# Kotlin alapú szoftverfejlesztés - 2. előadás

## Loop-ok

- `while`

```
while(a > b) {
    ...
}
```

- `do-while`

```
do {

} while(a > b)
```

- `for` loop → nem a pontosvesszős C-s verzió van
  - listán végigmenetel

```
val myNumbers = listOf(1, 2, 5, 14, 42)
for(number in myNumbers) {
    ... //lefut minden értékre a listából
}
```

  - adott számig menetel → `range`-el

```
//NÖVEKVŐ range-ek
for(i in 0..10) { //zárt, azaz 10 IS benne van
    println(i)
}
for(i in 0 until 10) { //nyílt, azaz 10 már NINCS benne
    println(i)
}
//csökkenő range
for(i in 10 downTo 0) {
    println(i)
}
```

  - többesével lépkedés

```
for(i in 0..100 step 5) {
    println(i)
}
```

## Osztályok

- osztályok létrehozása

```
class Person //törzs nélkül is hagyható
```

- osztály konstruktor paraméterekkel

```
class Person(name: String, age: Int) //zárójelben a konstruktor paramétere
vannak
fun main() {
    val person = Person("Steve", 33)
    println(person.name)
    println(person.age)
}
```

- osztály property-kkel (field-nél több)

```
class Person(name: String, age: Int) {
    val name: String = name//azaz a neve nem változhat majd
    var age: Int = age
}
```

  - kötelező alapértelmezett értékeket adni a property-knek
- rövidebb verziók

```
class Person(val name: String, var age: Int)
fun main() {
    val person = Person("Steve", 33)
    println(person.name)
    person.age = 34
}
```

  - property mögött keletkezik egy **privát field**,
  - hozzá egy **getter** *val* és *var* esetén, és egy **setter** is *var* esetén
- saját getter/setter → ki kell venni a fejlécből a property-t

```
class Person(val name: String, age: Int) {
    var age: Int = age
        get() {
            return field //ez a field kulcsszó a property mögötti field-et
hivatkozza meg
        }
        set(value) {
            field = value
        }
}
```

  - pl csak nagyobb érték elfogadása

```
set(value) {
    if(value > field) {
        field = value
    }
}
```

- lazy delegate → lusta viselkedés

    - ha null akkor számoljuk csak ki, különben a régebben kiszámolt verziót adjuk vissza
    - így csak először tart sokáig a feldolgozás
    - megoldás → property delegation
    -

```
var pi: Double by lazy { //alapból szálbiztos
    var sum = (1..50_000).sumByDouble { 1.0 / it / it}
    sqrt(sum * 6.0)
}
//szálbiztos mentes mód
var pi: Double by lazy(mode = LazyThreadSafelyMode.NONE) { ... }
```

- Observable delegate → ha változik az adott érték, lefut ez a delegate

```
var name: String by Delegates.observable("Megan") { property, oldValue,
newValue ->
    println("Name changed from $oldValue to $newValue")
}
```

- Vetoable delegate → ha változk az adott érték, akkor fut le, új érték véglegesítését is meg tudjuk akadályozni

```
//új érték véglegesítését meg tudjuk akadályozni
var age: Int by Delegates.vetoable(20) { property, oldValue, newValue ->
    newValue > oldValue
}
```

# Konstruktorok

- minden property-t inicializálni kell, vagy absztrakttá tenni!
- primary constructor → a zárójeles rész, és utána levő property-k

```
class Car(model: String, age: Int) {
    val model: String = model
    var age: Int
    var miles: Double

    init { //inicializálás init blokkal is lehetséges
        this.age = age
        this.miles = miles
    }
}
```

- értékadás fentről lefele sorrendben történik

```
class Car(model: String, age: Int) {
    val model: String = model
    var age: Int = age
    var miles: Double = miles
    val description: String = "$model ($age age, $miles miles)"
    //felette levőknem már van értéke, ezért használhatjuk a többi property-
t az értékadásánál
}
```

