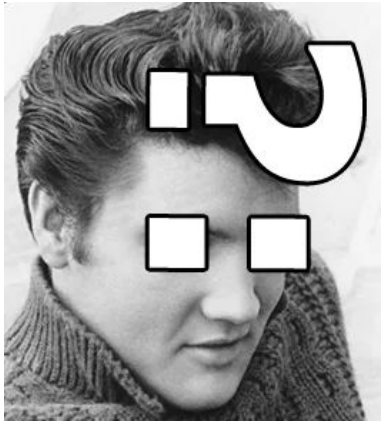
```
employee.department?.head?.name?.let { println(it) }
```

Operator ?:

```
var name: String? = "Kotlin"
val length: Int? = name?.length
print(if (length != null) length else 0)
```

```
var name: String? = "Kotlin"
val length: Int? = name?.length
print(length ?: 0)
```

- Ako izraz s lijeve strane od ?: nije null, ?: operator vraća lijevi izraz, inače vraća izraz sa desne strane.
- Izraz na desnoj strani će se izvršiti samo ako je lijeva strana null.



Null sigurnost

Primjer 2.

Napiši funkciju koja uzima nadimak korisnika kao parametar (koje može biti null) i vraća broj karaktera u nadimku. Ako je nadimak null, vrati poruku "Ne postoji nadimak".

Null sigurnost

```
val notNullText: String = "Definitely not null"
val nullableText1: String? = "Might be null"
val nullableText2: String? = null

fun funny(text: String?) {
     if (text ! = null)
          println(text)
     else
          println("Nothing to print :(")
}

fun funnier(text: String?) {
     val toPrint = text ?: "Nothing to print :("
     println(toPrint)
}
```

Nesiguran poziv

Operator (!!) pretvara bilo koju vrijednost u ne-null tip i izbacuje NPE exception ako je vrijednost null.

```
fun printDepartmentHead(employee: Employee) {
    println(employee.department!!.head!!.name!!)
}
```

Koristiti ovaj operator samo ako ste 100% sigurni da varijabla ne može biti null!

Izbjegavajte korištenje nesigurnih poziva!

Kompaktne funkcije

Kompaktne funkcije

- Kompaktne funkcije ili funkcije sa jednim izrazom čine vaš kod sažetijim i čitljivijim.

```
fun double(x: Int): Int {
    x * 2
}
```

```
fun double(x: Int):Int = x * 2
```

