Kotlin for dummies by two dummies

“Kotlin developer” på 1-2-3

bouvet

Hvem er vi?

Jan Olav Kjøde

Sondre Dyrkorn-Berg

(Simen Kjernlie)

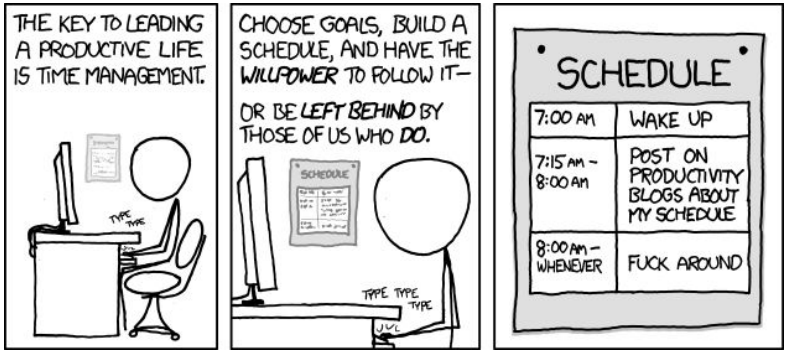
Agenda

- Kursets målsetning
- Introduksjon til Kotlin
- Teori del 1: Kotlin basics; variabler, funksjoner, null-safe, osv
- Praktisk del 1
- Teori del 2: Klasser, arv, interface, collections, lambda, osv
- Praktisk del 2
- Oppsummering

Kursets målsetning

Alle deltakerne skal få en teoretisk og praktisk forståelse av "Kotlin basics" og det som kreves for å komme i gang med Kotlin.

Etter denne sesjonen skal deltakerne kunne lage et enkel program ved hjelp av Kotlin og videre skal deltakerne ha en overordnet forståelse av de ulike hovedkomponentene i Kotlin. De skal dermed kunne begynne å introdusere Kotlin i større prosjekter som de sitter på i dag for å opparbeide seg mer erfaring og kompetanse



Intro

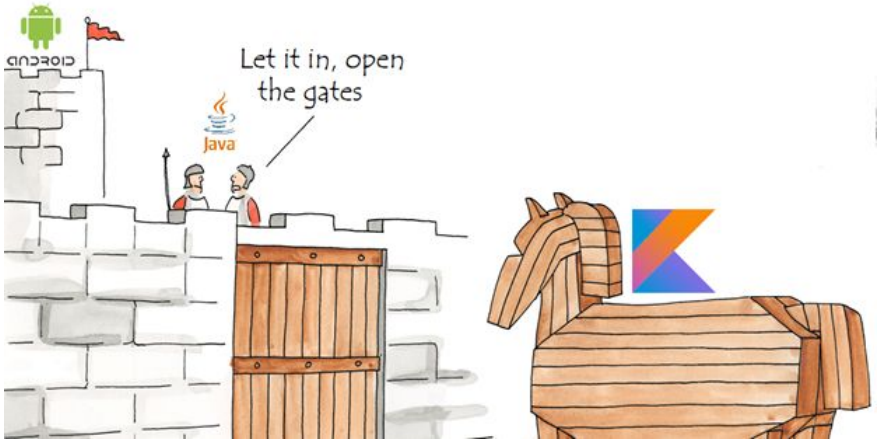
- Nytt programmeringsspråk
- Utviklet av Jetbrains
  - Open Source
- Inspirert mye av Scala, Groovy og tilsvarende

- Introdusert som et alternativ til java
- Lansert i 2011
- V1.0 stable ble lansert i 2016
- Objekt orientert og funksjonelt
- Populært på Android
  og siden 2019 foretrukket av Google

Hvorfor Kotlin

- Mindre boilerplate -> mindre kode -> mindre bugs
- Mer *declarative code*
- Mer *functional style*
- Bedre støtte for *immutability*
- Interoperabilitet med Java kodebase
- Null-safe

