

Android programozás

Kotlin nyelvi alapok - 1.

Ekler Péter

peter.ekler@aut.bme.hu

Ütemezés

- Programozási alapismeretek
- Környezet bemutatása
- Projekt felépítése, kód szervezés
- Elnevezési konvenciók, szerkesztési konvenciók
- A Kotlin nyelv kialakulása, hasonlóságok és különbségek a Java nyelvhez képest
- A Kotlin nyelv szintaxisa
- Konstansok, változók, elágazások, függvények, stb.
- Vezérlési struktúrák
- Kivételkezelés
- Osztályok, leszármaztatás
- Objektum, enumerációk, konstruktor típusok
- Egyszerű alkalmazások fejlesztése

Ütemezés

- Interfészek, abstract osztály
- Adatstruktúrák, listák kezelése
- Típusok jelentése, típus konverzió
- Nullable típusok
- Szöveges adatok kezelése
- Függvények szerepe és lehetőségei
- Lambdák
- Extension nyelvi elem
- Függvény paraméterek
- Operátorok felüldefiniálása
- Komplex vezérlési struktúrák
- Szálkezelés
- Aszinkron nyelvi elemek
- További példák

Tartalom

- Fejlesztőkörnyezet bemutatása
- Első projekt létrehozása
- Programozási alapok
- Kotlin nyelvi alapok
- Konstansok, változók
- Tömbök

Kurzus példái:

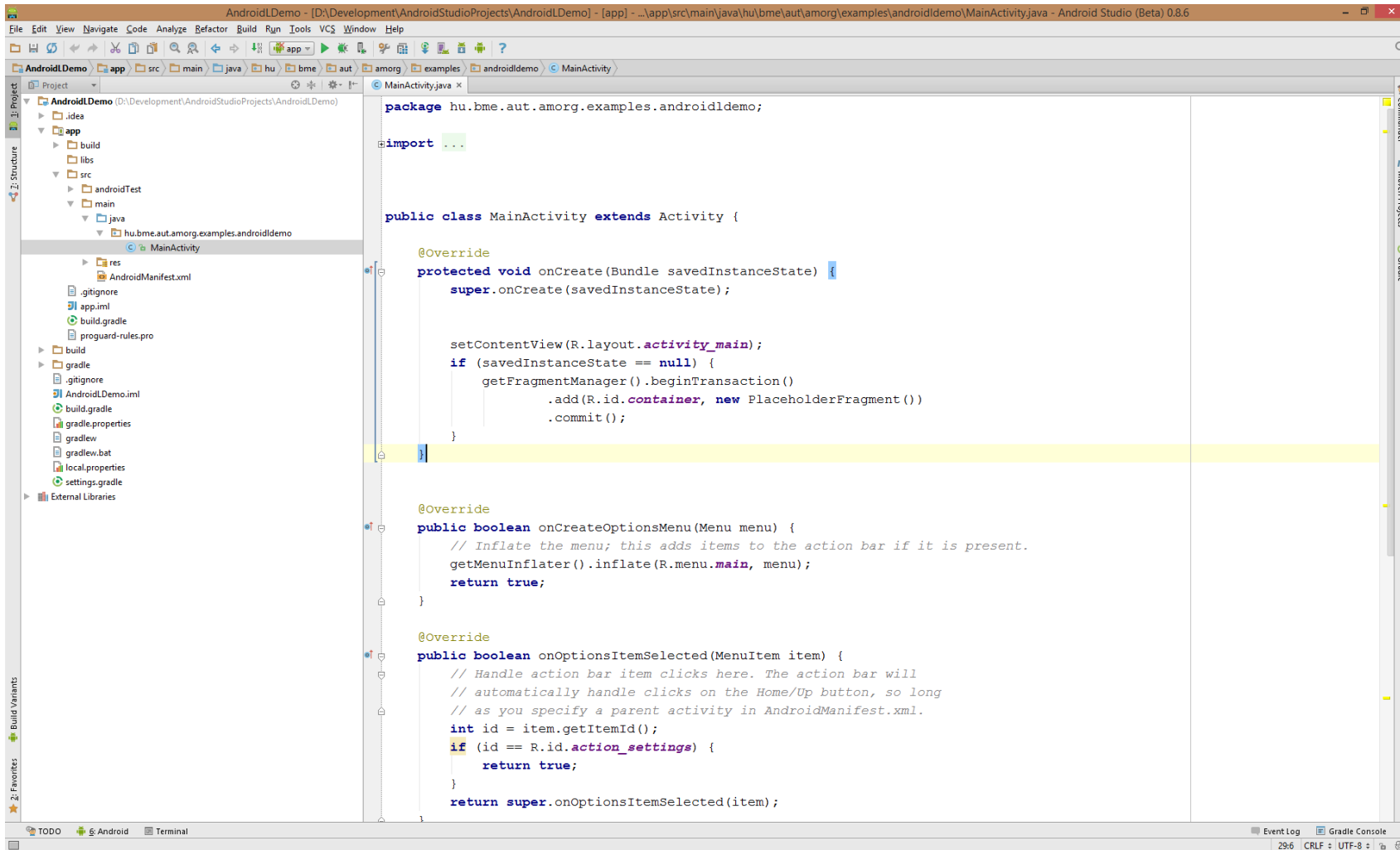
[https://github.com/peekler/
AndroidProgrammingBasics](https://github.com/peekler/AndroidProgrammingBasics)

Android fejlesztőkörnyezet

Android elterjedtsége



Fejlesztő eszköz: Andorid Studio

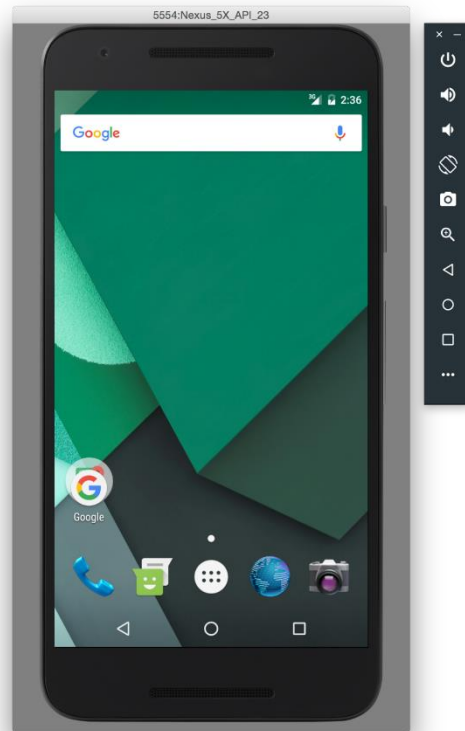


Telepítési útmutató:

<https://bit.ly/2V4eXEB>

Emulátor

- Teljes operációs rendszer emulálása (lassú)
 - Beépített alkalmazások elérhetők
 - Ctrl+F11 (screen orientáció állítás)
- Alternatíva: Genymotion emulátor (<https://www.genymotion.com/>)



Emulátor elérése konzolról

- Csatlakoztatott emulátorok/eszközök listázása:
 - adb devices
- Shell elérése
 - adb shell
- Csatlakozás telneten keresztül:
 - Indítsunk telnet klienst
 - o localhost 5554
- SMS küldése:
 - sms send <küldő száma> <üzenet>
- Hanghívás
 - gsm call <hívó száma>

Debugolás folyamata

- On-device debug teljes mértékben támogatott
 - Megfelelő USB driver szükséges!
 - Készüléken engedélyezni kell az USB debugolást
- Minden alkalmazás önálló process-ként fut
- Minden ilyen process saját virtuális gépet (VM) futtat
- Minden VM egy egyedi portot nyit meg, melyre a debugger rácsatlakozhat (8600, 8601, stb.)
- Létezik egy úgynevezett „base port” is (8700), mely minden VM portot figyel és erre csatlakozva az összes VM-et debugolhatjuk

Első Android projekt – Felület létrehozása

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/btnDemo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Demo"/>

    <TextView
        android:id="@+id/tvHello"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

</LinearLayout>
```

Egyedi ID



Első Android projekt – Kotlin kód

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        btnDemo.setOnClickListener {  
            tvHello.text = "Demo"  
        }  
    }  
}
```

Felület beállítása

Egyedi ID-k használata és
eseménykezelő beállítása



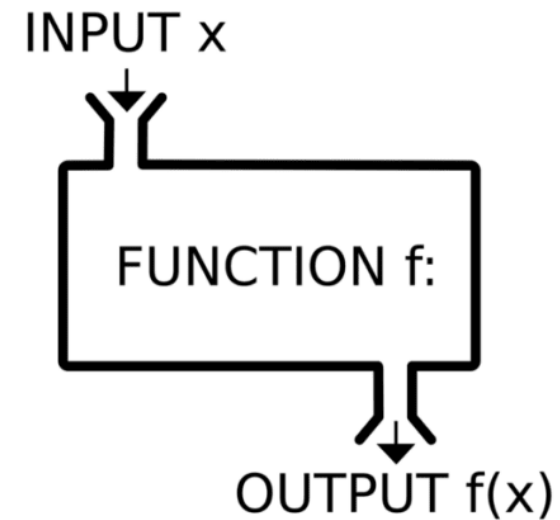
Érdekesség: Függvény mint paraméter

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        btnDemo.setOnClickListener(::demoClick)  
    }  
  
    fun demoClick(view: View) {  
        tvHello.text = "Demo"  
    }  
}
```

Programozási alapok

Programozás alap elemei

- Konstansok
- Változók
- Típusok
- Speciális értékek: null
- Értékek
- Függvények
- Objektum Orientáltság
 - Osztályok
 - Objektumok
 - Leszármaztatás
 - ...



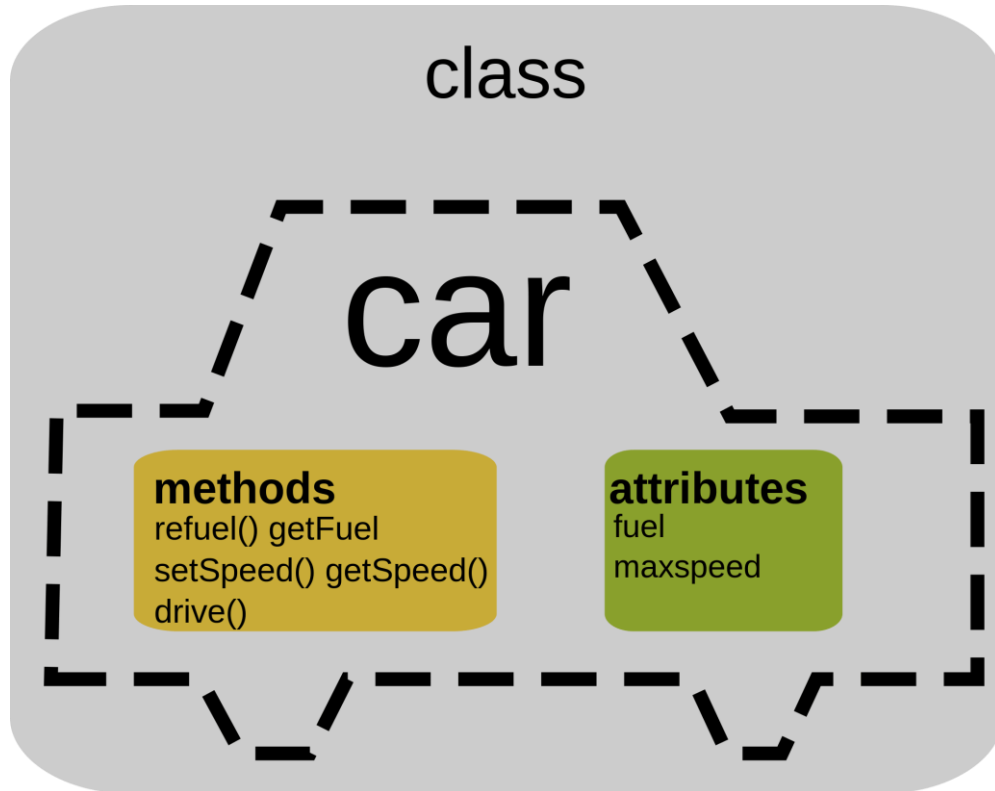
Források:

<https://compscihelp.com/java/1-1.php>

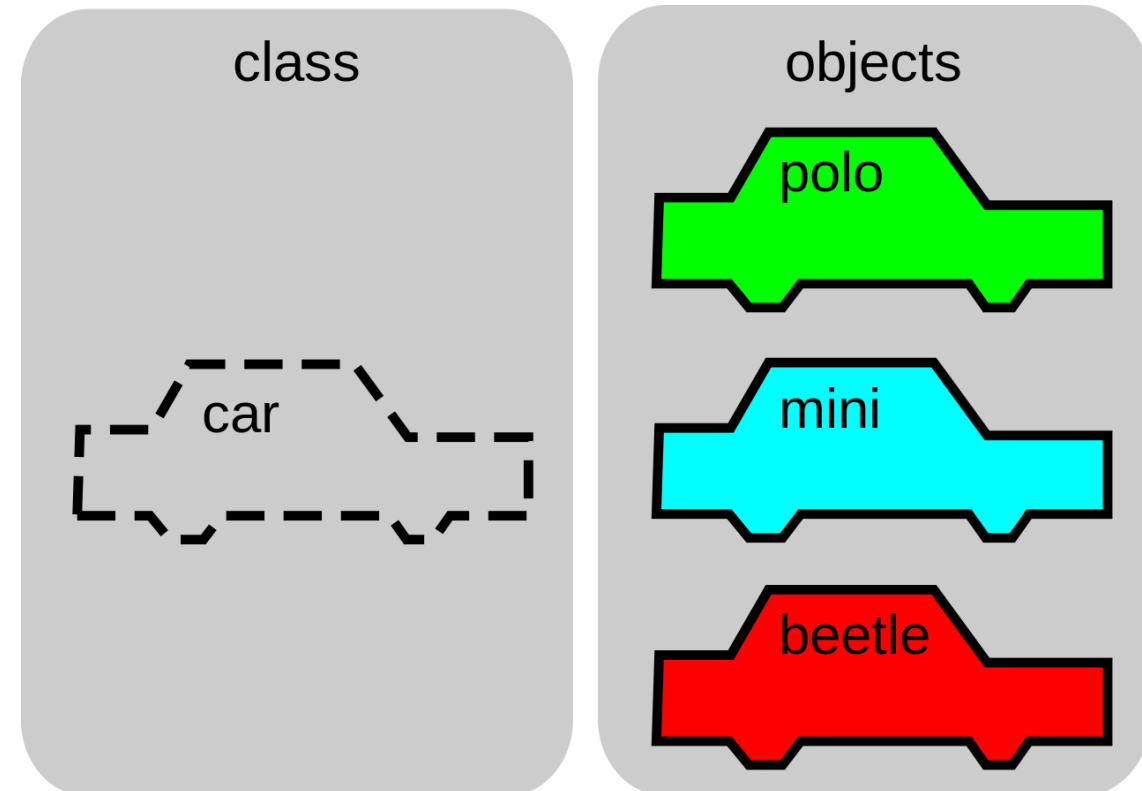
<https://dev.to/navi/why-functional-programming-matters-2o95>

Osztályok, objektumok

- Osztály:



- Objektum:



Források:

<https://www.miltonmarketing.com/coding/programming-concepts/object-oriented-programming-oop/>

https://commons.wikimedia.org/wiki/File:CPT-OOP-objects_and_classes.svg

Kotlin nyelvi alapok

Forrás: <https://kotlinlang.org/docs/reference/>



Mire utal a Kotlin név?

- A. Semmire, csak egy kitalált szó
- B. Egy lengyel falu nevére
- C. Egy sziget nevére
- D. Key Object Tool Language INsight rövidítése

Mire utal a Kotlin név?

- A. Semmire, csak egy kitalált szó
- B. Egy lengyel falu nevére
- C. Egy sziget nevére
- D. Key Object Tool Language INsight rövidítése

Kotlin története

- 2011-ben jelent meg először
- JetBrains gondozásában
- Nyílt forráskódú: 2012
- v1.0 verzió: 2016
- 2017-es Google I/O: hivatalos támogatás Androidra
- Statikusan típusos
- Objektum orientáltság mellett a funkcionális programozást is támogatja

A Kotlin főbb jellemzői

- JVM (Java Virtual Machine) byte kódra (vagy akár JavaScriptre is) fordul
- Meglévő Java API-k, keretrendszerek és könyvtárak használhatók
- Automatikus konverzió Java-ról Kotlinra
- Null-safety
 - Vége a NullPointerException korszaknak
- Kód review továbbra is egyszerű
 - A nyelv alapos ismerete nélkül is olvasható a kód

Miért érdemes Kotlinba programozni?

40%-al kevesebb
kód mint Java-ban

Olvasható kód

Többféle lehetőség:
Támogatja az objektum
orientált és a
funkcionális
programozást egyaránt

Kevesebb futás idejű
NullPointerException

Nincs
boilerplate kód

Írhatunk Kotlin
kódot meglévő
Java
projektekbe is

Kedvenc Java
osztálykönyvtár
ak továbbra is
használhatók

Kód hosszúság Java7 vs Java8 vs Kotlin - *CaesarCipherDecoder*