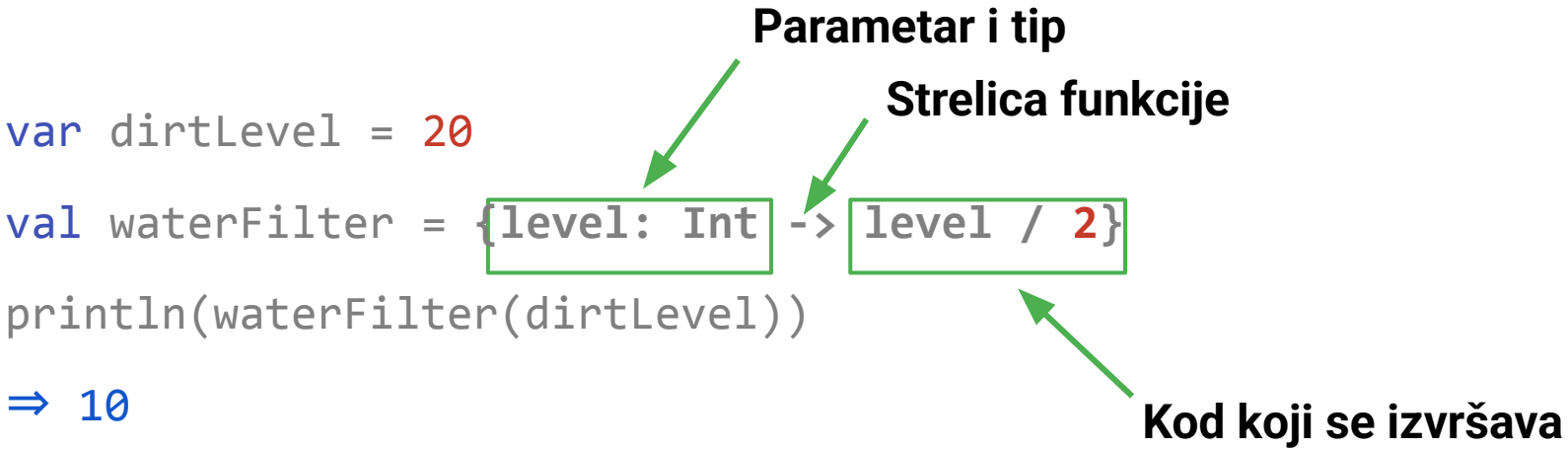
Lambda funkcije i funkcije višeg reda

Kotlin funkcije su first-class objekti

- Kotlin funkcije mogu biti sačuvane u varijablama i strukturama podataka
- Mogu se proslijeđivati kao argumenti drugim funkcijama višeg reda i vraćati iz njih.
- Možete koristiti funkcije višeg reda za kreiranje novih "ugrađenih" funkcija.

Lambda funkcije

Lambda funkcije su funkcije koje nemaju ime, a mogu se zapisati direktno unutar varijabli ili funkcija.



Lambda izrazi

```
val sum: (Int, Int) → Int = { x: Int, y: Int → x + y }
val mul = { x: Int, y: Int → x * y }
```

Prema Kotlinovoj konvenciji, ako je posljednji parametar funkcije funkcija, tada se lambda izraz proslijeđen kao odgovarajući argument može staviti izvan zagrada:

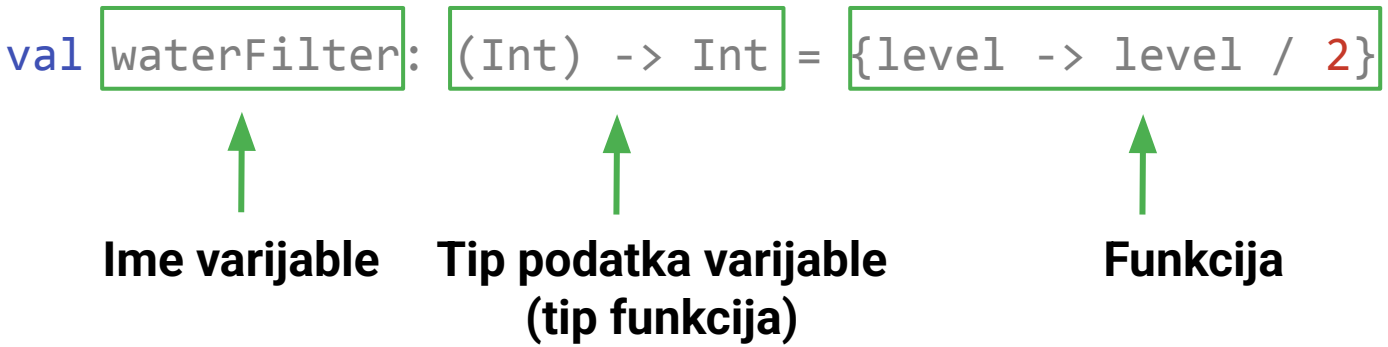
```
val badProduct = items.fold(1, { acc, e → acc * e })

val goodProduct = items.fold(1) { acc, e → acc * e }
```

Ako je lambda jedini argument, zagrade se mogu u potpunosti izostaviti:

```
run({ println("Not Cool") })
run { println("Very Cool") }
```

Sintaksa za tipove funkcija

Sintaksa Kotlina za tipove funkcija usko je povezana sa sintaksom za lambda funkcije. Možete deklarisati promjenljivu koja sadrži funkciju.




```
val waterFilter = {level -> level / 2}
```

Funkcije višeg reda

Funkcije višeg reda primaju funkcije kao parametre ili vraćaju funkciju.

```
fun encodeMsg(msg: String, encode: (String) -> String): String {
    return encode(msg)
}
```

Tijelo koda poziva funkciju koja je proslijeđena kao drugi argument, i prosljeđuje prvi argument toj funkciji

Funkcije višeg reda

Da biste pozvali ovu funkciju, proslijedite joj string i funkciju.

```
val enc1: (String) -> String = { input -> input.toUpperCase() }
println(encodeMsg("abc", enc1))
```

Korišrenje tipa funkcije odvaja njenu implementaciju od njene upotrebe.