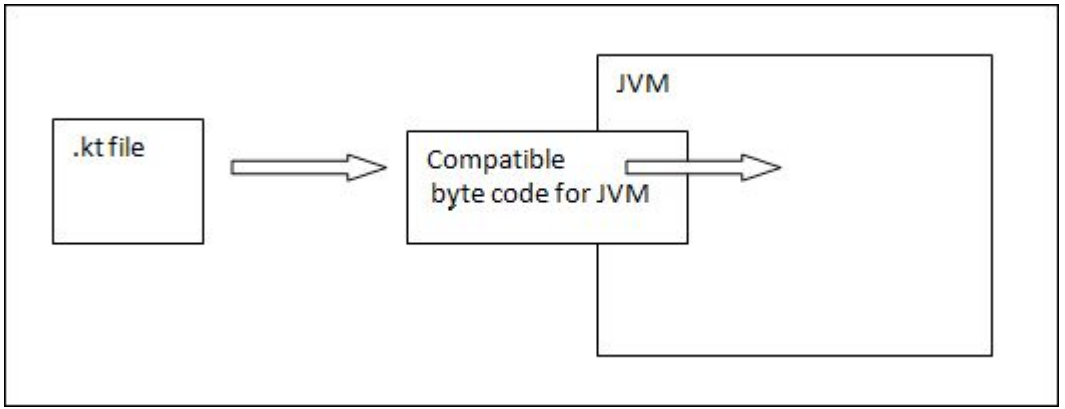
- Lett å lære
- God ytelse og mindre runtime
- Nytt språk med moderne snadder



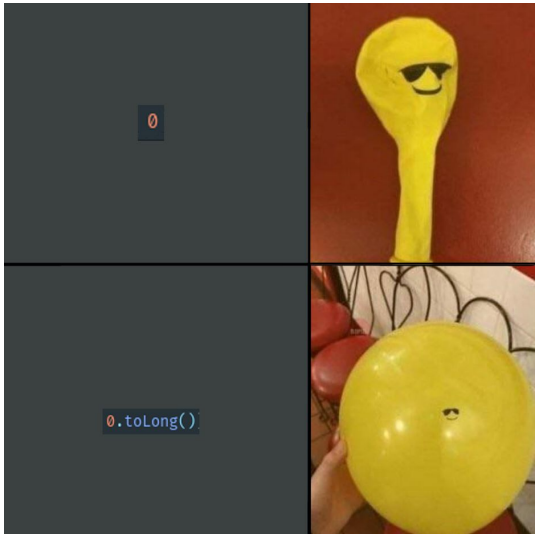
Et mer konsist språk, som er lettere å ta i bruk i en eksisterende java kodebase og har veldig bra interop. Et tryggere språk, hvor flere feil fanges opp av kompilatoren før koden kjøres.

Arkitektur

- Kotlin kan kompileres til:
  - JVM
  - Javascript
  - Native
- Dominerende språk på Android



Variabler



- Alt er et objekt
  - Double, Float
  - Long, Int
  - Short, Byte,
  - String, Char,
  - Long[], Array<Long>
  - Any, Unit, Nothing
- Null-safe
  - Som standard kan ikke et objekt være ***null***
- Semikolon er *optional*
  - Konvensjonen er å kun bruke dem ved flere *statements* på samme linje
- Intern konvertering mellom datatyper er ikke tillatt
  - Istedenfor har variabler metoder for konvertering, som .toLong()

Variabler

- Val → (Immutable reference)
  - Kan **ikke** endres når verdien er tilordnet, og ligner på *final* i java
- Var → (Mutable reference)
  - Kan endres senere i programmet, og ligner på vanlig java variable
- "Type" er valgfritt og blir *Inferred* av kompilatoren

```
var changeableName: String = "Variable"
val finalName: String = "Value"

changeableName = "Can be changed"
//finalName = "Immutable"   -> compiler error

val typeInferred = "Compiler recognizes the String"

println("Value" == finalName)  // java: .equals()
println("Value" === finalName) // java: ==
```