```
class CaesarCipherKotlinDecoder(val firstLetterInAbc: Char, val lastLetterInAbc: Char) : CeaserDecoder {  
    val alphabet: CharArray = CharArray(lastLetterInAbc - firstLetterInAbc + 1) { it -> it.plus(firstLetterInAbc.toInt()).toChar() }  
    override fun decode(text: String, n: Int): String {  
        return String(text.toCharArray().map { alphabet[calculateDecodedPosition(it, n)] }.toCharArray())  
    }  
    override fun calculateDecodedPosition(character: Char, n: Int): Int {  
        val characterIndex = alphabet.indexOf(character);  
        var decodedIndex = characterIndex - n  
        if (decodedIndex < 0) decodedIndex = alphabet.size + decodedIndex  
        return decodedIndex;  
    }  
}
```

Kotlin: 12 lines

```
public class CaesarCipherJavaDecoder implements CeaserDecoder {  
    private char firstLetterInAbc;  
    private char lastLetterInAbc;  
    private List<Character> alphabet = new ArrayList<Character>();  
    public CaesarCipherJavaDecoder(char firstLetterInAbc, char lastLetterInAbc) {  
        this.firstLetterInAbc = firstLetterInAbc;  
        this.lastLetterInAbc = lastLetterInAbc;  
        generateAlphabet(firstLetterInAbc, lastLetterInAbc);  
    }  
    private void generateAlphabet(char firstLetterInAbc, char lastLetterInAbc) {  
        int size = lastLetterInAbc - firstLetterInAbc + 1;  
        for (Integer i = 0; i < size; i++) {  
            char newLetter = (char) (i + (int) firstLetterInAbc);  
            alphabet.add(newLetter);  
        }  
    }  
    @Override  
    public String decode(String s, Integer n) {  
        char[] chars = s.toCharArray();  
        char[] decodedChars = new char[chars.length];  
        for (int i = 0; i < decodedChars.length; i++) {  
            int decodedPosition = calculateDecodedPosition(chars[i], n);  
            decodedChars[i] = alphabet.get(decodedPosition);  
        }  
        return String.valueOf(decodedChars);  
    }  
    @Override  
    public int calculateDecodedPosition(char character, int n) {  
        int characterIndexInAbc = character - firstLetterInAbc;  
        int decodedIndex = characterIndexInAbc - n;  
        if (decodedIndex < 0) decodedIndex = alphabet.size() + decodedIndex;  
        return decodedIndex;  
    }  
}
```

Java7: 33 lines

```
public class CaesarCipherJava8Decoder implements CeaserDecoder {  
    private char firstLetterInAbc;  
    private char lastLetterInAbc;  
    private List<Character> alphabet;  
    public CaesarCipherJava8Decoder(char firstLetterInAbc, char lastLetterInAbc) {  
        this.firstLetterInAbc = firstLetterInAbc;  
        this.lastLetterInAbc = lastLetterInAbc;  
        generateAlphabet(firstLetterInAbc, lastLetterInAbc);  
    }  
    private void generateAlphabet(char firstLetterInAbc, char lastLetterInAbc) {  
        Stream<Character> alphabetStream = IntStream.range(0, lastLetterInAbc - firstLetterInAbc + 1).mapToObj(i -> (char) (i + (int) firstLetterInAbc));  
        alphabet = alphabetStream.collect(Collectors.toList());  
    }  
    @Override  
    public String decode(String text, Integer n) {  
        Stream<Character> s = text.codePoints().mapToObj(c -> alphabet.get(calculateDecodedPosition((char)c, n)));  
        return s.map(String::valueOf).collect(Collectors.joining());  
    }  
    @Override  
    public int calculateDecodedPosition(char character, int n) {  
        int characterIndexInAbc = character - firstLetterInAbc;  
        int decodedIndex = characterIndexInAbc - n;  
        if (decodedIndex < 0) decodedIndex = alphabet.size() + decodedIndex;  
        return decodedIndex;  
    }  
}
```

Java8: 24 lines

Kotlin kód szerkesztő

- Teljes Kotlin támogatás: AndroidStudio, IntelliJ és IntelliJ Community Edition
- REPL (Read Eval Print Loop)
 - Kotlin fordító/futtató (interpreter)
 - Megnyitás pl. Android Studio-ból: Tools -> Kotlin -> Kotlin REPL
 - Futtatás: CTRL + Enter
- Java->Kotlin automatikus átalakítás

Változók és konstansok

- Változtatható érték, deklaráció kulcsszava: **var**

```
var a: Int = 10  
a = 20
```

Nincs szükség „;”-re a sor végén

```
var a: Int = 10; a = 10
```

„;” használható sorok belüli elválasztásra

- Nem változtatható érték, deklaráció kulcsszava: **val**

```
val a: Int = 10  
a = 20
```

Fordítási hiba: „*Val cannot be reassigned*”

Változók – Nem kötelező a típus megadása

```
var a = 10.1
```

A típust a fordító a kontextus alapján automatikusan kitalálja

Típusok

- Minden valójában objektum -> hívhatók függvények és lekérhetőek tulajdonságok a változókon

```
1.toLong()
```

- Némelyik típus (számok, karakterek, boolean) futási időbe primitív típusként vannak reprezentálva meg, de osztályokként látszanak

Szám típusok

Típus	Bit méret
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

A karakterek nem tartoznak a szám típusok közé
(Java-val ellentétben)

Szám kifejezések

Típus	Kifejezés
Byte, Short, Int, Long	3, 1_200
Long	3L, 1_200L
Hexadecimal (Lehet :Byte, Short, Int, Long)	0xAF, 0XAF, 0xFF_EC_DE_5E
Binary (Lehet: Byte, Short, Int, Long)	0b0101, 0B0101, 0b01_01
Double	1.23, 1.23e10, 1.2_66
Float	1.2F, 1.2f, 1.2_66f

Kasztolás (típus váltás)

```
val b: Byte = 1
```

```
val c: Int = b
```

Fordítási hiba: *Type mismatch: inferred type is Byte but Int was expected*

- Implicit kasztolás nem engedélyezett (mivel tipikus hibaforrás szokott lenni)
 - Például: integer változóhoz nem lehet byte értéket hozzárendelni
- Use explicit casting:

```
val c: Int = b.toInt()
```

Karakterek

```
var someCharacter: Char = 'a'
```

Karakter jelölés: 'a'

Boolean típus

```
var someBoolean: Boolean = true  
var otherBoolean = false
```

Két lehetséges érték: **true**, **false**

```
someBoolean && otherBoolean || someBoolean && !otherBoolean
```

Beépített boolean műveletek

Tömbök

```
public class Array<T> {  
    /**  
     * Creates a new array with the specified [size], where each element is calculated by  
     * calling the specified  
     * [init] function. The [init] function returns an array element given its index.  
     */  
    public inline constructor(size: Int, init: (Int) -> T)  
    ...  
}
```

- Típusos tömbök

Tömb létrehozás (utility function)

```
var myIntArray = intArrayOf(1, 2, 3)  
var myIntArray2 = Array<Int>(3, {it * 1})  
  
println(Arrays.toString(myIntArray))  
println(Arrays.toString(myIntArray2))  
println(myIntArray[2])  
[1, 2, 3]  
[0, 1, 2]  
3
```

Tömb létrehozás az Array osztály konstruktorával

it: az aktuális elem indexe

Hozzáférés az elemekhez a [] operátorral lehetséges

Tömbök megjelenítése a Java Arrays.toString függvénnyel

- Vegyes típusokat tartalmazó tömb

```
var mixedArray = arrayOf(1, "something")  
println(Arrays.toString(mixedArray))  
[1, something]
```

String (szöveg) típus

A szövegek a String típussal (osztály) kezelhetők

```
var someString: String = String(charArrayOf('m', 'y', ' ', 's', 't', 'r', 'i', 'n', 'g'))
```

Minden String valójában karakterek tömbje

```
var someOtherString = "my string"
```

String készítés egyszerűen a "..."-el lehetséges

String minták (template)

- A String template-k \$ jellel kezdődnek
- Kirétekelésre kerülnek és az értékek egy nagy stringé konkaténálódnak
- A templatek tartalmazhatnak:

Egyszerű változókat

```
val nrOfProgrammers = 10  
print("There are $nrOfProgrammers in the room")
```

There are 10 in the room

Komplex kifejezéseket

```
val nrOfProgrammers = 10  
val nrOfManagers = 12  
print("There are ${nrOfProgrammers + nrOfManagers} man in the room")
```

There are 22 man in the room

String Literal Típusok

Escapelt stringek

- Tartalmazhatnak escape karaktereket: `\t`, `\b`, `\n`, `\r`, `\'`, `\"`, `\\` és `\$`
- Elválasztó: idézőjel (`"`)
- Például:

```
var escapedString = "Hello!\nThis is me!"
```

- Tartalmazhat template-eket

Nyers stringek

- Tartalmazhat újsor és más nem-escapelt karaktereket
- Elválasztó: (`"""`)

Pl:

```
var rawString = """Hello!
    |This is me!
    """.trimMargin()
```

- A `trimMargin` használható a kezdő üres spacek és `|` jel előtti spacek eltávolítására, valamint az első és az utolsó üres sor törlésére

```
print(rawString)
Hello!
This is me!
```

- Tartalmazhat template-eket

== operátor

```
var s1 = "hello"  
var s2 = "hello"  
println(s1 == s2) //eredmény: true
```

A “==” operátor használható bármilyen objektum összehasonlítására. Valójában egy .equals() függvényt fog hívni ha a == mindkét oldalán egy nem null értékű objektum áll.

Gyakoroljunk – tömbök, listák

- Rendezés
- Véletlen sorrend
- Műveletek tömbökkel
- Keresés
- Egyéb műveletek

Köszönöm a figyelmet!



Ekler Péter
peter.ekler@aut.bme.hu

Android programozás

Kotlin nyelvi alapok - 2.

Ekler Péter

peter.ekler@aut.bme.hu

Emlékeztető

Emlékeztető

- Fejlesztőkörnyezet bemutatása
- Első projekt létrehozása
- Programozási alapok
- Kotlin nyelvi alapok
- Konstansok, változók
- Tömbök

Konstansok, változók (val vs. var)

- Egyszeri értékadás – „val”

```
val score: Int = 1 // azonnali értékadás
val idx = 2      // típus elhagyható
val age: Int     // típus szükséges ha nincs azonnali értékadás
age = 3          // későbbi értékadás
```

- Változók (megváltoztatható) – „var”

```
var score = 0 // típus elhagyható
score += 1
```

- String sablonok

```
var score = 1
val scoreText = "$score pont"

score = 2
// egyszerű kifejezések string-ek esetében:
val newScoreText = "${scoreText.replace("pont", "volt, most ")} $score"
```

Függvények

- Függvény szintaxis

```
fun add(a: Int, b: Int): Int {  
    return a + b  
}
```

Tartalom

- Null kezelés
- Vezérlési struktúrák
- Függvények
- Osztályok
 - Elsődleges konstruktor
 - Másodlagos konstruktor-ok
- Objektumok létrehozása
- Gyakorlás

Kurzus példái:

<https://github.com/peekler/AndroidProgrammingBasics>

Null érték Kotlinban, Nullable típusok

Null értékek kezelése, Null típus

- Alapértelmezetten egy változó értéke ne lehet null

```
var a: Int = null
```

Fordítási hiba: *Null can not be a value of a non-null type Int*

- A '?' operátor, mint új „típus” jelzi, ha egy változó lehet null

```
var a: Int? = null
```

- További példák

- Lista, melyben lehetnek null elemek

```
var x: List<String?> = listOf(null, null, null)
```

- Lista, mely lehet null

```
var x: List<String>? = null
```

- Lista, mely lehet null és az elemei is lehetnek null-ok

```
var x: List<String?>? = null  
x = listOf(null, null, null)
```

Null ellenőrző operátor: ,?'

- Mi történik, ha egy metódust hívunk egy nullable elemen?

```
var someNullableValue: Int? = 7  
someNullableValue.toByteArray()
```

Fordítási hiba: *Only safe (?) or non-null asserted (!!)* calls are allowed on a nullable receiver of type Int?

Használjuk a null ellenőrző operátort

```
var someNullableValue: Int? = 7  
someNullableValue?.toByteArray()
```

Jelentése: "ha nem null"

`someNullableValue?.toByteArray()` függvény nem hívódik meg, ha a `someNullableValue` értéke null

Null jelző operátor

- Mi történik ha egy metódust hívunk egy null értéken

```
var someNullableValue: Int? = 7  
someNullableValue.toByteArray()
```

Fordítási hiba: *Only safe (?) or non-null asserted (!!)* calls are allowed on a nullable receiver of type Int?

Használjuk a **null jelző operátort**

```
var someNullableValue: Int? = 7  
someNullableValue!!.toByteArray()
```

Kiejtés: “double bang”

Ha egy függényt hívunk egy null objektumon a “double bang” operátorral, akkor KotlinNullPointerException kivétel dobódik

Az Elvis operátor

- `?:` “Elvis” operator (Elvis Presley haj stílusára utal...)
- Ha a kifejezés bal oldala nem null akkor akkor azt használja, egyébként a jobb oldalon levő kifejezés kerül kiértékelésre

```
fun foo(car: Car) {  
    var extras: MutableList<String> = car.defaultExtras ?: mutableListOf()  
    //some code  
}
```

```
fun bar(car: Car) {  
    var manufacturer: String = car.manufacturer ?: throw IllegalArgumentException(  
        "Manufacturer cannot be null")  
    //some code  
}
```

Kvíz

Mi a kimenete a "bakeACake" függvény hívásnak, ha sugar = 0 és eggs = 1 értékekkel hívjuk.

```
fun bakeACake(sugar: Int?, eggs: Int) {  
    sugar ?: throw Throwable("Hey, sugar is needed!!!")  
}
```

- 1) Semmi, mivel nem fordul a kód
- 2) A kód lefut hiba nélkül és nem dobódik kivétel
- 3) Kivétel fog dobódni

Kvíz

Mi a kimenete a "bakeACake" függvény hívásnak, ha sugar = 0 és eggs = 1 értékekkel hívjuk.

