Kotlin alapú szoftverfejlesztés - 3. előadás

Objektum orientáltság

Öröklés

- Java → alapból minden osztályból le lehet származni
- Kotlin → külön kell engedélyezni, hogy egy osztályból le lehessen származtatni!
- egy osztály csak egy másik osztályból tud leszármazni!
- `open`-ből le lehet belőle származni és példányosítani is lehet
- `abstract`-ból le lehet belőle származni de példányosítani nem lehet

```
abstract class Entity(var x: Double, var y: Double) { //abstract -> tudok belőle példányosítani
    open fun progress() { //open kulcsszó -> felül tudom majd írni

    }
}
class UFO(x: Double, y: Double) : Entity(x, y) { //öröklés, és paraméterek átadása az ősosztály konstruktor paraméterének
    override fun progress() { //kötelező az override-ot kiírni!
        x += Random.nextDouble(from = -5.0, until = 5.0) //Random használata
    }
}
```

Interfész

- függvények, és property-k is lehetnek benne!
    - értéket NEM tartalmazhat az interfész-ben, majd csak az tud értéket is adni neki, aki implementálja
- több interfészt is implementálhat egy osztály

```
interface Renderable {
    val isVisible: Boolean //legyen egy ilyen property majd az osztályban
    fun render(canvas: Canvas) {
    }
}

class UFO(...) : Entity(...), Renderable { ... }
```

Const

- `const val WIDTH = 40.0`
- inline-olásra kerülnek, azaz konkrétan az adott helyekre lesznek "égetve" a kódban

Típusellenőrzés és cast-olás

- egyben megvalósítható a kettő

```
for(entity in entities) {
    if(entity is Renderable && entity.isVisible) {
        entity.render(canvas) //itt már NEM kell cast-olni, mert már
BEBIZONYÍTOTTUK, hogy ez Renderable!
    }
}
```

- kézzel cast-olás

```
(entity as Renderable).render(canvas)
```

Sealed class-ok

- korlátozhatjuk, hogy csak a velük egy fájlban levő osztályok származhatnak le belőle

```
sealed class Response //sealed class -> csak a VELÜK EGY FÁJLBAN levő osztályok
származhatnak le belőle
class Success(val data: String): Response()
class Error(val exception: Exception): Response()

fun getDataFromApi(): Response {
    return try { //try-t is használhatjuk kifejezésként, így a megfelelő ág-gal
tér majd vissza a return
        val data = URL("https://hotlinlang.org").readText()
        Success(data)
    } catch(e: Exception) {
        Error(e)
    }
}
fun main() {
    val result = getDataFromApi()
    val msg = when (result) { //a when így jól együttműködik a sealed class-al
        is Success -> println(result.data
        is Error -> result.exception.message
    }
    println(msg)
}
```

Enumerációk

- ők i osztályok!
- függvényeket a törzstől ;-vel választjuk el

```
enum class Size {
    SMALL, MEDIUM, LARGE
}
enum class MenuItem(val price: Double) {//adható property nekik
    Hamburger(4.65), //ezek 1-1 példány
    Fries(3.5); //ITT kötelező pontosvessző kell, hogy a függvényeket
leválasszuk a törzstől
```

```
    fun purchase() { //függvény is tartozhat
        println("Spending $price...")
    }
}
fun main() {
    Size.SMALL

    val menuItem: MenuItem = MenuItem.Fries
    menuItem.price //visszaadja a 3.5-öt
    menuItem.purchase

}
```

```
enum class MenuItem(val price: Double) {
    Hamburger(4.65),
    Fries(3.5) {
        override fun purchase() { //ha nyitott a függvény, akkor
felüldefiniálhatjuk azt!
            println("Getting some fries...")
        }
    };
    fun purchase() { //kinyithatjuk open-nel a függvényt
        println("Spending $price...")
    }
}
fun main() {
    val menuItem: MenuItem = MenuItem.Fries
    menuItem.price
    menuItem.purchase

}
```

- fontos, saját függvényt nincs értelme írni 1 bizonyos enum elemhez, mert az nem lesz meghívható (mert csak adott enum elemhez van)

Láthatóságok

- *package* → adott package-ben levő látja
- *public* → mindenki látja, ez az alapértelmezett, ezt nem szoktuk kiírni
- *private* → osztályon belül látható
- *protected* → leszármazottban ÉS package-ben is látható
- *internal* → modulon, fordítási egységen levő láthatóság
    - gradle esetén azon belüli
- ha private elsődleges constructor-t akarunk, akkor

```
class SecretValue private constructor(initialValue: Int) { ... }
```

- ha privát propoerty-t akarunk

```
var state: Int = initialValue
    private set //így privát lesz a settere
```

Delegálás

- class delegation, implementation by delegation
- ```
  //így az InstumentedSet összes MutableSet-ből örökölt függvénye
  automatikusan implementálva lesz a paraméterként kapott set-en keresztül
  class InstrumentedSet<E>(private val set: MutableSet<E>): MutableSet<E> by
  set {
      var addCount = 0

      override fun add(element: E): Boolean { //az általunk kívánt
  függvényeken módosíthatunk
          addCount++
          return set.add(element)
      }
      override fun addAll(elements: Collection<E>): Boolean {
          addCount += elements.size
          return set.addAll(elements)
      }
  }
  ```

Kotlin típusossága

- **statikusan típusos** → ha egyszer valami Int lett, már nem lehet String
- **erősen típusos** → ritkán van implicit konverzió (`var x: Double = 50` ezt így magától nem lehet)
    - csak kézzel tudunk átkonvertálni, így véletlenül nem lesz semmi átalakítva
  ```