Strings

- *Inline* støtte med $ og ${}
- Støtter flerlinje tekst

```
val firstName = "Simen"
val lastName = "Kjernlie"
val fullName = "${lastName.toUpperCase()}, $firstName"
val multiLineString = """
    SELECT *
    FROM my_table
    WHERE id = :id
    -- can use " without escaping
  """.trimIndent()
```

Funksjoner og metoder

- Syntaksforskjeller
  - Nøkkelord: *fun*
  - Returtype på slutten av signatur
  - Returtype blir *inferred* i oneliner (kan være mer enn en linje)
  - Argument navn og type
- Kan være på toppnivå utenfor klasser

```
fun squareWithBlock(n: Int): Int {
    return n * n
    //always 'return' in a block
}
fun squareOneLiner(n: Int) =  n * n
println("square(3) == ${square(3)}") //> 9
```

Unit som returtype

- Samme som *void* i Java
- Returneres hvis ingen (andre) verdier blir returnert
- Returnerer et faktisk objekt av typen *Unit*
- *Unit* kan bli *inferred*
- Trenger ikke *return*

```
fun printer(n: Int) {
    println("The number is $n")
}

fun printerWithUnit(n: Int): Unit {
    println("The number is $n")
}
```

Digg med metoder i Kotlin!

- Argumentene i en funksjon kan ha standardverdier

```
fun printTemperature(degrees: Double, unit: String = "Celcius") {
    println("temperature is $degrees $unit" )
}

printTemperature(37.0, "Celcius")
printTemperature(37.0)

> temperature is 37.0 Celcius
> temperature is 37.0 Celcius
```

- Navngitte argumenter
  - Rydder opp i forvirrende metoder med samme argumenttyper
    - Hva er (true, true)?

```
fun confusing(name: String, isActive: Boolean, isAdmin: Boolean){}


confusing("Ole", true, true)
confusing(name = "Ole", isAdmin = true, isActive = false)
```

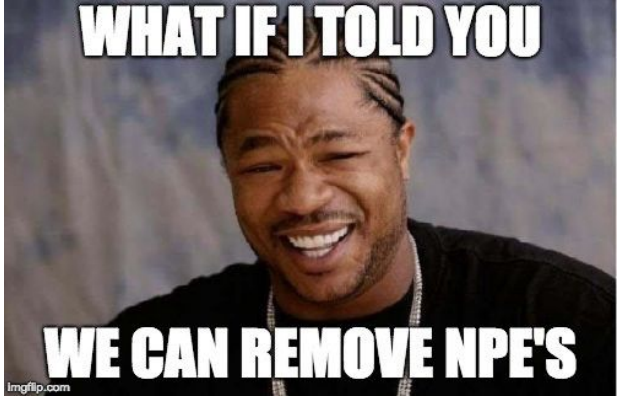
Nullable typer

- Hver type har en komplementær nullable type
  - F.eks: String & String? eller Int & Int?
- Kompilatoren forhindrer at man tildeler *null* til et objekt som ikke kan være *null*

```
//var middleName: String = null //will not compile
var middleName: String? = null
if (middleName != null) {
    //No need to .get() as in Optional
    println("Middle name: $middleName")
}
```

bouvet

Nullsafety, Elvis og !!