```
fun bakeACake(sugar: Int?, eggs: Int) {  
    sugar ?: throw Throwable("Hey, sugar is needed!!!")  
}
```

- 1) Semmi, mivel nem fordul a kód
- 2) A kód lefut hiba nélkül és nem dobódik kivétel
- 3) Kivétel fog dobódni

Vezérlési szerkezetek

If – Klasszikus módon

- Java-hoz hasonlóan

```
val nrOfPizza = 4
var isEnoughPizza = false
if (nrOfPizza > 10) isEnoughPizza = true
if (isEnoughPizza)
    print("We have enough pizza!")
else
    print("We don't have enough pizza!")
```

If – kifejezésként

- Kotlinban szinte mindennek van értéke (while és for ciklusoknak nincs)
- Az “if” eredményét lehet értékként használni

```
val nrOfPizza = 4
var isEnoughPizza = false
if (nrOfPizza > 10) isEnoughPizza = true
if (isEnoughPizza)
    print("We have enough pizza!")
else
    print("We don't have enough pizza!")
```

```
val nrOfPizza = 4
var isEnoughPizza = if (nrOfPizza > 10) true else false
if (isEnoughPizza)
    print("We have enough pizza!")
else
    print("We don't have enough pizza!")
```

```
val nrOfPizza = 4
print("We ${if (nrOfPizza > 10) "have" else "don't have"} enough pizza ")
```

If – láncolatok (range) feltételekben

```
val nrOfProgrammers = 10

if (nrOfProgrammers in 1..5)
    print("Too few!")
else
    print("Enough")

„Enough”
```


A **when** elágazás

- A **when** elágazás automatikusan megáll

```
val nrOfProgrammers = 10
when (nrOfProgrammers) {
    0 -> print("No meetup")
    in 10..50 -> print("Ok")
    100 -> print ("Full")
    else -> print("What??")
}
Ok
```

A **when** utasítás mint kifejezés

- A 'when' utasításnak is van értéke: az utoljára felvett kifejezés értéke

```
fun getFortuneCookie(birthday: Int): String{
    val sentences = arrayOf("You will have a great day!",
        "Things will go well for you today.",
        "Enjoy a wonderful day of success.",
        "Take it easy and enjoy life!")

    return when (birthday){
        in 1993..1998 -> "Hey!!!"
        1990,1992 -> "Not lucky"
        else -> {
            var fortuneIndex = birthday.rem(sentences.size)
            sentences[fortuneIndex]
        }
    }
}
```

A **when** utasítás –**if-else** láncok kiváltására

```
fun whatShouldIDoToday(mood: String, weather: String, temperature: Int): String {  
    return when {  
        mood == "sad" && weather == "rainy" -> "Watch a good film"  
        weather == "sunny" && temperature > 20 -> "Go and play in the garden!"  
        temperature < 5 -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}
```

Argumentum nélkül is használható

Ciklusok

- Egyszerű ciklus

```
var myArray = arrayOf(10, 20, 30)
for (element in myArray) {
    print("$element ")
}
10 20 30
```

- Ciklus indexekkel és elemekkel

```
var myArray = arrayOf(10, 20, 30)
for ((index, element) in myArray.withIndex()) {
    println("$element at position $index")
}
10 at position 0
20 at position 1
30 at position 2
```

Ciklusok range-eken

```
for (i in 'd'..'k') print(i)  
defghijk
```

```
for (i in 0 .. 10) print(i)  
0 1 2 3 4 5 6 7 8 9 10
```

```
for (i in 10 downTo 1) print(i)  
10 9 8 7 6 5 4 3 2 1
```

```
for (i in 1..10 step 2) print(i)  
1 3 5 7 9
```

While ciklus

```
var nrOfPizza = 10
while (nrOfPizza > 0) {
    nrOfPizza--
}

var nrOfPizzaOrdered = 0
do {
    nrOfPizzaOrdered++
} while (nrOfPizzaOrdered < 10)
```

Függvények

Függvények

Paraméter szintaxis: <name1>: <type1>, <name2>: <type2>, ...

- Függvényeket a **fun** kulcsszóval deklarálhatunk

- Példa visszatérési érték nélkül:

```
fun printMessage(message: String) {  
    println(message)  
}
```

- Példa visszatérési értékkel:

```
fun generateRandomNumber(bound: Int): Int {  
    return Random().nextInt(bound)  
}
```

A visszatérési érték típusa a **return** után található

Unit visszatérési érték

- Kotlin-ba minden függvény visszatér valamilyen értékkel
- Ha egy függvénynek nincs valós visszatérési értéke, Unit-al tér vissza
- A `Unit` típusnak egy értéke van: `Unit`

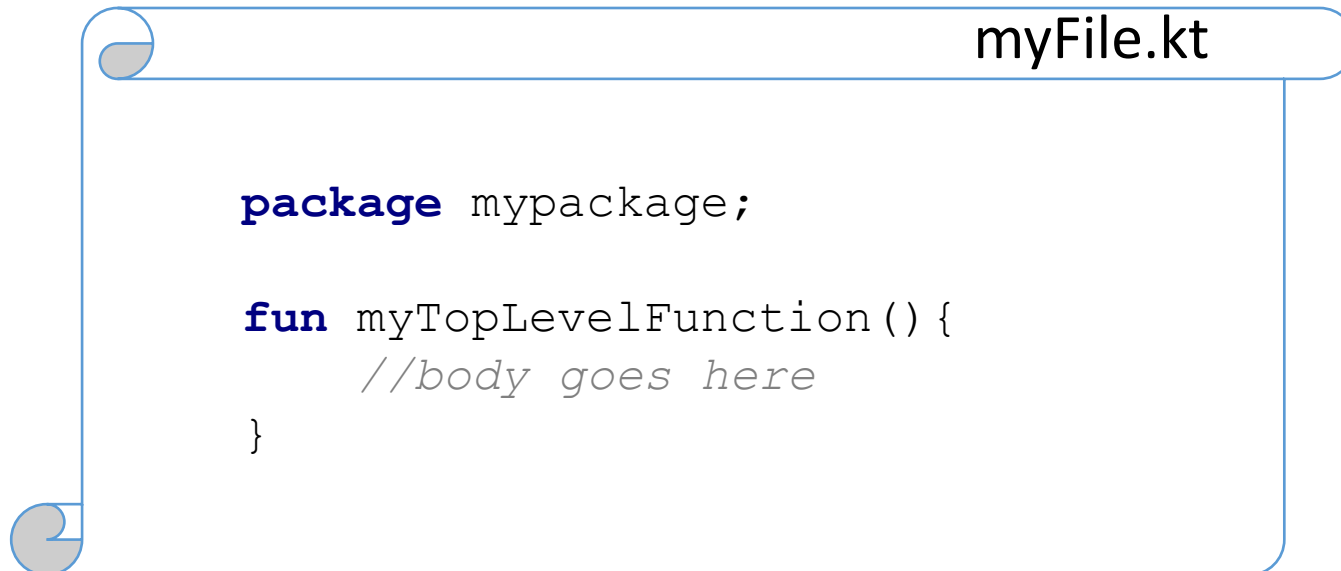
Unit visszatérés esetén a visszatérési érték típusa elhagyható

```
fun printMessage(message: String): Unit {  
    println(message)  
    return Unit //or return  
}
```

A return szintén elhagyható

Függvények – top level függvények

- Top level függvények
 - Osztályokon kívül is létrehozhatók egy file-ban.
 - Meghívhatók a main függvényből, más top level függvényből, osztályon belüli függvényekből vagy `init{}` blokkokból.

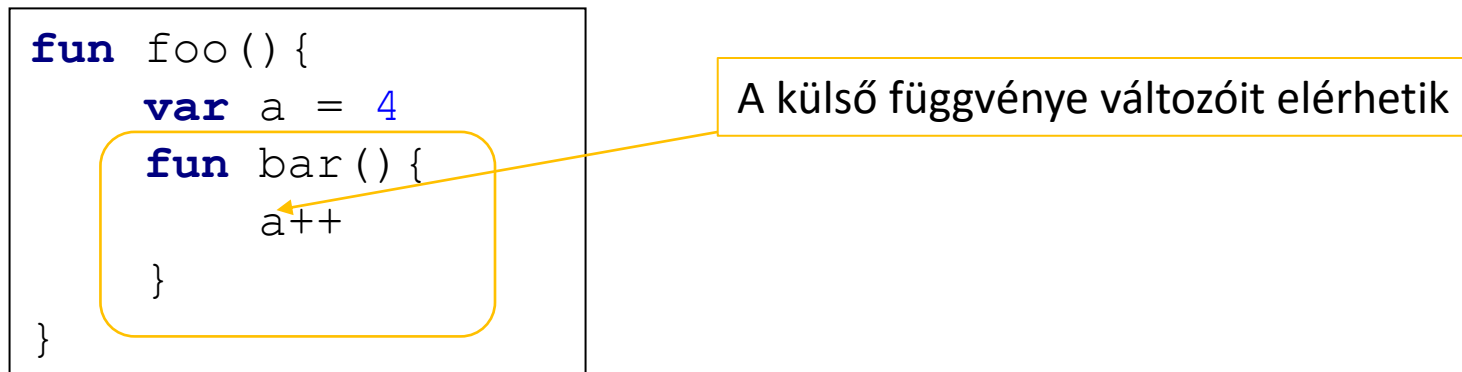


```
package mypackage;

fun myTopLevelFunction() {
    //body goes here
}
```

Függvények – lokális függvények

- Lokális függvények
 - Függvénye elhelyezhető egy függvényen belül is



Függvények – tag függvények

- Tag függvények
 - Osztályon belül kerülnek deklarálásra
 - Egy osztály példányon (objektumon) hívható a `.,'` operátorral

```
class Car {  
    fun start() {  
        // ...  
    }  
}
```

```
var car: Car = Car()  
car.start()
```

vagy

```
Car().start()
```

Függvények – alapértelmezett paraméter értékek (default arguments)

- A függvény paramétereknek lehet alapértelmezett értéke

```
fun generateRandomNumber(bound: Int = 100): Int {  
    return Random().nextInt(bound)  
}
```

A paraméter elhagyható

```
fun main(args: Array<String>) {  
    println("Your random number is: ${generateRandomNumber(50)}")  
    println("Your second random number is: ${generateRandomNumber()}")  
}
```

Your random number is: 3

Your second random number is: 36

Függvények – nevesített argumentumok

```
fun printRandomNumber(bound: Int = 100, message: String = "Your random number is") {  
    var randomNumber = Random().nextInt(bound)  
    println("$message $randomNumber")  
}
```

```
fun main(args: Array<String>) {  
    printRandomNumber()  
    printRandomNumber(40, "Let's see...")  
    printRandomNumber(bound = 20, message = "Hey...")  
    printRandomNumber(message = "Hm...")  
}
```

Függvény hívása
argumentumok
megnevezésével

Your random number is 82
Let's see... 23
Hey... 18
Hm... 81

Amennyiben nem minden
paramétert adunk meg, kell
használni az argumentum
megnevezést


Egy-kifejezésű (single expression) függvények

- Amennyiben egy függvény egy egyszerű kifejezést tartalmaz, a {} zárójelek elhagyhatók és a függvény törzse az = jel után írható
- A visszatérési típus szintén elhagyható ha a fordító ki tudja találni a kifejezésből a típust

```
fun add(a: Int, b: Int) = a + b
```

Egy-kifejezésű függvények – példa (1)

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    return when {  
        mood == "sad" && weather == "rainy" -> "Watch a good film"  
        weather == "sunny" && temperature > 20 -> "Go and play in the garden!"  
        temperature < 5 -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}
```




Tegyük Kotlinosabbá



```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    val isSadAndRainy = mood == "sad" && weather == "rainy"  
    val isSunnyAndWarm = weather == "sunny" && temperature > 20  
    val isCold = temperature < 5  
    return when {  
        isSadAndRainy -> "Watch a good film"  
        isSunnyAndWarm -> "Go and play in the garden!"  
        isCold -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}
```


Egy-kifejezésű függvények – példa (2)

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    val isSadAndRainy = mood == "sad" && weather == "rainy"  
    val isSunnyAndWarm = weather == "sunny" && temperature > 20  
    val isCold = temperature < 5  
    return when {  
        isSadAndRainy -> "Watch a good film"  
        isSunnyAndWarm -> "Go and play in the garden!"  
        isCold -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}
```



```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    return when {  
        isSadAndRainy(mood, weather) -> "Watch a good film"  
        isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
        isCold(temperature) -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}  
  
private fun isSadAndRainy(mood: String, weather: String): Boolean {  
    return mood == "sad" && weather == "rainy"  
}  
  
private fun isSunnyAndWarm(weather: String, temperature: Int): Boolean {  
    return weather == "sunny" && temperature > 20  
}  
  
private fun isCold(temperature: Int): Boolean {  
    return temperature < 5  
}
```

Egy-kifejezésű függvények – példa (3)

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    return when {  
        isSadAndRainy(mood, weather) -> "Watch a good film"  
        isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
        isCold(temperature) -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}  
  
private fun isSadAndRainy(mood: String, weather: String): Boolean {  
    return mood == "sad" && weather == "rainy"  
}  
  
private fun isSunnyAndWarm(weather: String, temperature: Int): Boolean {  
    return weather == "sunny" && temperature > 20  
}  
  
private fun isCold(temperature: Int): Boolean {  
    return temperature < 5  
}
```

Single expression függvény

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    return when {  
        isSadAndRainy(mood, weather) -> "Watch a good film"  
        isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
        isCold(temperature) -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}  
  
private fun isSadAndRainy(mood: String, weather: String): Boolean = mood == "sad" && weather == "rainy"  
private fun isSunnyAndWarm(weather: String, temperature: Int): Boolean = weather == "sunny" && temperature > 20  
private fun isCold(temperature: Int): Boolean = temperature < 5
```

Egy-kifejezésű függvények – példa (4)

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    return when {  
        isSadAndRainy(mood, weather) -> "Watch a good film"  
        isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
        isCold(temperature) -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}
```

Visszatérési típus elhagyható

```
private fun isSadAndRainy(mood: String, weather: String): Boolean = mood == "sad" && weather == "rainy"  
private fun isSunnyAndWarm(weather: String, temperature: Int): Boolean = weather == "sunny" && temperature > 20  
private fun isCold(temperature: Int): Boolean = temperature < 5
```

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    return when {  
        isSadAndRainy(mood, weather) -> "Watch a good film"  
        isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
        isCold(temperature) -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}
```

```
private fun isSadAndRainy(mood: String, weather: String) = mood == "sad" && weather == "rainy"  
private fun isSunnyAndWarm(weather: String, temperature: Int) = weather == "sunny" && temperature > 20  
private fun isCold(temperature: Int) = temperature < 5
```

Egy-kifejezésű függvények – példa (5)

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    return when {  
        isSadAndRainy(mood, weather) -> "Watch a good film"  
        isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
        isCold(temperature) -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}
```

Alapértelmezett érték
beállítható single
expression függvénnyel is

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int =  
    getDefaultTemperature()): String {  
    return when {  
        isSadAndRainy(mood, weather) -> "Watch a good film"  
        isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
        isCold(temperature) -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}  
  
fun getDefaultTemperature() = 24
```

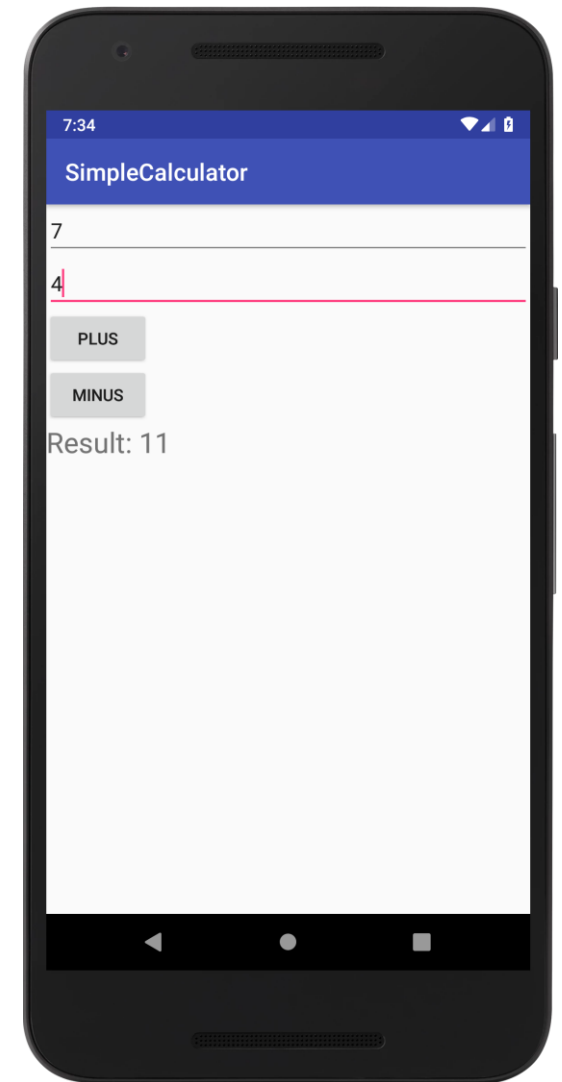
Gyakoroljunk

- Készítsünk egy alkalmazást, amely a Fibonacci számsorozatot jeleníti meg

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, ...

Házi feladat!

- Készítsünk egy egyszerű számológép alkalmazást!



Köszönöm a figyelmet!



Ekler Péter
peter.ekler@aut.bme.hu

Android programozás

Kotlin nyelvi alapok - 3.

Ekler Péter

peter.ekler@aut.bme.hu

Emlékeztető

Emlékeztető

- Null kezelés
- Vezérlési struktúrák
- Függvények

Változók `null` értéke

- Alapból a változók értéke nem lehet `null`

```
var a: Int = null  
error: null can not be a value of a non-null type Int
```

- A ‘?’ operátorral engedélyezhetjük a `null` értéket

```
var a: Int? = null
```

- Lista, melyben lehetnek `null` elemek
- Lista, mely lehet `null`
- Lista, mely lehet `null` és az elemei is lehetnek `null`-ok

```
var x: List<String?> =  
    listOf(null, null,  
    null)
```

```
var x: List<String>? = null
```

```
var x: List<String?>?  
= null  
x = listOf(null,  
    null, null)
```

Null tesztelés és az Elvis operátor

```
var nullTest : Int? = null  
nullTest?.inc()
```

- `inc()` nem hívódik meg, ha `nullTest` `null`

```
var x: Int? = 4  
var y = x?.toString() ?: ""
```

> ha `x` `null`, akkor `y` `""` értéket kap

“Double bang” operator

- Kivételt dob, ha a változó értéke null

```
var x: Int? = null  
x!!.toString()  
kotlin.KotlinNullPointerException
```

Függvények

- Függvény szintaxis

```
fun add(a: Int, b: Int): Int {  
    return a + b  
}
```

- Kifejezés törzs, visszatérési típus elhagyható

```
fun add(a: Int, b: Int) = a + b
```

- Érték nélküli visszatérés – Unit

```
fun printAddResult(a: Int, b: Int): Unit {  
    println("$a + $b értéke: ${a + b}")  
}
```

- Unit elhagyható