Proslijeđivanje reference na funkciju

Koristiti `::` operator da proslijedite referencu funkcije kao argument drugoj funkciji.

```
fun enc2(input:String): String = input.reversed()

encodeMessage("abc", ::enc2)
```

Proslijeđivanje funkcije definirane sa imenom, a ne lambda funkcije

Operator `::` omogućava Kotlinu da zna da prosljeđujete referencu na funkciju kao argument, a ne da pokušavate da pozovete funkciju.

Sintaksa poziva sa posljednjim parametrom

Kotlin preferira da svaki parametar koji prima funkciju bude posljednji parametar.

```
encodeMessage("acronym", { input -> input.toUpperCase() })
```

Možete proslijediti lambda funkciju kao parametar bez stavljanja unutar zagrada.

```
encodeMsg("acronym") { input -> input.toUpperCase() }
```

Korištenje funkcija višeg reda

Mnoge ugrađene funkcije u Kotlinu su definisane koristeći sintaksu poziva sa poslednjim parametrom..

```
inline fun repeat(times: Int, action: (Int) -> Unit)
```

```
repeat(3) {
    println("Hello")
}
```

Funkcije višeg reda

Primjer br. 3

U svijetu se koriste tri glavne temperaturne skale: Celsius, Fahrenheit i Kelvin.

- Celsius -> Fahrenheit: °F = 9/5 (°C) + 32
- Kelvin -> Celsius: °C = K - 273.15

```
fun main() {
    // Dopuni
}


fun ispisTemperature(
    pocMjerenje: Double,
    pocJedinica: String,
    krajJedinica: String,
    formulaKonverzije: (Double) -> Double
) {
    val mjerenje = String.format("%.2f", formulaKonverzije(pocMjerenje)) // dva decimalna mjesta
    println("$pocMjerenje stepeni $pocJedinica je $mjerenje stepeni $krajJedinica.")
}
```

TODO

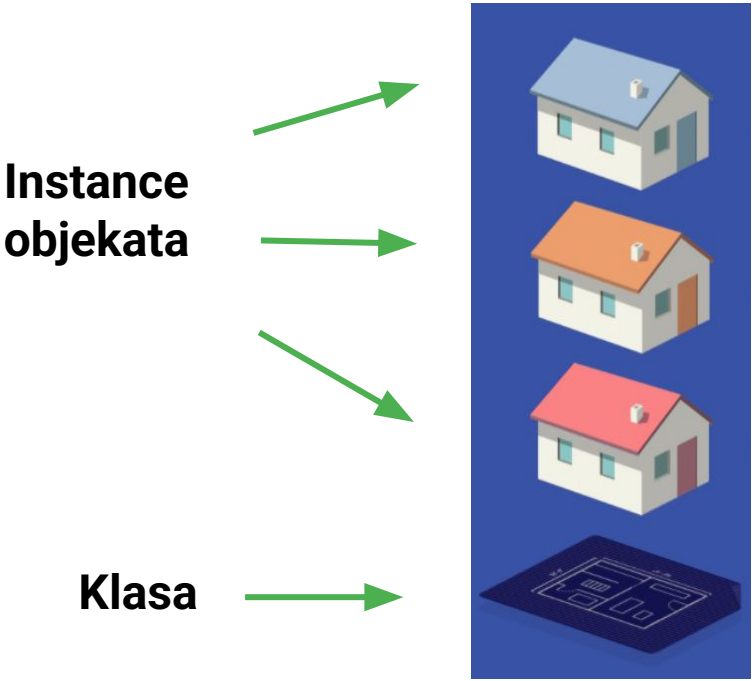
Uvijek će generirati NotImplementedError u vrijeme izvršavanja ako se pozove, navodeći da operacija nije implementirana.

```
// javlja grešku u vrijeme izvršavanja ako se ova funkcija pozove, ali se kompajlira
fun findItemOrNull(id: String): Item? = TODO("Find item $id")

// Neće se kompajlirati
fun findItemOrNull(id: String): Item? = { }
```

Klase

Klase

- Klase su nacrti za objekte.
- Klase definišu metode koje rade sa njihovim instancama objekata.