- Håndtering av *null*
  - ?. -> verdien dersom ikke *null*, ellers *null*
  - ?: -> verdien dersom utrykket før var *null* (elvis)

```
val middleName: String? = null
val upperMiddleName: String? = middleName?.toUpperCase()
val defaultIfNull: String = middleName?.toUpperCase() ?: ""
```

- !!
  - "The not-null assertion operator"
  - Konverterer enhver verdi til en ikke-null type
  - Kaster *exception* hvis verdien er null

```
val l = b!!.length
```

Smart cast

- Any er super klassen til alle klasser
- Trenger ikke *cast* etter en type har blitt sjekket

- is = instanceof


```
val something: Any = getAnObject()
if (something is String) {
    println(something.toUpperCase())
}
```

bouvet

When = Switch på steroider

Pattern matching	• Smart casting	`val surprise: Any = getSomething()`
		`var whatIsIt: String? = null`
		`when (surprise) {`
		`  is String -> whatIsIt = surprise.toUpperCase()`
		`  42 -> whatIsIt = "Life"`
		`  3.14 -> whatIsIt = "PI"`
		`}`

Pattern to avoid var	• Bruke *when* som et uttrykk
	• Må dekke alle muligheter
	• Kan ha en default (else)
		`val surprise: Any = getSomething()`
		`val whatIsIt: String = when (surprise) {`
		`  is String -> surprise.toUpperCase()`
		`  42 -> "Life"`
		`  3.14 -> "PI"`
		`  else -> "Whatever"`
		`}`

When without arguments	• Kan erstatte lange if-else blokker
	• Må dekke alle muligheter
		`val result = when {`
		`  char == 'A' || char == 'a' -> 1`
		`  else -> -1`
		`}`

bouvet

Praktisk del 1.

1. De som ikke har gjort det; installer IntelliJ IDEA
2. Gå til Tools/Kotlin/Configure Kotlin Plugin Updates og oppdater til 1.3.61
3. Klone prosjekt fra Github: https://github.com/bouvet-sandvika/kotlin_workshop
4. Gjør oppgaver under **exercises/part1**
   a. **Functions**
   b. **Strings**
   c. **Types**
5. Nyt resultatet og drikk kaffe

Klasser

- Primær og sekundær konstruktører
- Standard er *public* og *final*
- Ingen *new* ved opprettelse
- *Get* og *Set* blir autogenerert
  - Kun *var* har setter, ikke *val*

Klasser

bouvet

Java	Kotlin

`public class Person {`
`  private final String firstName;`
`  private String lastName;`
`  public Person(String firstName, String lastName) {`
`  this.firstName = firstName;`
`  this.lastName = lastName;`
`  }`
`  public String getFirstName() {`
`  return firstName;`
`  }`
`  public String getLastName() {`
`  return lastName;`
`  }`
`  public void setLastName(String lastName) {`
`  this.lastName = lastName;`
`  }`
`}`
	`class Person(val firstName: String, var lastName: String){}`

`//bruk av person i java`
`Person simen = new Person("Simen", "Kjærnlie");`
`println(simen.getFirstName());`
`simen.setLastName("Kjernlie");`
	`//bruk av person i kotlin`
	`val simen = Person("Simen", "Kjærnlie")`
	`println(simen.firstName)`
	`simen.lastName = "Kjernlie"`

Klassen *Person* i bytekode

```
$ javap Classes\$Person.class
Compiled from "classes.kts"
public final class no.bouvet.Classes$Person {
  public final java.lang.String getFirstName();
  public final java.lang.String getLastName();
  public final void setLastName(java.lang.String);
  public no.bouvet.Classes$Person(java.lang.String, java.lang.String);
}
```

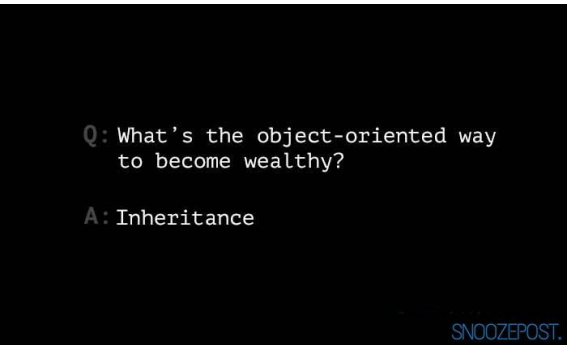
Klasser

- “Nice to have”-features
  - Default verdier
    - Færre konstruktører
  - Navngitte argumenter

```
class Person(val lName: String, val fName: String, val mName: String? = null) {

    fun initials() = "${fName.first()}${mName?.get(0)?:""}${lName[0]}"

}


val defaultMiddleName = Person("Kjøde", "Jan" )
val namedArguments = Person(
    fName = "Jan",
    mName = "Olav",
    lName = "Kjøde")


println(defaultMiddleName.initials()) //> JK
println(namedArguments.initials())    //> JOK
```