- másodlagos konstruktorok létrehozása

```
class Car(model: String, year: Int) {
    val model: String = model
    var year: Int = year
    var miles: Double = miles

    constructor(
        model: String
        year: String
        mileage: String
    ): this(model, year.toInt()) { //secondary constructor első sora
        this.miles = mileage.toDouble()
    }

    constructor(data: Array<String>) : this(
        model = data[1]
        year = data[3]
        mileage = data[7]
    )
}
```

  - a secondary constructor-nak át kell hívnia a primary constructor-ra
- primary constructor nélküli megoldás

```
class Car {
    val model: String
    val year: Int
    var miles: Double = 0.0
    val age: Int
    constructor(model: String, year: Int) {
        this.model = model
        this.year = year
        age = getCurrentYear() - year
    }
    constructor(model: String, year: String, mileage: String) {
        this.model = model
        this.year = year.toInt()
        this.miles = mileage.toDouble()
        age = getCurrentYear() - this.year
    }
}
```

  - ilyenkor nem tudunk property-kből property-t létrehozi → max duplikálással tudjuk megoldani

# Data class

- data class létrehozása

```
data class Person(val name: String, var age: Int)
```

  - automatikusan legenerálja a Person osztályhoz a következőket:
    - **toString, hashCode, equals**
    - újrafordításkor újragenerálódik
  - destructuring declarations

```
data class Person(val name: String, var age: Int)
fun main() {
    val emma = Person("Emma", 19)
    val (name, age) = emma //destructuring declerations
    //val name = emma.name
    //val age = emma.age
    //fontos, hogy nem neveket, hanem SORRENDET figyel
    val (age, name) = emma //itt pont fordítva lenne!!!
}
```

  - copy függvény

```
data class Person(val name: String, var age: Int)
fun main() {
    val emma = Person("Emma", 19)
    val olderEmma = emma.copy(age = 21) //copy függvény
}
```

    - elnevetett paraméterrel megadható az a néhány paraméter, amin változtatni akarunk
    - a többi lemásolásra kerül
    - ***val***-lal is működik, azzal érdemes használni!

# Objektum orientált megközelítés támogatása

- **Singleton osztály** = **object**

```
object Logger { //mivel objektum, nem lehet belőle példányt létrehozni
    var isEnable = true

    fun log(message: String) {
        if(isEnable) {
            println(message)
        }
    }
}
fun main() {
    val logger: Logger = Logger //hozzárendelhetjük egy val-hoz
    Logger.log("hello") //így is elérhető
    Logger.isEnable = false
    Logger.log("world") //ez már nem kerül kiírásra
}
```

- Statikus dolgok pótlása

```
class Document {
    object Counter { //egy lehet belőle => olyan mint valami static
attribútum
        var count = 0
    }
    companion object { //ha companion, mivel 1 lehet belőle, nem kell neki
nevet adni
        var count2 = 0
    }

    val id = Counter.count++
    val id2 = count2++ //már így is elérhetjük
}
fun main() {
    repeat(5) {
        Document()
    }
    println(Document.Counter.count) //így lehet elérni, mert a Document-en
belül van a Counter
    println(Document.count2) //ha companion is előtte van, hivatkozhatunk rá
így is
}
```

  - egy osztályon belül több **object** is lehet
  - egy osztályon belül egyetlen egy object lehet **companion**
    - ennek nem kötelező nevet adni → statikus blokként fog viselkedni
    - ilyenkor Companion nevet kap alapértelmezett módon (*nem fontos*)
- Egymásba ágyazott osztályok

```
class Outer {
    var outerValue = 0
    class Inner { //sima class
        init {
            println(outerValue) //ez NEM érhető el
        }
    }
    inner class Inner2 { //az inner kulcsszóval thetjük ezt elérhetővé
        init {
            println(outerValue) //így már ELÉRHETŐ
        }
    }
}
```

  - **belső osztályból** kotlinban nem érhető el **külső osztály**!