Klasa vs. instanca objekta

Klasa House

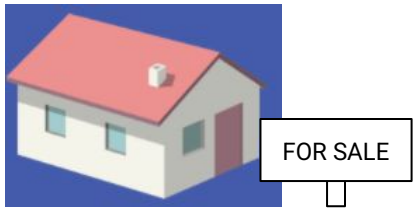
Podaci

- Boja kuće (String)
- Broj prozora (Int)
- Da li je za prodaju (Boolean)

Ponašanje

- `updateColor()`
- `putOnSale()`

Instance objekta



Definiranje i upotreba klase

Definicija klase

```
class House {
  val color: String = "white"
  val numberOfWindows: Int = 2
  val isForSale: Boolean = false

  fun updateColor(newColor: String){...}
  ...
}
```

Kreiranje nove instance objekta

```
val myHouse = House()
println(myHouse)
```

Konstruktori

Kada je konstruktor definisan u zaglavlju klase, može biti:

- Bez parametara

```
class A
```

- Sa parametrima
  - Nisu označeni sa `var` ili `val` → kopija postoji samo u okviru konstruktora

```
class B(x: Int)
```

  - Označeni sa `var` ili `val` → kopija postoji u svim instancama klase

```
class C(val y: Int)
```

Konstruktor primjeri

```
class A
```

```
val aa = A()
```

```
class B(x: Int)
```

```
val bb = B(12)
println(bb.x)
=> compiler error unresolved
reference
```

```
class C(val y: Int)
```

```
val cc = C(42)
println(cc.y)
=> 42
```

Default parametri

Instance klase mogu imati default vrijednosti.

- Koristite default vrijednosti da biste smanjili broj potrebnih konstruktora.
- Default parametri mogu biti kombinovani sa obaveznim parametrima.
- Sažetije je (nema potrebe za više verzija konstruktora).

```
class Box(val length: Int, val width:Int = 20, val height:Int = 40)

val box1 = Box(100, 20, 40)
val box2 = Box(length = 100)
val box3 = Box(length = 100, width = 20, height = 40)
```

Primarni konstruktor

Deklarišite primarni konstruktor u zaglavlju klase.

```
class Circle(i: Int) {
    init {
        ...
    }
}
```

Ovo je isto kao i:

```
class Circle {
    constructor(i: Int) {
        ...
    }
}
```

Init blok

- Sav neophodan inicijalizacioni kod se izvršava u posebnom `init` bloku
- Višestruki `init` blokovi su dozvoljeni
- `init` blokovi postaju tijelo(body) primarnog konstruktora.

Primjer sa init blokom

Korištenje `init` ključne riječi:

```
class Square(val side: Int) {
    init {
        println(side * 2)
    }
}

val s = Square(10)
=> 20
```

Višestruki konstruktori

- Koristite `constructor` ključnu riječ za definiranje sekundarnih konstruktora
- Sekundarni konstruktori moraju:
  - Pozivati primarni konstruktor korištenjem `this` ključne riječi

    ILI
  - Drugi sekundarni konstruktor koji poziva primarni konstruktor
- Tijelo sekundarnog konstruktora nije potrebno

Primjer više konstruktora

```
class Circle(val radius:Double) {
    constructor(name:String) : this(1.0)
    constructor(diameter:Int) : this(diameter / 2.0) {
        println("in diameter constructor")
    }
    init {
        println("Area: ${Math.PI * radius * radius}")
    }
}
val c = Circle(3)
```

Atributi

- Definiranje atributa korištenjem `val` ili `var`
- Pristupite ovim svojstvima koristeći dot . notacija sa imenom atributa
- Postavljanje vrijednosti atributa korištenjem dot . notacije sa imenom atributa (samo ako su deklarirani sa `var`)

Person klasa sa name atributom

```
class Person(var name: String)

fun main() {
    val person = Person("Alex")
    println(person.name)
    person.name = "Joey"
    println(person.name)
}
```

pristupiti sa .<ime_atributa>

postaviti sa .<ime_atributa>

Custom get i set metodi

Ako ne želite default `get/set` ponašanje:

- Override `get()` za atribut
- Override `set()` za atribut ako je definiran kao `var`)

Format:

```
var propertyName: DataType = initialValue
        get() = ...
        set(value) {
            ...
        }
```

Custom get metod

```
class Person(val firstName: String, val lastName:String) {
    val fullName:String
        get() {
            return "$firstName $lastName"
        }
}

val person = Person("John", "Doe")
println(person.fullName)
=> John Doe
```

Custom set metod

```
var fullName:String = ""
    get() = "$firstName $lastName"
    set(value) {
        val components = value.split(" ")
        firstName = components[0]
        lastName = components[1]
        field = value
    }

person.fullName = "Jane Smith"
```

Member funkcije

- Klase mogu sadržavati i funkcije
- Funkcije se deklariraju kao što smo radili na prethodnom predavanju
  - `fun` ključna riječ
  - Mogu imati default ili zahtjevane parametre
  - Specificiraj tip podatka kojeg funkcija vraća (ako ne, onda je `Unit`)

Klase

Primjer br. 3

Zamislite da trebate kreirati aplikaciju za slušanje muzike.

Kreirajte klasu koja može predstavljati strukturu pjesme. Klasa Song mora uključivati ove elemente koda:

- atribute za naziv, izvođača, godinu izdanja i broj pregleda
- atribut koje pokazuje da li je pjesma popularna. Ako je broj pregleda manji od 1000, smatrajte to nepopularnim.
- Metoda koja ispisuje opis pjesme u ovom formatu:

  "[Naziv], u izvedbi [izvođača], objavljen je u [godini izdanja]."

Nasljeđivanje

Nasljeđivanje

- Kotlin ima single-parent klasno nasljeđivanje
- Svaka klasa ima tačno jednu roditeljsku klasu, koja se zove superklasa
- Svaka potklasa nasljeđuje sve članove svoje nadklase uključujući i one koje je naslijedila sama nadklasa

Ako ne želite da budete ograničeni nasljeđivanjem samo jedne klase, možete definirati interfejs jer možete implementirati onoliko njih koliko želite.

Interface-i

- definira ugovor kojeg se moraju pridržavati sve klase koje implementiraju interface
- Može sadržavati potpise metoda i imena atributa
- Može proizaći iz drugih interfejsa

Format: `interface NameOfInterface { interfaceBody }`

Interface primjer

```
interface Shape {
    fun computeArea() : Double
}

class Circle(val radius:Double) : Shape {
    override fun computeArea() = Math.PI * radius * radius
}


val c = Circle(3.0)
println(c.computeArea())
=> 28.274333882308138
```

Extending klase

- Za proširenje klase:
- Kreirajte novu klasu koja koristi postojeću klasu kao svoju jezgru (podklasu)
- Dodajte funkcionalnost klasi bez kreiranja nove (funkcije proširenja)

Kreiranje nove klase

- Kotlin klase po defaultu nisu takve da mogu se od njih kreirati subklase
- Koristite ključnu riječ open da dozvolite podklasu
- Atributi i funkcije se redefiniraju pomoću ključne riječi override

Kotlin klase su po defaultu final

Deklaracija klase

```
class A
```

Pokušaj definiranja podklase

```
class B : A
```

```
=>Error: A is final and cannot be inherited from
```

Korištenje open ključne riječi

Koristiti `open` da se definira klase od koje se mogu nasljeđivati

Deklaracija klase

```
open class C
```

Subklasa od klase C

```
class D : C()
```

Overriding metoda

- Mora se koristiti `open` za atribute i metode koje se trebaju override(u suprotnom se generira greška pri kompajliranju)
- Mora se koristiti `override` kada se radi override atributa i metoda
- Nešto markirano kao `override` može biti override u subklasama (ako nije označeno sa `final`)

Abstract classes

- Klasa se označi kao `abstract`
- Ne može se instancirati, mora imati definiranu podklasu
- Slično sučelju s dodanom mogućnošću pohranjivanja stanja
- Atributi i funkcije sa ključnom riječi `abstract` mora biti override - ponovno definirane
- Može sadržavati i ne-abstraktne atribute i funkcije