Arv

- *Super* må bruke *open*
  - Klasser er *closed* som standard
- Bruker ‘:’
- Må kalle konstruktøren til *super*




```
open class Person(val name: String)
class IdentifiablePerson(val ssn: String, name: String) : Person(name)


val citizen = IdentifiablePerson("16039112345", "Sondre")
println("Name: ${citizen.name}, ssn: ${citizen.ssn}")

> Name: Sondre, ssn: 16039112345
```

bouvet

Interface

- Bruker ‘:’ (samme som arv)
- Må overskrive metoder

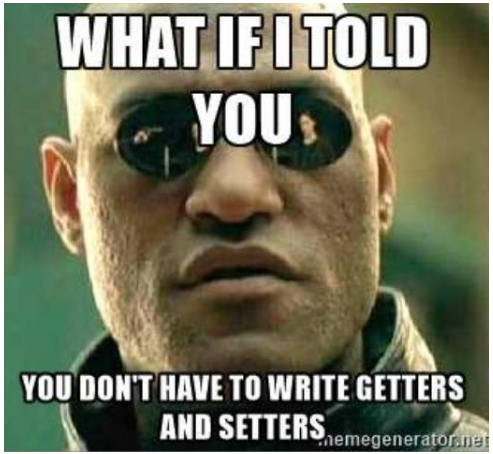
- Vår erfaring: Noen fordeler med å blande Java interface med Kotlin lambda

```
interface PersonService {
    fun addPerson(personToAdd: Person)
}


class PersonServiceImpl : PersonService {
    override fun addPerson(personToAdd: Person) {
        println("Persisting to database: $personToAdd")
    }
}
```

bouvet

Dataklasser

- Når dataene er viktige
- *Immutable* (hvis *val*)
- Autogenererte metoder (kan overskrives)
  - copy()
  - toString()
  - equals()




```
data class Person(
        val firstName: String,
        val lastName: String,
        val age: Int,
        val sex: String = "Not given"
)

val simen = Person("Simen", "Kjernlie", 52)
println(simen)

> Person(firstName=Simen, lastName=Kjernlie, age=52, sex=Not given)

val otherSimen = simen.copy(sex = "Male", age = 26)
println(otherSimen)


> Person(firstName=Simen, lastName=Kjernlie, age=26, sex=Male)
```

Enums

- Properties er enklere enn i Java
  - Ingen konstruktør
  - Ingen *gettere*

Java	Kotlin

`public enum Color {`
`  RED(0xFF0000),`
`  GREEN(0x00FF00),`
`  BLUE(0x0000FF);`
 	
`  private final int rgb;`
 	
`  Color(int rgb) {`
`  this.rgb = rgb;`
`  }`
`}`
	`enum class Direction {`
	`  NORTH, SOUTH, WEST, EAST`
	`}`
	
	`//Can have properties`
	`enum class Color(val rgb: Int) {`
	`  RED(0xFF0000),`
	`  GREEN(0x00FF00),`
	`  BLUE(0x0000FF)`
	`}`

Collections

- *Immutable* som standard

```
val list = listOf("Bobbine", "Erika", "Thomesine")
val set  = setOf("Bobbine", "Erika", "Thomesine")
val map  = mapOf("B" to "Bobbine", "E" to "Erika")
val modifiedList = list + "Georgine"
println(list) //[Bobbine, Erika, Thomesine]
println(modifiedList) //[Bobbine, Erika, Thomesine, Georgine]
```