```
fun printAddResult(a: Int, b: Int) {  
    println("$a + $b értéke: ${a + b}")  
}
```

Tartalom

- Osztályok
 - Elsődleges konstruktor
 - Másodlagos konstruktor-ok
- Objektumok létrehozása
- Gyakorlás

Kurzus példái:

<https://github.com/peekler/AndroidProgrammingBasics>

Gyakoroljunk

- Készítsünk egy alkalmazást, amely a Fibonacci számsorozatot jeleníti meg

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, ...

Gyakoroljunk – megoldás Fibonacci számok

```
fun fibonacci(n: Int = 100) {  
    var i = 1  
    var t1 = 0  
    var t2 = 1  
  
    while (i <= n) {  
        print("$t1 + ")  
  
        val sum = t1 + t2  
        t1 = t2  
        t2 = sum  
  
        i++  
    }  
}
```

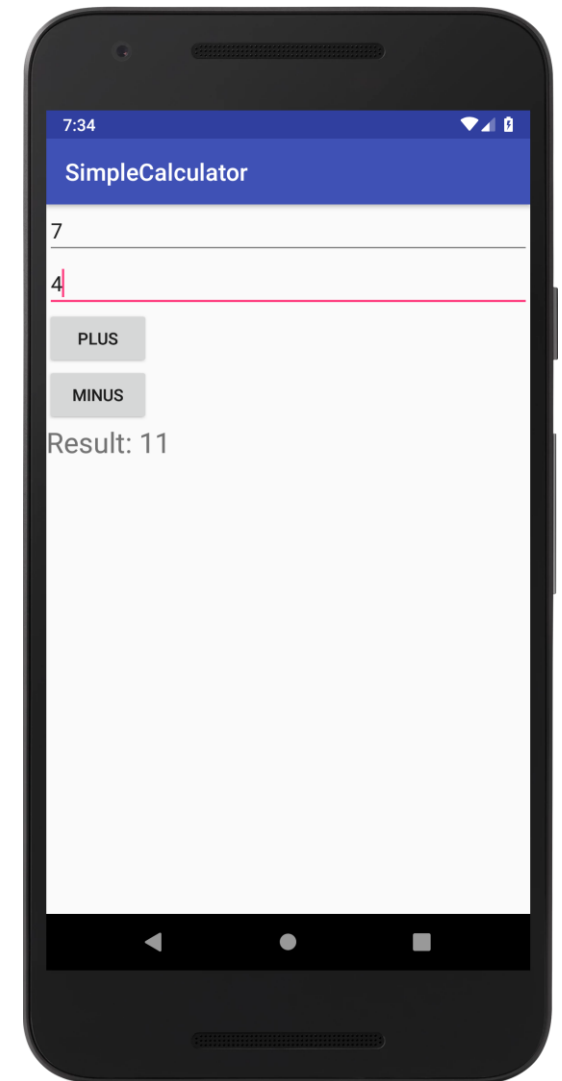
Gyakoroljunk

```
fun fibonacci(): Sequence<Int> {  
    // fibonacci terms  
    // 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, ...  
    return generateSequence(Pair(0, 1), { Pair(it.second, it.first + it.second) }).map { it.first }  
}  
  
println(fibonacci().take(10).toList()) // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

<https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/generate-sequence.html>

Házi feladat!

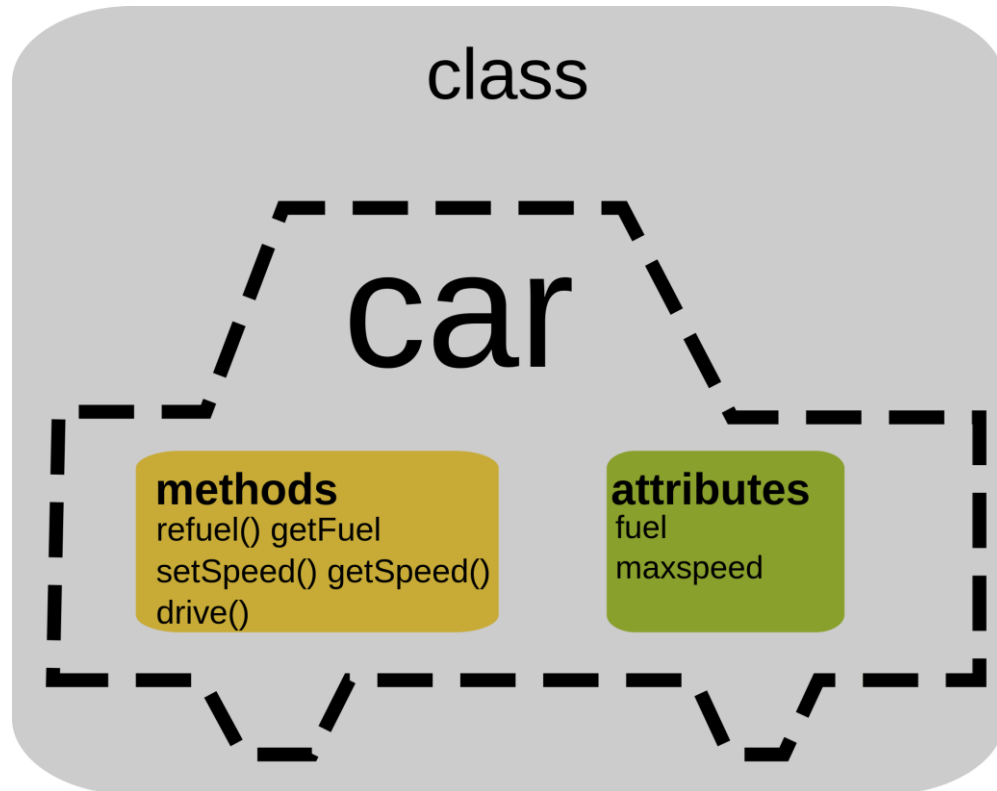
- Készítsünk egy egyszerű számológép alkalmazást!



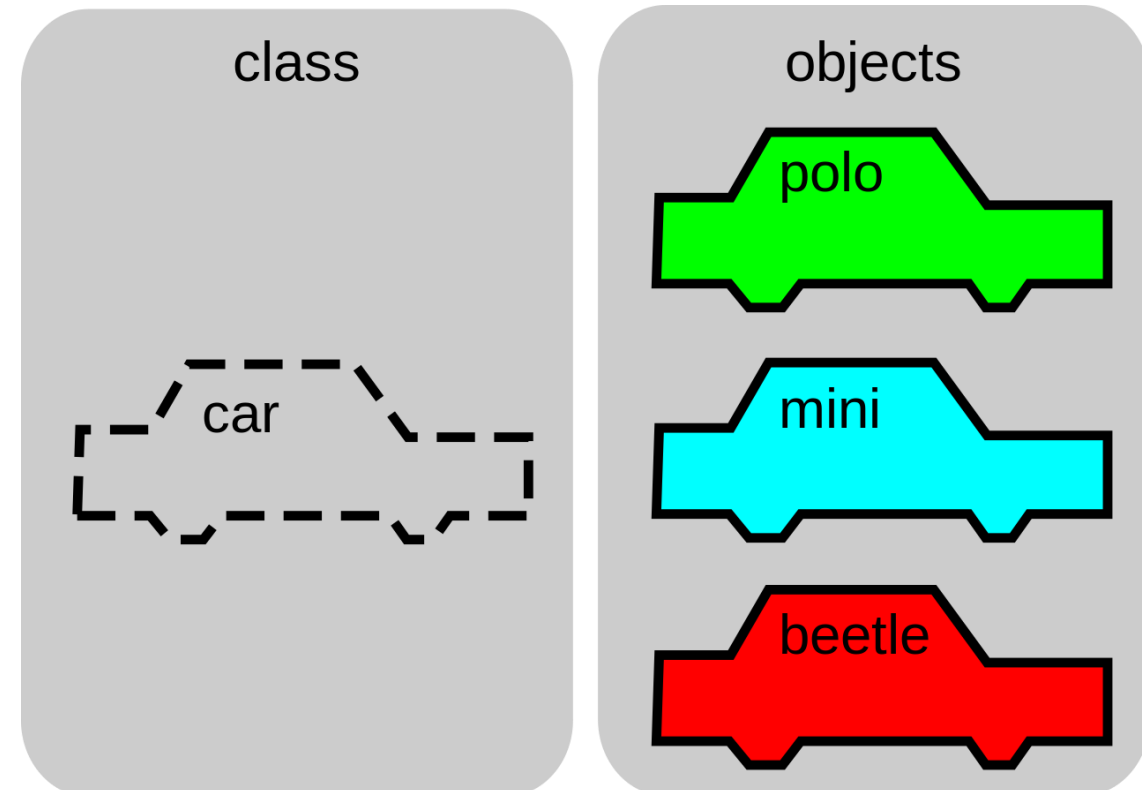
Osztályok, konstruktor

Osztályok, objektumok

- Osztály:



- Objektum:



Források:

<https://www.miltonmarketing.com/coding/programming-concepts/object-oriented-programming-oop/>

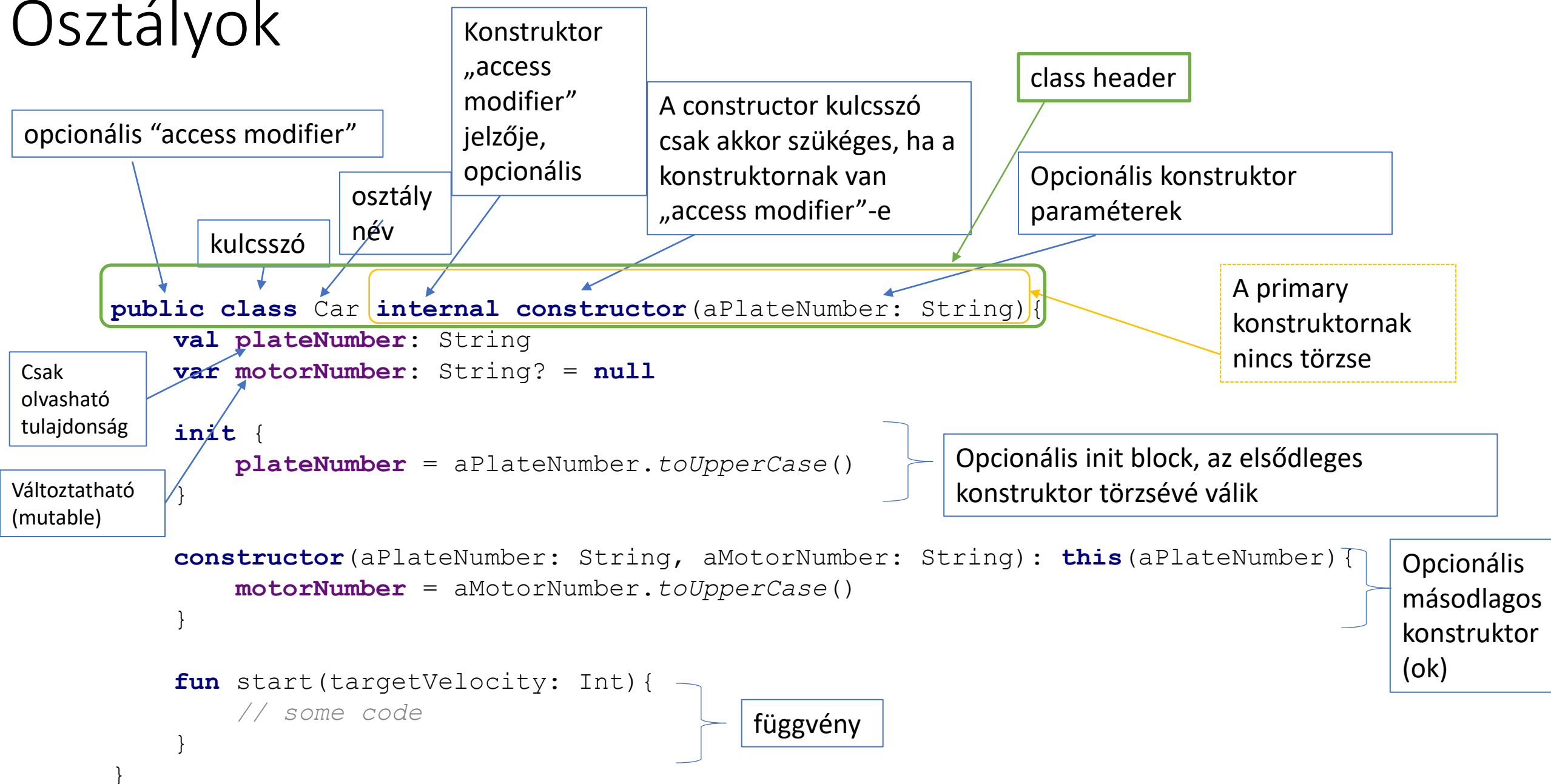
https://commons.wikimedia.org/wiki/File:CPT-OOP-objects_and_classes.svg

Osztályok (class)

- Legrövidebb osztály deklarációs mód `class Car`
- A `kotlin.Any` minden osztály őssztálya
- Egy osztály tartalmazhat:
 - Konstruktorokat
 - Property-eket
 - Init blokkot
 - Függvényeket
 - Belső és beágyazott (inner és nested) osztályokat
 - Objektum deklarációkat

```
public open class Any public constructor() {  
    public open operator fun equals(other: kotlin.Any?):  
        kotlin.Boolean { ... }  
  
    public open fun hashCode(): kotlin.Int { ...}  
  
    public open fun toString(): kotlin.String { ...}  
}
```

Osztályok



Osztályok – Kotlin vs Java

```
public class Car internal constructor(aPlateNumber: String){  
    val plateNumber: String  
    var motorNumber: String? = null  
  
    init {  
        plateNumber = aPlateNumber.toUpperCase();  
    }  
  
    constructor(aPlateNumber: String, aMotorNumber: String):  
        this(aPlateNumber){  
        motorNumber = aMotorNumber.toUpperCase()  
    }  
  
    fun start(targetVelocity: Int){  
        // some code  
    }  
}
```

Kotlin kód

```
public final class Car {  
    @NotNull  
    private final String plateNumber;  
  
    @NotNull  
    public final String getPlateNumber() {  
        return this.plateNumber;  
    }  
  
    @Nullable  
    private String motorNumber;  
  
    @Nullable  
    public final String getMotorNumber() {  
        return this.motorNumber;  
    }  
  
    public final void setMotorNumber(  
        @Nullable String var1) {  
        this.motorNumber = var1;  
    }  
  
    public Car(@NotNull String aPlateNumber) {  
        super();  
        this.plateNumber =  
            aPlateNumber.toUpperCase();  
    }  
  
    public Car(@NotNull String aPlateNumber,  
        @NotNull String aMotorNumber) {  
        this(aPlateNumber);  
        this.motorNumber =  
            aMotorNumber.toUpperCase();  
    }  
  
    public final void start(int targetVelocity) {  
        // some code  
    }  
}
```

Java kód

Konstruktor

- Egy osztálynak egy elsődleges konstruktor és egy vagy több másodlagos konstruktor lehet
- Az elsődleges konstruktor az osztály fejléc (header) része és nincs törzse alapértelmezetten

Elsődleges konstruktor példák

A konstruktor paraméterek használhatók az init blokkban és property inicializáláskor

```
class FibonacciSequence(givenNrOfElements: Int, firstElement: Int = 1) {
```

```
    private val nrOfElements: Int = givenNrOfElements  
    var elements: MutableList<Int> = mutableListOf()
```

```
    init {  
        if (nrOfElements > 0) elements.add(firstElement)  
        if (nrOfElements > 1) elements.add(1)  
        for (i in 2..nrOfElements - 1) {  
            elements.add(elements[i - 2] + elements[i - 1])  
        }  
    }  
}
```

```
fun main(args: Array<String>) {  
    val fibonacciSequence = FibonacciSequence(10, 0)  
    print(fibonacciSequence.elements)  
}
```

A primary konstruktor nem tartalmaz kódot alapértelmezetten, hanem a törzs az **init** blokkba helyezhető el

Az **init** blokk a primary konstruktor részévé válik.