Primjer abstract klase

```
abstract class Food {
    abstract val kcal : Int
    abstract val name : String
    fun consume() = println("I'm eating ${name}")
}
class Pizza() : Food() {
    override val kcal = 600
    override val name = "Pizza"
}
fun main() {
    Pizza().consume()    // "I'm eating Pizza"
}
```

Kada koristiti

- Interfejs se koristi kada želite da definirate širok spektar ponašanja koji može biti implementiran na različite načine.
- Klasa se koristi kada je ponašanje specifično za taj tip.
- Interfejsi se koriste za simuliranje višestrukog nasljeđivanja.
- Apstraktna klasa se koristi kada želite da ostavite neke atribute ili metode apstraktnim, da bi ih implementirale podklase.
- Kotlin dozvoljava nasljeđivanje samo jedne klase, ali možete implementirati više interfejsa.

Funkcije proširenja
Extension functions

Funkcije proširenja

Možete dodati funkcije postojećoj klasi (čak i ako ne možete direktno modifikovati tu klasu) koristeći funkcije proširenja.

- Izgledaju kao da ih je implementator dodao
- Ne modifikuju stvarnu klasu
- Ne mogu pristupiti privatnim instancama ili metodama

Format: `fun ClassName.functionName( params ) { body }`

Zašto koristiti funkcije proširenja?

- Dodajte funkcionalnost klasama koje nisu otvorene.
- Dodajte funkcionalnost klasama koje ne posjedujete.
- Odvojite osnovni API od pomoćnih metoda za klase koje posjedujete.

Definišite funkcije proširenja na lako dostupnom mestu, kao što je isti fajl gde se nalazi klasa, ili u funkciji sa jasnim nazivom.

Primjer funkcije proširenja

Dodati `isOdd() Int` klasi:

```
fun Int.isOdd(): Boolean { return this % 2 == 1 }
```

pozivanje `isOdd()` na objektu klase `Int:`

```
3.isOdd()
```

Funkcije proširenja su vrlo korisne u Kotlinu!

Specijalne klase

Data klasa

- Specijalna klasa koja postoji samo da bi čuvala skup podataka.
- Označava se sa `data` ključnom riječi
- Generira get metode za svaki atribut (i set metod za var tipove)
- Generira `toString(),equals(),hashCode(),copy()` metode, i operatore za destrukturiranje

  Format: `data class <NameOfClass>( parameterList )`

Data klasa primjer

Definirajte data klasu:

```
data class Player(val name: String, val score: Int)
```

Korištenje data klase:

```
val firstPlayer = Player("Esmir", 10)
println(firstPlayer)
```

```
=> Player(name=Esmir, score=10)
```

data klase čine kod sažetijim i čitljivijim, što ubrzava razvoj i olakšava rad sa podacima.

Pair i Triple

- `Pair` i `Triple` su predefinisane klase podataka koje čuvaju 2 ili 3 podatka.
- Pristup varijablama sa `.first`, `.second`, `.third` respektivno
- Obično je bolje koristiti `data` klase određenog imena

Pair i Triple primjeri

```
val bookAuthor = Pair("Harry Potter", "J.K. Rowling")
println(bookAuthor)
=> (Harry Potter, J.K. Rowling)

val bookAuthorYear = Triple("Harry Potter", "J.K. Rowling", 1997)
println(bookAuthorYear)
println(bookAuthorYear.third)
=> (Harry Potter, J.K. Rowling, 1997)
    1997
```

Pair to

Posebna `to` varijanta za `Pair` omogućava da izostavite zagrade i tačke (koristi se kao infiksna funkcija).