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Elsődleges konstruktor - refaktor

```
class FibonacciSequence(givenNrOfElements: Int, firstElement: Int = 1) {  
  
    private val nrOfElements: Int = givenNrOfElements  
    var elements: MutableList<Int> = mutableListOf()  
  
    init {  
        if (nrOfElements > 0) elements.add(firstElement)  
        if (nrOfElements > 1) elements.add(1)  
        for (i in 2 until nrOfElements) {  
            elements.add(elements[i - 2] + elements[i - 1])  
        }  
    }  
}  
  
fun main(args: Array<String>) {  
    val fibonacciSequence = FibonacciSequence(givenNrOfElements = 10, firstElement = 0)  
    print(fibonacciSequence.elements)  
}
```

Until kulcsszó használata

Függvény paraméter neveket érdemes megadni az olvashatóság érdekében

Elsődleges konstruktor – további refaktor

Primary konstruktorba a var és val kulcsszavak is megadhatók

```
class FibonacciSequence(givenNrOfElements: Int, firstElement: Int = 1) {  
  
    private val nrOfElements: Int = givenNrOfElements  
    var elements: MutableList<Int> = mutableListOf()  
  
    init {  
        if (nrOfElements > 0) elements.add(firstElement)  
        if (nrOfElements > 1) elements.add(1)  
        for (i in 2 until nrOfElements) {  
            elements.add(elements[i - 2] + elements[i - 1])  
        }  
    }  
  
    fun main(args: Array<String>) {  
        val fibonacciSequence =  
        FibonacciSequence(givenNrOfElements = 10, firstElement = 0)  
        print(fibonacciSequence.elements)  
    }  
}
```

```
class FibonacciSequence(private val nrOfElements: Int,  
firstElement: Int = 1) {  
  
    var elements: MutableList<Int> = mutableListOf()  
  
    init {  
        if (nrOfElements > 0) elements.add(firstElement)  
        if (nrOfElements > 1) elements.add(1)  
        for (i in 2 until nrOfElements) {  
            elements.add(elements[i - 2] + elements[i - 1])  
        }  
    }  
  
    fun main(args: Array<String>) {  
        val fibonacciSequence = FibonacciSequence(nrOfElements = 10,  
firstElement = 0)  
        print(fibonacciSequence.elements)  
    }  
}
```

Konstruktor paramétereknek lehet alapértelmezett értéke

Paraméter nélküli konstruktor generálása

- Ha a primary konstruktor minden paraméterének van alapértelmezett értéke akkor egy üres () konstruktor is generálódik

```
class Student(name: String = "")  
  
var noNameStudent = Student()  
var student = Student("Aurora Johnson")
```

```
class Student(name: String)  
  
var noNameStudent = Student()  
var student = Student("Aurora Johnson")
```

Fordítási hiba, mivel nincs paraméteres konstruktor

Másodlagos konstruktor - példa

Delegálás az elsődleges konstruktornak

```
class Student(private var firstName: String, private var lastName: String) {  
  
    var id: Int? = null  
    var address: String? = null  
  
    constructor(firstName: String, lastName: String, id: Int) : this(firstName, lastName) {  
        this.id = id  
    }  
  
    constructor(firstName: String, lastName: String, id: Int, address: String) :  
this(firstName, lastName, id) {  
        this.address = address  
    }  
  
}
```

Delegálás a másik másodlagos konstruktornak

Másodlagos konstruktor

- Egy osztálynak lehet egy vagy több másodlagos konstruktora a **constructor** kulcsszó használatával
- Ha egy osztálynak van elsődleges konstruktora akkor minden másodlagos konstruktor köteles delegálni irányába a **this** kulcsszóval direct vagy indirect módon
 - Az init blokk ugyanúgy meghívódik az elsődleges konstruktoron keresztül
 - Ha nincs elsődleges konstruktor az init blokk akkor is meghívódik, mivel a delegálás akkor is megtörténik implicit módon
- Fontos: osztály mezők/property-k nem jönnek létre a másodlagos konstruktor paraméterein keresztül

```
class Student(var firstName: String, var lastName: String){  
    constructor(firstName: String, lastName: String, var id: Int): this(firstName, lastName){  
        ...  
    }  
}
```

Másodlagos konstruktor kiváltás

```
class Student(private var firstName: String, private var lastName: String) {  
    var id: Int? = null  
    var address: String? = null  
    constructor(firstName: String, lastName: String, id: Int) : this(firstName, lastName) {  
        this.id = id  
    }  
    constructor(firstName: String, lastName: String, id: Int, address: String) : this(firstName, lastName, id) {  
        this.address = address  
    }  
}
```

Sokszor alapértelmezett értékek használatával kikerülhető a másodlagos konstruktor létrehozása

```
class Student(var firstName: String, var lastName: String, var id: Int? = null, var  
address: String? = null)
```

Használat:

```
var student = Student("Aurora", "Smith")  
var student2 = Student("Aurora", "Smith", 678, "Budapest, Kossuth street 1")  
var student3 = Student("Aurora", "Smith", address = "Budapest, Kossuth street 1")  
var student4 = Student(firstName = "Aurora", lastName = "Smith", id = 678, address = "Budapest,  
Kossuth street 1")
```


Üres konstruktor

- Nem absztrakt osztályok esetén, ha nincs sem elsődleges sem másodlagos konstruktor akkor egy üres konstruktor kerül bevezetésre
 - Amennyiben ennek láthatóságát állítani szeretnénk, akkor ki kell írunk manuálisan az üres konstruktort és jelölni a láthatóságot

```
class MyClass private constructor () {  
    ...  
}
```

Mezők és tulajdonságok

Mezők és tulajdonságok

- Java-val ellentétben direkt mezőket nem lehet létrehozni
- Kotlin-ban property-ket használunk amiknek van gettere és settere
- Ha egy property-nek szükséges egy „tároló mező” (backing field), a Kotlin generál egyet

Propertyk használata

```
print(car.name)
```

Property kiolvasásakor a getter hívódik meg

```
car.name = "Nissan Leaf"
```

Property értékének
megváltoztatásakor a setter
hívódik meg

- A getter láthatósága megegyezik a property láthatóságával

Mutable (változtatható) property-k

Deklarálhatók az elsődleges konstruktorban vagy az osztályon belül is

Osztályon
belüli
deklaráció:

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]  
    [<getter>]  
    [<setter>]
```

Példa

Változtatható (mutable) property-k

```
class Car(val numberPlate: String, var manufacturer: String, var type: String) {  
    var name: String  
    get() = "$manufacturer $type"  
    set(value: String) {  
        val splittedName = value.split(" ".toRegex())  
        manufacturer = splittedName[0]  
        type = splittedName[1]  
    }  
  
    private var currentFuelLevel: Int = 0  
}
```

Egyedi getter

Egyedi setter

A setter paramétere „value”
alapértelmezetten, de meg lehet
változtatni

A paraméter típusa jelölhető

Az alapértelmezett getter és setter
automatikusan létrejön

Éréték mezők (Backing Field)

```
class Car(val numberPlate: String, var manufacturer: String, var type: String) {  
  
    var name: String  
        get() = "$manufacturer $type"  
        set(value: String) {  
            val splittedName = value.split(" ".toRegex())  
            manufacturer = splittedName[0]  
            type = splittedName[1]  
        }  
  
    private var currentFuelLevel: Int = 0  
        get() = field  
        set(value) {  
            field = value  
        }  
}
```

A **name** property-nek nincs érték mezője, mivel más property-ktől függ csak

Alapértelmezett getter és setter

A backing field a „field”-en keresztül el is érhető

Backing Field – Példa

```
class Car(val numberPlate: String, var manufacturer: String, var type: String) {  
  
    var name: String  
        get() = "$manufacturer $type"  
        set(value: String) {  
            val splittedName = value.split(" ".toRegex())  
            manufacturer = splittedName[0]  
            type = splittedName[1]  
        }  
  
    private var currentFuelLevel: Int = 0  
        set(value) {  
            if (value <= 100)  
                field = value  
            else  
                field = 100  
        }  
}
```

Biztosítjuk, hogy a **currentFuelLevel** ne legyen 100%-nál nagyobb

Alapértelmezett getter és egyedi setter

Kvíz

Melyik állítás igaz

- 1) A programozó nem deklarálhat mezőket az osztályba
- 2) A mezők automatikusan generálódnak a fordító által ha szükséges
- 3) A mezők és a property-k nem ugyanazok

Kvíz

Melyik állítás igaz

- 1) A programozó nem deklarálhat mezőket az osztályba
- 2) A mezők automatikusan generálódnak a fordító által ha szükséges
- 3) A mezők és a property-k nem ugyanazok

Nem változtatható (Unmutable) propertyk

Konstruktorba vagy osztály törzsébe is deklarálhatók

Példa deklaráció osztály törzsébe:

```
val <propertyName>[: <PropertyType>] [= <property_initializer>]  
    [<getter>]
```

Setter nem készíthető hozzá

```
class Car(val numberPlate: String, var manufacturer: String, var type: String) {  
  
    var name: String  
    get() = "$manufacturer $type"  
    set(value: String) {  
        val splittedName = value.split(" ".toRegex())  
        manufacturer = splittedName[0]  
        type = splittedName[1]  
    }  
  
    private var currentFuelLevel: Int = 0  
    set(value) {  
        if (value <= 100)  
            field = value  
        else  
            field = 100  
    }  
  
    val isFuelLow: Boolean  
    get() = currentFuelLevel < 10  
}
```

Nem változtatható property-k

Nem készül backing **field** hozzá

Backing field - példák

Visszafordított Java kód

Kotlin kód

```
class A {  
    val b  
        get() = 10  
  
    val c = 7  
  
    var d = 8  
}
```

```
public final class A {  
    private final int c = 7;  
    private int d = 8;  
    public final int getB() {  
        return 10;  
    }  
    public final int getC() {  
        return this.c;  
    }  
    public final int getD() {  
        return this.d;  
    }  
    public final void setD(int var1) {  
        this.d = var1;  
    }  
}
```

Kvíz

Melyik állítás igaz?

- 1) Propertyket az osztály törzsében is deklarálhathatunk
- 2) Egyedi setter készítése kötelező mutable property esetén
- 3) Propertyk az elsődleges és a másodlagos konstruktor paraméterei között is készíthetők

Kvíz

Melyik állítás igaz?

- 1) Propertyket az osztály törzsében is deklarálhathatunk
- 2) Egyedi setter készítése kötelező mutable property esetén
- 3) Propertyk az elsődleges és a másodlagos konstruktor paraméterei között is készíthetők

Kvíz

Az alábbi osztály esetén a *foo* property-nek lesz generált backing field-je?

```
class B{  
    val foo: Int  
    get() = 10  
}
```

- 1) Igen
- 2) Nem

Kvíz

Az alábbi osztály esetén a *foo* property-nek lesz generált backing field-je?

```
class B{  
    val foo: Int  
    get() = 10  
}
```

- 1) Igen
- 2) Nem

Belső és beágyazott osztályok

Beágyazott osztályok

```
class Outer{  
    var someProp = 2;  
  
    class Nested{  
        fun someFun () {  
            someProp++  
        }  
    }  
}
```

Fordítási hiba: *Unresolved reference someProp*

A külső (outer) osztály elemei nem érhetők el

```
Outer.Nested().someFun()
```

Belső osztály példányosítása és függvény hívás

Belső osztályok

```
class Outer{  
    var someProp = 2;  
  
    inner class Nested{  
        fun someFun() {  
            someProp++  
            this@Outer.someProp++  
        }  
    }  
}  
  
Outer().Nested().someFun()
```

Ha egy beágyazott osztály **inner** típusú akkor hozzáférhet a külső osztály propertyjeihez

Nincs fordítási hiba

Nincs különbség a két sor között

A külső osztály példánya a „@” jellel elérhető

A belső osztály példányosításához kell egy példány a külső osztályból

Csomagok (package) kezelése

- Java-hoz hasonlóan a fileokat csomagokba szervezzük

```
package mydomain.myapp  
  
private fun someTopLevelFunction() {}  
  
class SomeTopLevelClass
```

A forrás fileok tipikusan a csomag jelöléssel kezdőnek

A `someTopLevelFunction` teljes neve
`mydomain.myapp.someTopLevelFunction`

A `SomeTopLevelClass` osztály teljes neve
`mydomain.myapp.SomeTopLevelClass`

- Ha egy file-ba nincs package megadva akkor alapértelmezetten a „default” csomagba tartozik

Import-ok kezelése

- Importolni lehet:
 - Osztályokat
 - Top level property-ket
 - Függvényeket és propertyket objektum deklarációkban
 - Enum konstansokat

Megjegyzés: Nincs olyan static import mint Java-ban

```
import somepackage.SomeTopLevelClass
import somepackage.SomeTopLevelClass.SomeInnerClass
import somepackage.someTopLevelFunction
import someotherpackage.someTopLevelFunction as someAlias
import somepackage.*
```

Alapértelmezett import-ok

- Minden Kotlin file-ba az alábbi csomagok alapértelmezetten importálásra kerülnek:
 - `kotlin.*`
 - `kotlin.annotation.*`
 - `kotlin.collections.*`
 - `kotlin.comparisons.*` (since 1.1)
 - `kotlin.io.*`
 - `kotlin.ranges.*`
 - `kotlin.sequences.*`
 - `kotlin.text.*`
 - `java.lang.*`
 - `kotlin.jvm.*`

Öröklés, absztrakt osztályok, interfészek

Öröklés

Alapértelmezetten nem lehet leszámaztatni egy osztályból, csak, ha **open** jelző van az osztály előtt

Alapértelmezetten a függvények nem definiálhatók felül, csak ha **open** kulcsszó van előttük

Függvény felüldefiniálására az **override** kulcsszót kell használni

A Button osztály View osztály leszámazottja, a View a Button osztály szülő osztálya

Ha a leszámazott osztálynak van konstruktora, az ősz konstruktort is meg kell hívni

Az őszosztály `onClick()` függvényének hívása

```
open class View(var id: Int){
    open fun onClick(){
        //some code
    }
}

class Button(id: Int, var color: Int): View(id){
    private val silver = 0xC0C0C0

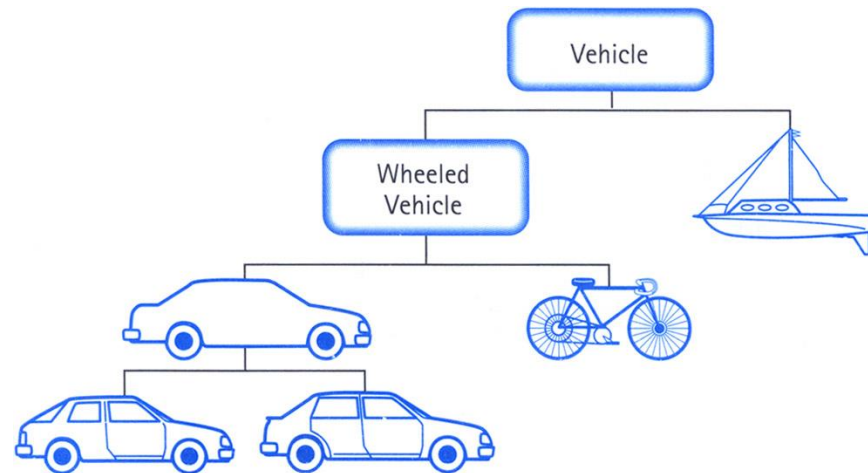
    override fun onClick(){
        super.onClick()
        color = silver
    }
}
```

```
var loginButton: View = Button(1, 0xAAAAAA)
loginButton.click()
```

Példa használat

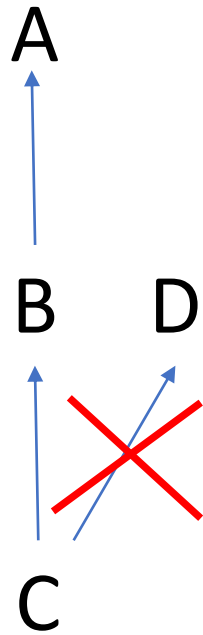
Gyakoroljunk

- Készítsünk osztály struktúrát járművek kezelésére



Öröklés – Egy direkt őszosztály

- Egy osztálynak csak egy őszosztálya lehet, azon keresztül lehet több indirekt őszosztály is



Öröklés – override tiltás

A `click` open függvény ezért felül lehet definiálni

A **`final`** kulcsszó használatával lehet tiltani egy leszármazott felüldefiniálja

Fordítási hiba:
click in Button is final and cannot be overridden

```
open class View(var id: Int) {  
    open fun click() {  
        //some code  
    }  
}  
  
open class Button(id: Int) : View(id) {  
    final override fun click() {  
        // some code  
    }  
}  
  
open class ImageButton(id: Int) : Button(id) {  
    override fun click() {  
    }  
}
```

Öröklés – Property felüldefiniálás

Tulajdonságok is
felüldefiniálhatók,
val-ból lehet **var**
(**var** nem írható felül
val-ra)

Property-k gettere is
felüldefiniálható

```
open class View(var id: Int) {  
    open val color: Int = 0x000000  
  
    open val tooltipText: String  
        get() = "This is a tooltip text of view: $id"  
}  
  
open class Button(id: Int, override var color: Int = 0xFFFFFFFF, var text: String) : View(id) {  
    override val tooltipText: String  
        get() {  
            return text  
        }  
}
```

Absztrakt osztályok

Ha egy osztály **abstract** függvényeket tartalmaz, önmagának is **abstract**-nak kel lennie

```
abstract class View(var id: Int) {  
    abstract val color: Int  
    abstract fun click()  
}  
  
class Button(id: Int) : View(id) {  
    override val color: Int = 0x000000  
  
    override fun click() {  
        // some code  
    }  
}
```

abstract függvényeknek nincs implementációja és az **open** kulcsszó nem kötelező

abstract függvény implementáció leszármaztatott osztályban

Abstract osztály – Nem abstract függvények felüldefiniálása abstract függvénnyel

```
open class View(var id: Int) {  
    open fun click() {  
        //some code  
    }  
}  
  
abstract open class Button(id: Int) : View(id) {  
    override abstract fun click()  
}
```

Lehetséges!

Interfészek – mit tartalmazhatnak?

Inferfész az **interface** kulcsszóval deklarálható

Tartalmazhatnak property-ket, amiknek nincs backing field-jük

Abstract property-ket is tartalmazhatnak

Szintén nincs backing field

Abstract függvényeket is tartalmazhatnak, de nem kell az **abstract** kulcsszót használni

```
interface HttpRequest {  
    val baseUrl: String  
    get() = "https://myapp.com/api"  
  
    var apiUrl: String  
  
    val fullUrl: String  
    get() = baseUrl + apiUrl  
  
    fun send()  
  
    fun onError() {  
        //some code  
    }  
}
```

Interfészek – Implementáció

```
interface HttpRequest {  
    val baseUrl: String  
    get() = "https://myapp.com/api"  
  
    var apiUrl: String  
  
    val fullUrl: String  
    get() = baseUrl + apiUrl  
  
    fun send()  
  
    fun onError() {  
        //some code  
    }  
}
```

Az absztrakt tagokat implementálni kell

```
class LoginHttpRequest : HttpRequest {  
    override var apiUrl: String = "/login"  
  
    override fun send() {  
        // some code  
    }  
}
```

```
fun foo(request: HttpRequest) {  
    //some code  
}  
  
fun bar() {  
    val loginHttpRequest = LoginHttpRequest()  
    foo(loginHttpRequest)  
}
```

Példa használat

Több interfész implementálása

- Egy osztály több interfészt is implementálhat



```
interface A {  
    fun a()  
}  
  
interface B {  
    fun b()  
}  
  
class C : A, B {  
    override fun a() {  
        //some code  
    }  
  
    override fun b() {  
        //some code  
    }  
}
```


Abstract osztályok vs interfészek

Abstrac class

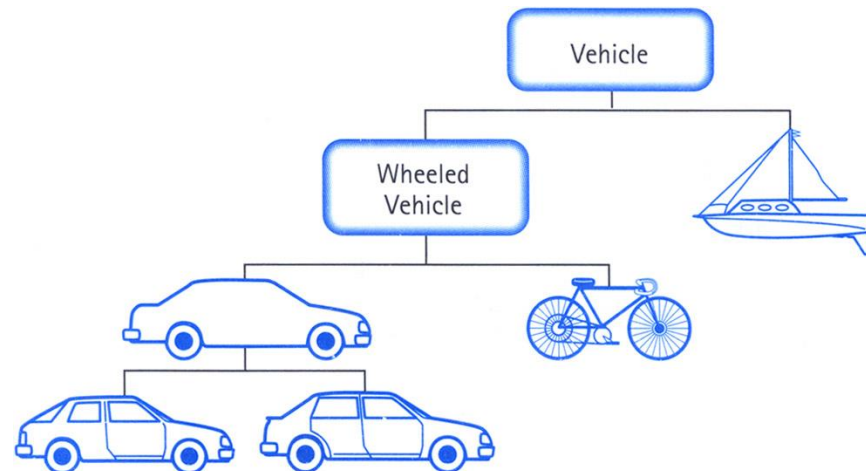
- **Lehet konstruktoruk**
- Tartalmazhatnak implementált függvényeket
- Nem lehet példányosítani
- Tartalmazhatnak property-ket backing field-el

Interface

- **Nem lehet** konstruktoruk
- Tartalmazhatnak implementált függvényeket
- Nem lehet példányosítani
- Csak backing field nélküli property-ket tartalmazhatnak

Gyakoroljunk ismét 😊

- Készítsünk osztály struktúrát járművek kezelésére



Kvíz

Melyik állítás igaz?

- 1) Abstract osztálynak lehet implementált függvénye, interfésznek nem
- 2) Az abstract osztály tartalmazhat abstract függvényt, de az interfészek nem
- 3) Az absztrakt osztályoknak lehet property-je backing field-el, az interfészeknek nem

Kvíz

Melyik állítás igaz?

- 1) Abstract osztálynak lehet implementált függvénye, interfésznek nem
- 2) Az abstract osztály tartalmazhat abstract függvényt, de az interfészek nem
- 3) Az absztrakt osztályoknak lehet property-je backing field-el, az interfészeknek nem

Data Class

```
data class Product(var name: String, var price: Double)
```

- Fő cél az adatkezelés
- A fordító automatikusan generálja az alábbiakat az elsődleges konstruktorba levő elemek alapján:
 - equals() és hashCode()
 - toString() (például: **Product(name=bread, price=10.0)**)
 - componentN() property-k elérése (N: sorszám)
 - copy() függvény

```
var bread = Product("bread", 10.0)  
var homeMadeBread = bread.copy(name = "home made bread")
```

- Beépített osztályok: Pair, Triple

Objektum deklaráció – Osztályok egy példánnyal (singleton)

Singleton létrehozása az **object** kulcsszóval

```
object User {  
    var username: String? = null  
    var password: String? = null  
  
    fun login(username: String, pwd: String) {  
        this.username = username  
        this.password = pwd  
    }  
  
    fun logout() {  
        username = null  
        password = null  
    }  
}
```

Az osztály neve egyben az objektum

Csak egy példány lesz mindig a User osztályból

```
fun someFunction() {  
    User.login("eva", "1234")  
}  
  
fun someOtherFunction() {  
    getPersonalizedAds(User.username)  
}
```

Példa használat

Object – Lehet őse

```
interface Person{  
    val name: String?  
}  
  
object User: Person{  
    var username: String? = null  
    var password: String? = null  
    override val name: String? = username  
  
    fun login(username: String, pwd: String) {  
        this.username = username  
        this.password = pwd  
    }  
  
    fun logout() {  
        username = null  
        password = null  
    }  
}
```

Lehet ősosztálya, implementálhat interfészt is

Object deklaráció vs Object kifejezés

Object Declaration

- Akkor jön létre a példány amikor először használjuk

```
object SomeSingleton {  
    val someProperty = 1  
}
```

Object Expression

- Egyből létrejönnek ahol használatba kerülnek

```
fun bar() {  
    foo(object: SomeInterface{  
        override fun someInterfaceMethod() {  
            // ...  
        }  
    })  
}
```

Anonim objektum

```
fun bar() {  
    val someObject = object {  
        val city = "Budapest"  
        val country = "Hungary"  
    }  
    print("$someObject.city is in ${someObject.country}")  
}
```

Készíthetünk egyszerűen objektumokat

Companion Object

Companion object neve

```
class SomeClass{  
    companion object SomeCompanionObject{  
        fun foo() {  
            // ...  
        }  
    }  
}
```

Használhatók az osztálynéven keresztül egyszerűen

```
val foo1 = SomeClass.SomeCompanionObject.foo()  
val foo2 = SomeClass.foo()
```

- A companion object-nek csak egy példánya lesz

Extension függvények

- Segítségükkel új függvények adathatók meglévő osztályokhoz anélkül, hogy leszármaztatnánk belőlük

```
val someList: List<Int> = listOf(1, 12, 3, 46, 8)
```

```
fun List<Int>.prettyPrint() {  
    println("#####this#####")  
}
```

```
someList.prettyPrint() // outputs: #####[1, 12, 3, 46, 8]#####
```

A **this** a fő objektumra utal (itt a someList-re)

Extension függvény a List<Int>

Extension függvények – Generikus Extension Függvény

```
fun <T> List<T>.prettyPrint() {  
    println("#####$this#####")  
}
```

```
val someStringList = listOf<String>("one", "two")
```

```
someStringList.prettyPrint() //outputs: #####[one, two]#####
```

Extension függvény a List<T>-hez

Extension függvények – További példák

- Beépített extension függvények
- also: bármilyen objektumon hívható

```
val someStringList = listOf<String>("one", "two")  
  
someStringList.get(0).also { print("The receiver is: $it") } // outputs: The receiver is:  
one
```

FYI

```
public inline fun <T> T.also(block: (T) -> Unit): T {  
    contract {  
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)  
    }  
    block(this)  
    return this  
}
```

Házi Feladat



- Objektum orientált programozás
- Készítsük el egy vegyesbolt termékeinek objektum modelljét
 - Termék típusok, leszármaztatás (pl.: Étel->Húsfélék->Hal, Marha, Sertés..)
- Készítsünk legalább 8 osztályt
- Használjunk legalább egy abstract osztályt és interfészt
- Próbáljuk ki az adatmodellt:
 - Készítsünk egy bolt osztályt, ami egy listába tárolja a bolt készletét, amibe különböző termékek lehetnek
 - Készítsünk egy bolt objektumot és legalább 10 termékkel töltsük fel
 - Készítsünk egy ciklust, amely kiírja a bolt nevét és a benne levő összes terméket és azok árát
- Tetszés szerint az adatmodell és a funkciók tovább bővíthetők

Köszönöm a figyelmet!



Ekler Péter
peter.ekler@aut.bme.hu

Android programozás

Kotlin nyelvi alapok - 4.

Ekler-Antal Éva

eva.ekler.antal@gmail.com

Ekler Péter

peter.ekler@aut.bme.hu

Tartalom

1. Láthatósági módosítók
2. Kivételek
3. Operátor túlterhelése
4. Függvény típusok
5. Szálkezelés, coroutine-ok
6. Adatstruktúrák

Kurzus példái:

[https://github.com/peekler/
AndroidProgrammingBasics](https://github.com/peekler/AndroidProgrammingBasics)

Láthatósági módosítók

Visibility modifiers

Láthatósági módosítók (Visibility Modifiers)

- 4 láthatósági módosító
 - **private**
 - **protected**
 - **internal**
 - **public**
- Mely nyelvi elemek használhatnak láthatósági módosítót?
 - Osztályok és interfészek tagja
 - Legfelső szintű deklarációk (nyelvi elemek deklarálva egy .kt fájlban a csomag deklarációval egy szinten)
- Az alapértelmezett láthatóság a **public**, akkor is, ha nincs megadva láthatósági módosító

```
package edu.kotlinexamplesmodule4

private val precision = 4

class Calculator{

    public fun add(a: Int, b:Int): Int {
        return a + b
    }

    fun equals(a: Int, b:Int): Boolean {
        return a == b;
    }

    private fun multiply(a: Int, b:Int): Int {
        return a * b
    }

    protected fun min(a: Int, b:Int): Int {
        return if (a < b) a else b;
    }

    internal fun max(a: Int, b:Int): Int {
        return if (a > b) a else b;
    }

}
```

Osztály tagok és interfész tagok láthatósági szabályai

- Egy osztály tartalmazhat:
 - Konstruktorokat
 - Property-eket
 - Init blokkot
 - Függvényeket
 - Belső és beágyazott (inner és nested) osztályokat
 - Objektum deklarációkat
- Egy interfész tartalmazhat:
 - Property-eket
 - Függvényeket
 - Beágyazott (nested) osztályokat
 - Objektum deklarációkat
- Osztályokban és interfészekben használható az összes láthatósági módosító (kivéve init blokknál):
 - **private** – csak az osztályon belül elérhető
 - **protected** – elérhető az osztályon belül és annak leszármazott osztályain belül
 - **internal** – bárki eléri azonos modulon belül, aki eléri az osztályt
 - **public** – bárholnan elérhető ahol az osztály is elérhető

Legfelső szintű deklarációk és láthatósági szabályaik

- Legfelső szinten (nyelvi elemek deklarálva egy .kt fájlban a csomag deklarációval egy szinten) deklarálni lehet

- Függvények
- Property-k
- Osztályok
- Objektumok
- Interfészek

- Láthatósági szabályok legfelső szinten
 - **private** (csak a deklarációs fájlban elérhető)
 - **protected** (nem használható legfelső szinten)
 - **internal** (csak azonos modulból elérhető)
 - **public** (elérhető mindenhol)

```
package topLevelDeclarations

private fun someTopLevelFunction() {}

internal var someProperty: String = "hello"
    private set

class SomeTopLevelClass

internal object SomeTopLevelObject

interface SomeTopLevelInterface
```

Mikor kell importálni?

- Ahhoz, hogy egy legfelső szintű deklarációt használjunk egy másik csomagból, importálnunk kell az **import** kulcsszóval

```
package package2

import com.example.myapplication.Calculator

fun main() {
    val calculator: Calculator = Calculator()
}
```

Modulok

- Egy module Kotlin együtt fordított Kotlin fájlok összesége
 - An IntelliJ IDEA module
 - A Maven project
 - A set of files compiled with one invocation of the Ant task

Kivételek

Exceptions

Mik azok a kivételek?

- A kivétel egy esemény, ami megszakítja a szoftver normál futását
- Minden kivételt egy-egy osztály reprezentál, annak a példányai jönnek létre a szoftverben, amikor a program hibára lép
- A hiba történhet 3 okból:
 - A felhasználó rossz adatot adott meg
 - Pl. nem létező fájl adott meg a programnak
 - Hardver problémák adódnak
 - Pl. elfogyott a tárhely a szoftvert futtató gépen
 - Programozói hibák a szoftverben
 - Pl. A programkódban a programozó rossz adatbázis nevet adott meg egy adat lekérésben

Kivétel dobása: **throw**, elkapása: **try,catch**

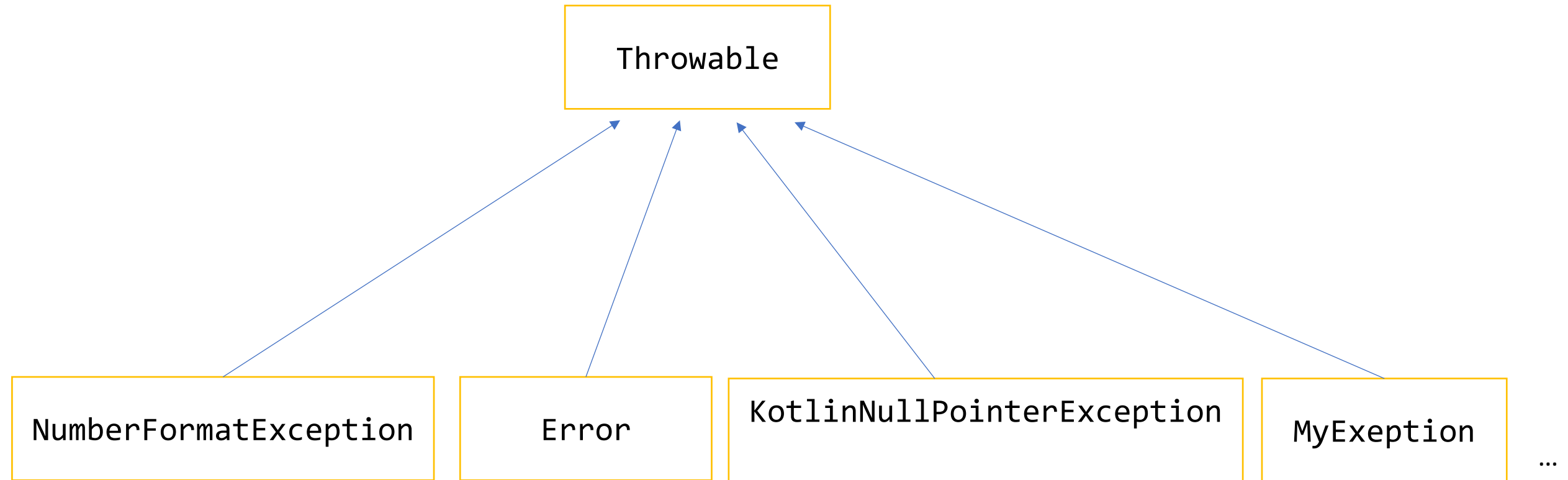
```
class Calculator{
    internal fun divide(a: Int, b:Int): Int {
        if (b != 0)
            return a/b;
        else
            throw WrongDivisor()
    }
}

class WrongDivisor: Throwable("Please provide a divisor other than zero")

fun main() {
    val calculator:Calculator = Calculator()
    try {
        calculator.divide(1, 0)
    } catch (e: WrongDivisor) {
        e.printStackTrace()
        //pl. új dialógus ablak a felhasználónak a hibaüzenettel
    }
}
```

Kivételekre használt osztályok

- Csak a Throwable osztály példányát vagy annak egy leszármazottjának példányát lehet a **throw** kulcsszóval eldobni mint exception



Kivétel dobása, elkapása

```
class Calculator{
    internal fun divide(a: Int, b:Int): Int {
        if (b != 0)
            return a/b;
        else
            throw WrongDivisor()
    }
}

class WrongDivisor: Throwable("Please provide a divisor other than zero")

fun main() {
    val calculator:Calculator = Calculator()
    try {
        calculator.divide(1, 0)
    } catch (e: WrongDivisor) {
        e.printStackTrace()
        //pl. új dialógus ablak a felhasználónak a hibaüzenettel
    }
}
```

Mi a stacktrace?

- Kivétel esetén a futtató környezet kiírja a console-ra azokat a függvényeket, amiknek a meghívásakor a kivétel keletkezett
- A stack legtetején van az a függvény, ahol a hiba gyökere van

```
"C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

```
    at edu.kotlinexamplesmodule4.Calculator.divide$app_debug(Calculator.kt:26)
```

```
    at edu.kotlinexamplesmodule4.CalculatorKt.main(Calculator.kt:44)
```

```
    at edu.kotlinexamplesmodule4.CalculatorKt.main(Calculator.kt)
```

```
Process finished with exit code 1
```

Operátorok túlterhelése

Operator Overloading

Mik az operátorok?