Omogućava čitljivi kod:

```
val bookAuth1 = "Harry Potter".to("J. K. Rowling")
val bookAuth2 = "Harry Potter" to "J. K. Rowling"
```

```
=> bookAuth1 and bookAuth2 are Pair (Harry Potter, J. K. Rowling)
```

Može se koristiti u kolekcijama kao što su Map i HashMap

```
val map = mapOf(1 to "x", 2 to "y", 3 to "zz")
```

```
=> map of Int to String {1=x, 2=y, 3=zz}
```

Organizacija koda

Jedan file, više entiteta

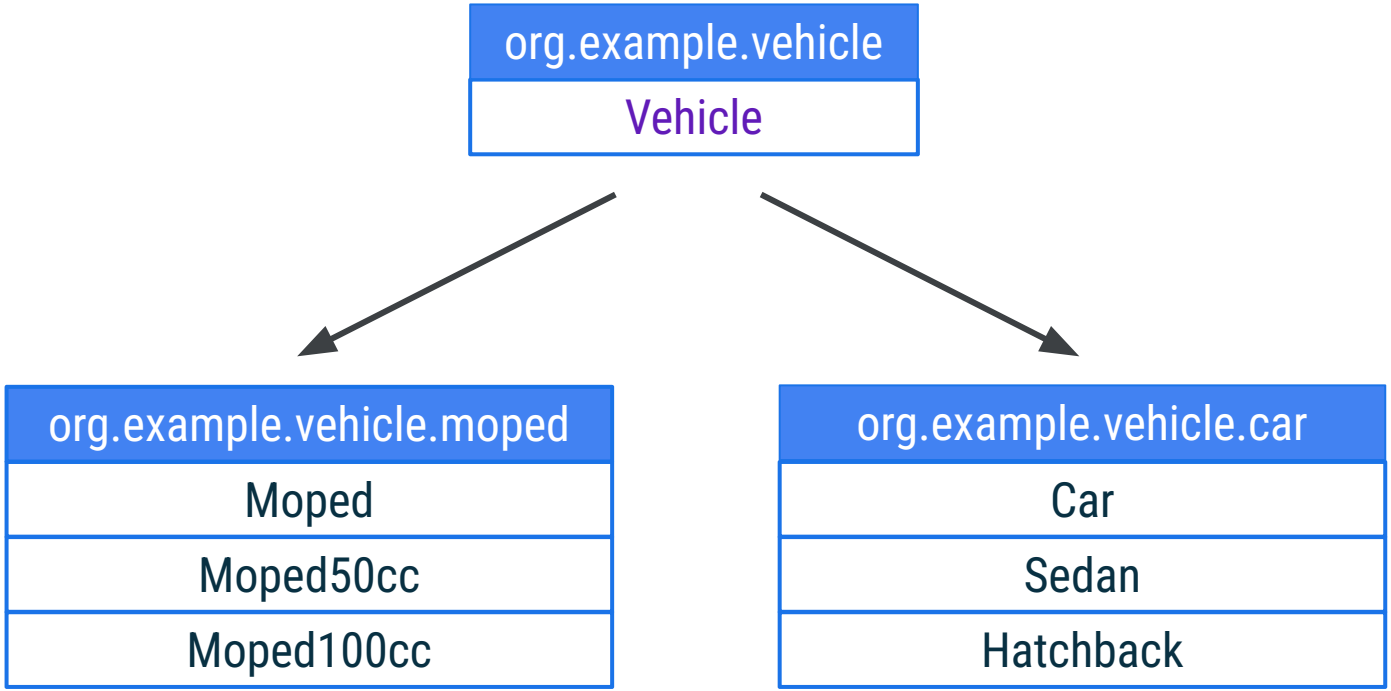
- Kotlin NE nameće pravilo da u jednom fajlu mora biti samo jedna klasa ili interfejs.
- Možete, i trebalo bi, da grupišete povezane strukture u isti fajl.
- Vodite računa o dužini fajla i pretrpanosti.

Paketi

- Paketi omogućavaju logičku organizaciju klasa i funkcija u vašem projektu.
- Imena paketa su obično pisana malim slovima i razdvojena tačkama,
- Deklaracija: Paket se deklariše u prvoj liniji koda (koja nije komentar) koristeći ključnu reč `package`.

```
package org.example.game
```

Primjer hijerarhije klasa



Visibility modifiers

Koristite modifikatore vidljivosti da ograničite koje informacije izlažete.

- `public` znači da je vidljivo van klase. Sve je po defaultu javno, uključujući varijable i metode klase.
- `private` znači da će biti vidljivo samo unutar te klase (ili unutar fajla ako radite sa funkcijama).
- `protected` je isto kao i `private`, ali će biti vidljivo i podklasama.