- Példák:
 - Matematikai: +, -, *, /, %
 - Kibővítési: +=, -=, *=, /=, %=
 - Inkrementáló, dekrementáló: ++, --
 - Összehasonlítási: ==, !=, <, >, <=, >=
 - Indexelési: []

Hogy működnek az operátorok a háttérben?

```
fun main() {  
    val a: Int = 1  
    val b: Int = 2  
  
    println(a + b)  
    println(a.plus(b))  
}
```



A + operátor a plus
függvénnnyel van
implementálva az Int
osztályban

Néhány operátor implementációs függvénye

- Teljes lista: <https://kotlinlang.org/docs/reference/operator-overloading.html>

Expression	Translated to
<code>a + b</code>	<code>a.plus(b)</code>
<code>a - b</code>	<code>a.minus(b)</code>
<code>a * b</code>	<code>a.times(b)</code>
<code>a / b</code>	<code>a.div(b)</code>
<code>a % b</code>	<code>a.rem(b)</code> , <code>a.mod(b)</code> (deprecated)
<code>a..b</code>	<code>a.rangeTo(b)</code>

Expression	Translated to
<code>a > b</code>	<code>a.compareTo(b) > 0</code>
<code>a < b</code>	<code>a.compareTo(b) < 0</code>
<code>a >= b</code>	<code>a.compareTo(b) >= 0</code>
<code>a <= b</code>	<code>a.compareTo(b) <= 0</code>

Expression	Translated to
<code>a++</code>	<code>a.inc()</code> + see below
<code>a--</code>	<code>a.dec()</code> + see below

Forrás: <https://kotlinlang.org/docs/reference/operator-overloading.html>

Hogyan terhelhetünk túl egy operátort a osztályainkban?

- Az **operator** kulcsszóval definiálni kell a túlterhelendő operátornak megfelelő nevű és paraméter listájú függvényt
 - Ez történhet az osztályban vagy kiterjesztett (extension) függvényként is

```
data class Book(val title: String, val price: Int){  
    operator fun plus(other: Book): Int{  
        return price + other.price  
    }  
}
```

- Használat:

```
val effectiveJava = Book("Effective Java", 200)  
val cleanCoders = Book("Clean Coders", 300)  
print(effectiveJava + cleanCoders)
```

Gyakorlat

- Terheljük túl a Book osztályban a + operátort úgy, hogy ha két könyvet összeadunk, akkor az a könyvek árait adja össze és vonjon le belőle 10-et, mintha 10 Forintnyi kedvezményt adtunk volna a vásárlónak!

Gyakorlat

- Terheljük túl a * szorzó operátort a String osztályban úgy, hogy a * operátor hívás eredményében a string eredeti értékét annyiszor ismételje meg, ahányszor a * operátor paramétere meghatározta azt!
- Elvárt működés:

```
var s = "something"
```

```
println(s*5)
```

```
//console: somethingsomethingsomethingsomethingsomething
```

Függvény típusok

Function Types

Függvény típusok

- Minden függvényt Kotlinban lehet:
 1. Változó értékeként megadni
 2. Paraméterként átadni egy másik függvénynek
 3. Vissztéríteni egy másik függvény visszatérési értékeként

```
val sum = fun (a: Int, b: Int): Int { return a + b }  
val calculatedSum = sum(1, 2)
```

**-> szükség van olyan típusra,
ami a függvény értékű
változók típusát meg tudja
adni**

```
val sum: (Int, Int) -> (Int) = fun (a: Int, b: Int): Int { return a + b }  
val calculatedSum = sum(1, 2)
```

Függvény típusok - példák

```
(Double) -> Unit
```

Függvény típus, melyek egy Double a bemenő paramétere, és nem térít vissza semmi hasznosat (Unit)

```
(Int, String) -> String
```

Függvény típus, melynek bemenő paramétereinek a típusai: Int, String, eredmény típusa: String

```
() -> Unit
```

Függvény típus, melynek semmilyen bemenő paraméter nincs és nem térít vissza semmi hasznosat

```
val foo: () -> Int
```

Egy változó, melynek típusa függvény típus, semmilyen bemenő paramétere ninc, Int típusú adatot ad vissza

Függvény típus példányosítása

```
val sum: (Int, Int) -> (Int) = fun (a: Int, b: Int): Int { return a + b }
```

- 3 módon
 1. Függvény literál használatával
 1. Lambda
 2. Anonymous függvény
 2. Létező deklaráció referenciájával (nem tárgyalt ebben az kurzusban)
 3. Interfész implementációval (nem tárgyalt ebben az kurzusban)

Függvény típus példányosítása lambda kifejezéssel

- A lambdák függvény literálok (A függvény literálok olyan függvények, melyeket a deklaráció helyén át is adunk mint egy kifejezést)

```
val square: (Int) -> (Int) = {number: Int -> number * number}
```

A zárójel a
visszatérési típus
körül opcionális

Opcionális megadni a
paraméter típusát

Lambda kifejezés (mindig kapcsos zárójelek között)

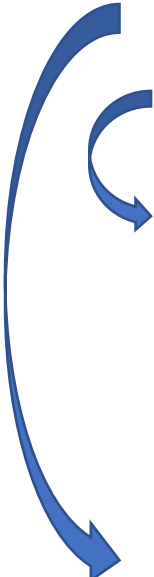
A lambda bemenő
paraméterei a nyíl előtt
vannak. Ha nincs bemenő
paraméter, akkor a nyíl
elhagyható.

A lambda törzs a nyíl után
kerül. Az utolsó kifejezés
lesz a lambda visszatérési
értéke.

Függvény típus meghívása:

```
print(square(4)) // outputs 16
```


Lambda kifejezések – példák



```
val square: (Int) -> (Int) = {number: Int -> number * number}
```

A lambda paraméterének típusa elhagyható

```
val square: (Int) -> (Int) = {number -> number * number}
```

A függvény típus elhagyható, ha azt a fordító ki tudja találni

```
val square = {number: Int -> number * number}
```

Olvashatóbb a kód, ha nem hagyjuk el a függvény típus explicit kiírását!

Függvény típus példányosítása anonymous függvénnnyel

- Az anonymous függvények olyanok mint a többi függvény, csupán a nevük nincs

```
val square: (Int) -> Int = fun (number: Int): Int { return number * number }
```

Opcionális paraméter
típus

Anonymous függvény

A bemenő paraméterek
zárójelek közé kerülnek,
ahogy más függvényeknél
is

Lehet törzse {} között vagy
lehet egy kifejezésű
függvény is. {} esetén
kötelező a **return**
kulcsszó használata

Anonymous függvény meghívása:

```
print(square(4)) // outputs 16
```

Anonymous függvények - példák

```
val square: (Int) -> Int = fun(number: Int): Int { return number * number }
```

A paraméter típusa elhagyható

```
val square: (Int) -> Int = fun(number): Int { return number * number }
```

A függvény típus elhagyható, ha azt a fordító ki tudja találni

```
val square = fun(number: Int): Int { return number * number }
```

Még többet az anonymous függvényekről

Olyanok, mint a többi függvény, de név nélkül

A paraméterei mindig zárójelben kell legyenek (a lambdánál nem kötelező)

A törzsük lehet egy blokk. A **return** kulcsszóval térítenek vissza értéket. (kivétel, amikor Unit-ot, azaz semmi hasznosat nem térítenek)

```
val square: (Int) -> Int = fun(number: Int): Int { return number * number }
```

A törzsük lehet egy kifejezés is

```
val square: (Int) -> Int = fun(number) = number * number
```

A paraméterek típusa elhagyható, ha a fordító ki tudja találni azokat

Egy kifejezésű törzsnél a visszatérési érték típusa eljagyható

Ha a függvény típus nincs explicit kiírva, akkor a paramétereknek kötelező megadni a típusát

```
val square = fun(number: Int) = number * number
```

Lambda vs anonymous függvény

lambda – hosszú verzió

```
val square: (Int) -> Int = {number: Int -> number * number}
```

Anonymous függvény – hosszú verzió

```
val square: (Int) -> Int = fun(number: Int): Int { return number * number }
```

lambda – rövid verzió

```
val square = {number: Int -> number * number}
```

Anonymous függvény – rövid verzió

```
val square = fun(number: Int) = number * number
```

Felsőbb szintű függvények

- Olyan függvények, melyeknek paraméterei között vannak függvény típusúak is

```
fun repeatTask(times: Int, task: () -> Unit){  
    for (index in 1..times)  
        task()  
}
```

Használat:


```
repeatTask(3, { -> println("Bake a cake!")})
```

Kimenet:

```
Bake a cake!  
Bake a cake!  
Bake a cake!
```

Lambdák – Ha a lambda utolsó a paraméterek listájában

```
repeatTask(3, { -> println("Bake a cake!") })
```




```
repeatTask(3){ -> println("Bake a cake!") }
```

Ha egy lambda utolsó a paraméterek listájában, akkor a lambda kihozható a () zárójel utánra.
Ha nem marad más paraméter a zárójelben, akkor a zárójel törölhető

Lambdák – egyedüli paraméter implicit neve

```
val square: (Int) -> Int = { number -> number * number }
```



```
val square: (Int) -> Int = { it * it }
```

Ha egy lambdának egyetlen paraméter van, akkor azt nem muszáj expliciti deklarálni, implicit deklarálva lesz **it** néven

Gyakorlat

- Készítsünk egy olyan felsőbb szintű függvényt, ami kap egy `Int` elemekből álló listát és egy konvertáló függvényt, ami egy `Int`-et át tud alakítani `String`-é. A felsőbb szintű függvényből hívjuk meg a konvertáló függvényt a bemenő lista minden elemére és így állítsuk elő az új listát, ami `String` elemekből fog állni.
 - Adott: `List<Int>`, `converter` függvény
 - Amikor: meghívjuk a `convertListItems` függvényt a fenti paraméterekkel
 - Akkor: a `convertListItems` visszatérít egy `List<String>`-et, amiben a bemenő `List<Int>` elemei át vannak alakítva `String`-é

Lambda kifejezések – példák listákkal

```
val someList: List<Int> = listOf(1, 12, 3, 46, 8, 4, 55, 66)
```

Feladat: Válasszuk ki a páros számokat, rendezzük őket, duplázzuk meg értéküket és írjuk ki őket

FYI `filter(predicate: (T) -> Boolean): List<T>`

FYI `sortedBy`: Returns a list of all elements sorted according to natural sort order of the value returned by specified **[selector]** function

FYI `map(transform: (Int -> R): List<R>` Applies the **[transform]** function to each element

```
someList.filter({ item: Int -> item % 2 == 0 }).sortedBy({ item: Int ->
item }).map({ item: Int -> item * 2 }).forEach({ item -> print("$item ") })
// outputs: 8 16 24 92 132
```



Kotlinizálás: A lambdákat ki lehet vinni a () zárójelen kívülre, és az explicit paraméter deklaráció elhagyható, használható az `it` változó helyette

```
someList.filter{ it % 2 == 0 }.sortedBy{ it }.map{ it * 2 }.forEach{
print("$it ") } // outputs: 8 16 24 92 132
```

Lambda kifejezések – még több példa listákkal

Legyen a Product osztály:

```
data class Product(var name: String, var category: String, var price: Double)
```

Készítsünk egy termék listát:

```
var products: List<Product> = mutableListOf(Product("bread", "food", 10.0), Product("milk",  
"food", 9.0), Product("t-shirt", "clothes", 20.0))
```

1. Feladat: Adjuk vissza az összes food típusú terméket rendezve őket ár szerint

```
val foodListByPrice: List<Product> = products.filter { it.category == "food" }.sortedBy {  
it.price }
```

2. Feladat: Írjuk ki a termékeket termék kategória szerint csoportosítva

```
products.groupBy { it.category }.forEach { key, value -> print("key: $key, value: $value") }
```

Felsőbb szintű függvények – példa függvény visszatérítésére

createLogger
egy függvényt ad
vissza

```
fun createLogger(basicMessage: String) = fun(logMessage: String) {  
    print("$basicMessage: $logMessage")  
}  
val logger = createLogger("Some basic message:")  
logger("Some log message")  
  
//kimenet: Some basic message: Some log message
```

Qvíz

- Mit ír ki a alábbi main függvény futáskor?

```
fun fooo(times: Int = 3, someParam: (Int) -> Int) {  
    for (i in 1..times)  
        print(someParam(i))  
}  
  
fun main(args: Array<String>) {  
    fooo(4, { p -> p * 2 })  
    fooo { p -> p * 2 }  
    fooo { it * 10 }  
}
```

- a) A “main” függvény le sem fordul, mert az “it” nevű változó nincs deklarálva
- b) A “main” függvény le sem fordul, mert a "times" paraméternek nincs értéke a két utolsó “fooo” hívásnál
- c) Minden lefordul, a main kiírja, hogy: 246246102030
- d) Minden lefordul, a main kiírja, hogy: 2468246102030

Qvíz - megoldás

- Mit ír ki a alábbi main függvény futáskor?

```
fun fooo(times: Int = 3, someParam: (Int) -> Int) {  
    for (i in 1..times)  
        print(someParam(i))  
}  
  
fun main(args: Array<String>) {  
    fooo(4, { p -> p * 2 })  
    fooo { p -> p * 2 }  
    fooo { it * 10 }  
}
```

- a) A “main” függvény le sem fordul, mert az “it” nevű változó nincs deklarálva
- b) A “main” függvény le sem fordul, mert a "times" paraméternek nincs értéke a két utolsó “fooo” hívásnál
- c) Minden lefordul, a main kiírja, hogy: 246246102030
- d) Minden lefordul, a main kiírja, hogy: 2468246102030

Szálkezelés, coroutine-ok

Threads, Coroutines

Párhuzamosság vs. konkurencia

- Párhuzamos futás
 - A feladatok egymással egy időben, párhuzamosan futnak
 - Két v több CPU core kell hozzá és ugyanennyi szál, h mindegyik core egy szálát tudjon futtatni egymással egy időben
- Konkurens futás
 - Úgy tűnik, mintha párhuzamosan futnának a feladatok, de valójában egy közös szál halmazon futnak időben felosztva egymás között

Szálak

- Kotlinban a konkurens programozás egyik kelléke
- Thread osztály képviseli

```
class MyThread: Runnable {  
  
    override fun run() {  
        println(Thread.currentThread().getName())  
        Thread.sleep(5) //pretend that some heavy calculation happens  
    }  
  
}
```

```
fun main() {  
    Thread(MyThread()).start()  
}
```

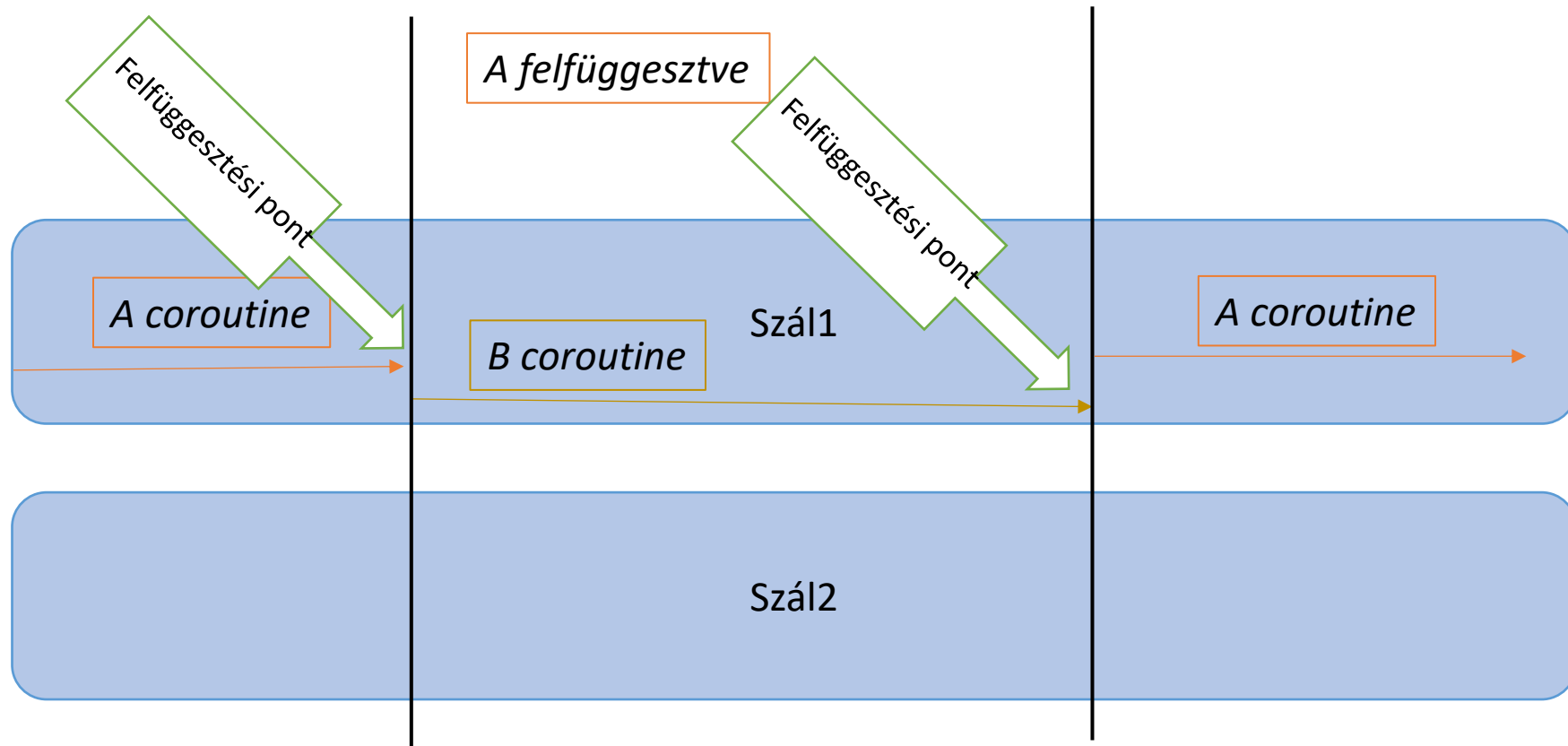
```
fun main() {  
    thread {  
        println(Thread.currentThread().getName())  
        Thread.sleep(5) //pretend that some heavy calculation happens  
    }  
}
```

Szálak

- A modern operációs rendszerek támogatják több szál indítását
- Konkurensen futnak
- A szálak azt az illúziót adják, mintha annyi CPU lenne, ahány szál
- Nemcsak annyi szál futhat ahány CPU van, hanem sokat lehet indítani (1 szálnak kb 1 MB memória szükséges JVM-en)
 - Hogyan lehet blokkolni egy szálat?
 1. CPU intenzív feladattal (CPU leterhelve)
 2. Blokkoló I/O feladattal (nincs kihasználva a CPU, csak várakozik)

Coroutines

- Co + routines = cooperating functions



Coroutines

- A Kotlin a konkurens futtatást implementálta a coroutine-okkal
- Aszincron, nem blokkoló kód írását teszi lehetővé
- Minden coroutine egy vagy több thread-en fut
- Csak egy parancs futtatható egy adott időben egy thread-en

Szál vs Coroutine

- Drága létrehozni
 - Viszonylag sokat létre lehet hozni (erőforrás kérdése)
 - Nem szekvenciális kóddal valósítható meg
- Olcsó létrehozni, kevés erőforrást igényel
 - A szálak számának többszörösét lehet létrehozni
 - Szekvenciálisan írható le a programkód

Demo

- Két coroutine indítása melyek egymással konkurens módon futnak. Feladatuk adatot lekérni egy szervertől a fő szálat nem blokkoló módon.

Felfüggesztő függvények

- Suspending (felfüggesztő) függvények
 - Nem blokkolják a hívó szálát
 - El lehet indítani
 - Fel lehet függeszteni (paused)
 - Folytatni lehet felfüggesztés után
 - Amíg fel van függesztve, nem blokkolja a szálát, amin fut
- Gyakran használt dispatcherek, amik meghatározhatják, hogy a felfüggesztő függvények milyen szálon fussanak:
 - Dispatchers.Default – CPU intenzív feladatok
 - Annyi szállal dolgozik a háttérben, ahány CPU core van a gépen
 - Dispatchers.IO – I/O intenzív feladatok

Hasznos olvasnivalók

- Coroutines hivatalos dokumentáció:
<https://kotlinlang.org/docs/tutorials/coroutines/coroutines-basic-jvm.html>
- Android coroutine felhasználási esetek:
<https://www.lukaslechner.com/kotlin-coroutines-use-cases-on-android/>
- A szálakról: [https://en.wikipedia.org/wiki/Thread_\(computing\)](https://en.wikipedia.org/wiki/Thread_(computing))
- Könyvajánló: Learning Concurrency in Kotlin: Build highly efficient and robust applications by Miguel Angel Castiblanco Torres

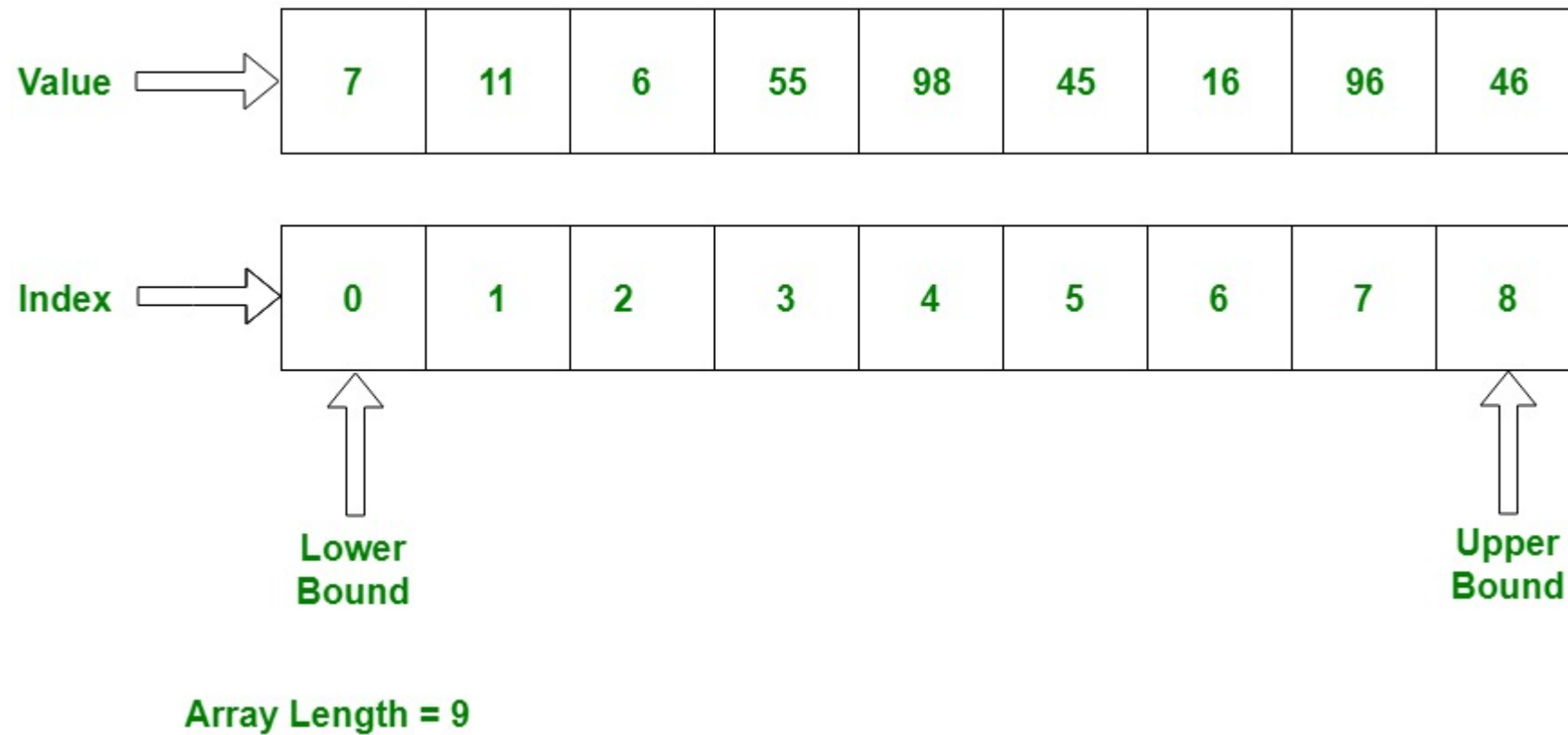
Adatstruktúrák

Collections

Adatstruktúrák

- Adatok különféle tárolási módja
- Különböző adatstruktúrák különböző esetekben hasznosak
 - Például: felsoroláshoz listák, kategorizáláshoz halmazok (set)
- Fő típusok
 - Array (tömb)
 - List (lista)
 - Set (halmaz)
 - Map (kulcs-érték jellegű)

Tömbök (Array)



Tömbök (Array)

- Azonos típusú objektumok
- Fix méret
- Statikus
- Az elemek a pozíció alapján kerülnek indexelésre
- Az első elem indexe 0
- Az utolsó elem indexe a tömb mérete mínusz 1
- A rendszer területet foglal a tömbnek a mérete alapján
 - A tömb ezért nem megváltoztatható (immutable)
 - Az elemek belül viszont megváltoztathatók (mutable)

Tömb kezelés példák

- Egyszerű tömb létrehozás

```
fun main() {  
    val arrayOfNumbers = arrayOf(2, 3, 5, 6, 10)  
    for (number in arrayOfNumbers) {  
        println(number)  
    }  
}
```

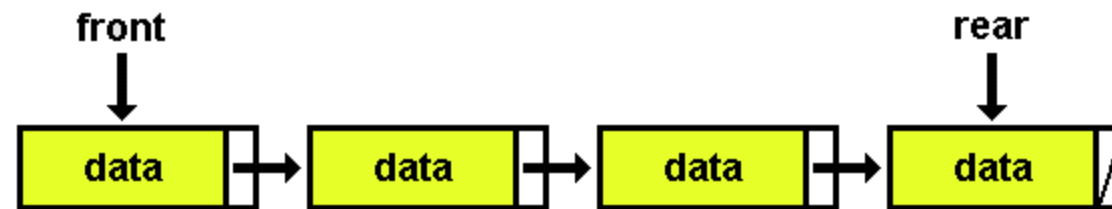
```
fun main() {  
    val arrayOfNumbers = arrayOf(2, 3, 5, 6, 10)  
    arrayOfNumbers.forEach { number -> println(number) }  
}
```

```
fun main() {  
    val someOtherArray = Array(5) { "" }  
    println(someOtherArray)  
}
```

Tömbök használata

```
fun main() {  
    val arrayOfNumbers = arrayOf(2, 3, 5, 6, 10)  
    val firstValue = arrayOfNumbers[0] // Eredmény: 2  
    arrayOfNumbers[0] = 100 // az első elem a ,100' lesz  
    val newFirstValue = arrayOfNumbers[0]  
    println(newFirstValue)  
}
```

Listák



Listák

- Dinamikus méret
- Lehet mutable vagy immutable
 - Méret szerint
 - Tartalom szerint
- Hiba mivel immutable:

```
val list = listOf(2, 3, 5, 6, 7)  
list[2] = 100
```

- Helyesen:

```
val list = mutableListof(2, 3, 5, 6, 7)  
list[2] = 100
```


Listák használata

```
fun main() {  
    val list = mutableListOf(2, 3, 5, 6, 7)  
    list[2] = 100 // works now  
    println(list[2]) // 100  
    list.add(index = 3, element = 500)  
    println(list[3]) // 500  
    list.remove(7)  
    println(list) // [2, 3, 100, 500, 6]  
    list.removeAt(0)  
    println(list) // [3, 100, 500, 6]  
}
```

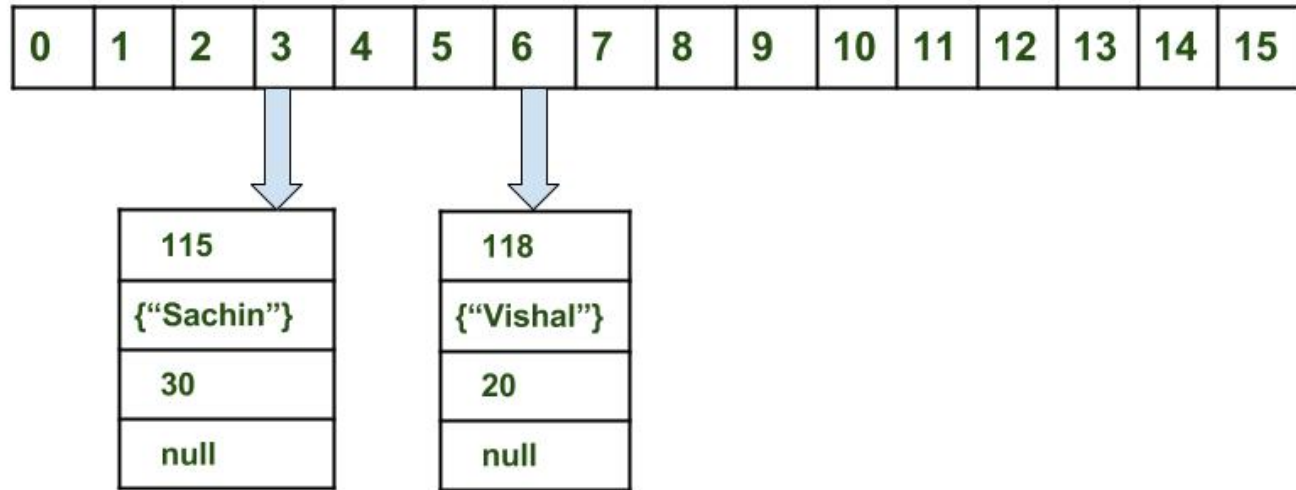
Halmazok (Set)

- Minden elem egyedi, nincs duplikátum
- Hasznos ha ki kell szűrni a duplikátumokat
 - Például ID-k vagy felhasználók tárolása
- Nincsenek indexek
- Rendezetlen *collection* típus

Halmazok (Set) - példa

```
data class Worker(  
    val id: Int,  
    val name: String  
)  
  
fun main() {  
    val workers = mutableSetOf(  
        Worker(id = 5, name = "Susan"),  
        Worker(id = 3, name = "Mike"),  
        Worker(id = 5, name = "Joe"),  
        Worker(id = 4, name = "Susan")  
    )  
    println(workers) // [Worker(id=5, name=Susan), Worker(id=3, name=Mike), Worker(id=5, name=Joe),  
        // Worker(id=4, name=Susan)]  
    val removedWorker = Worker(id = 5, name="Joe")  
    workers.remove(removedWorker)  
    println(workers) // [Worker(id=5, name=Susan), Worker(id=3, name=Mike), Worker(id=4, name=Susan)]  
}
```

Map-ek



Map-ek

- Kulcs-érték párokat tárol
- Mindegyik elemnek van egy egyedi kulcsa
- A pár értéke bármilyen objektum lehet (kulcs és érték is)
- Lehet mutable vagy immutable

Map példa

```
fun main() {  
    val httpHeaders = mutableMapOf(  
        "Authorization" to "your-api-key",  
        "ContentType" to "application/json",  
        "UserLocale" to "US")  
    httpHeaders["Authorization"] = "something else"  
    println(httpHeaders["Authorization"])  
    httpHeaders.put("Hello", "World")  
    println(httpHeaders)  
    httpHeaders.forEach { key, value -> println("Value for key $key is $value") }  
}
```

Gyakorló feladat

- Készítsünk egy alkalmazást, amely képes tárolni egy cég alkalmazottait
- Az alkalmazásnak csak az adattárolási/adakezelési (collection) részét kell megvalósítani
- Egy alkalmazott fő adatai:
 - Név, születési év, lakcím, hány éve van a cégnél
- Az alkalmazottakat egy Map-ben tárolja, mindegyik alkalmazott esetén a kulcs egy növekvő szám legyen, vagy egy egyedi string
 - `var uniqueString = UUID.randomUUID().toString()`
- Töltse fel legalább 5 tetszőleges elemmel a Map-et
- Készítsen egy függvényt, amely:
 - Kiírja az összes alkalmazott minden adatát
- Készítsen egy függvényt, amely név szerint ABC sorrendben kiírja az alkalmazottakat
- Készítsen egy függvényt, amely a cégnél eltöltött idő alapján csökkenő sorrendben felsorolja az alkalmazottak nevét és hogy mennyi ideje van a cégnél

Köszönöm a figyelmet!



Ekler-Antal Éva

eva.ekler.antal@gmail.com

Ekler Péter

peter.ekler@aut.bme.hu