- Men kan være *mutable*


```
val arrayList  = arrayListOf("Bobbine", "Erika", "Thomesine")
val array      = arrayOf("Bobbine", "Erika", "Thomesine")

val mlist = mutableListOf("Bobbine", "Erika", "Thomesine")
val mset  = mutableSetOf("Bobbine", "Erika", "Thomesine")
val mmap  = mutableMapOf("B" to "Bobbine", "E" to "Erika")
println(mmap)
```

Accessing elements

- Nås ved [index] or .get(index)
- + tilføyer et element (sist / til høyre)
- Kan bli kopiert til *mutable*

```
val fruits = listOf("Apple", "Banana")
val apple = fruits[0]
val banana = fruits.get(1)
val maybeFruit: String? = fruits.getOrNull(2)

val moreFruits = fruits + "Orange"
val mixedUp: String = "Orange" + fruits
val mutableFruits = moreFruits.toMutableList()
mutableFruits.add("Kiwi")
```

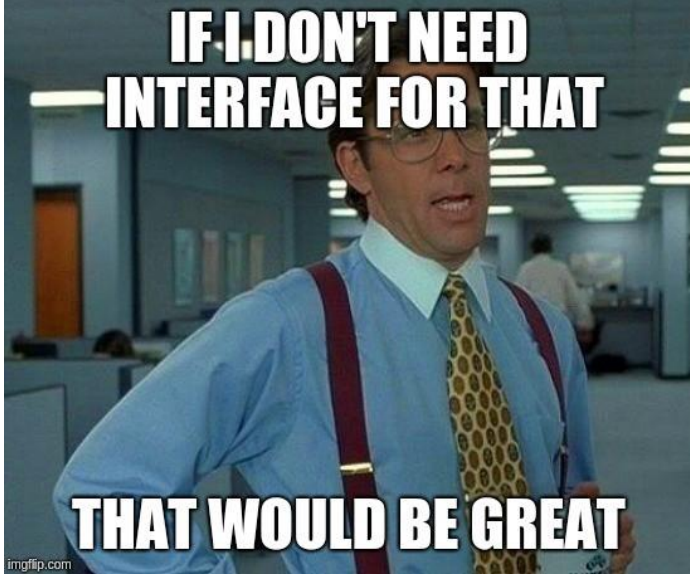
Filter and Map collections

- Likt som i Java
- Ingen *.stream()* eller *.collect()*
- Implisitt parameter er navngitt *it*

```
data class Employee(val name: String, val salary: Long)
val employees = listOf(
    Employee("Sondre", 1_000_000),
    Employee("Junior", 300_000),
    Employee("Jan-Olav", 5_000_000)
)
val highSalaries: List<Long> =
    employees.filter { emp -> emp.salary > 500_000 }
        .map { it.salary }
val average = highSalaries.average()
```

Lambdas

- “Function style”: (params) -> returnType
- Typer kan bli *inferred*




```
val concatenator: (s1: String, s2: String) -> String = { s1: String, s2: String -> s1 + s2 }
val concatenator: (s1: String, s2: String) -> String = { s1, s2 -> s1 + s2 }
val concatenator = { s1: String, s2: String -> s1 + s2 }

val name = concatenator("First", "Last")
```

Lambdas

- Send lambda som argument til funksjon

```
fun intOperator(v1: Int, v2: Int, op: (Int, Int) -> Int ): Int  = op(v1, v2)

val sum = intOperator(2, 3) { n1, n2 -> n1 + n2 }
val sum2 = intOperator(2, 3, Int::plus)
```

- Returner lambda fra funksjon

```
fun times(base: Int): (Int) -> Int = { value -> base * value }

val doubler = times(2)
val trippler: (Int) -> Int = times(3)
println("2 x 4 = ${doubler(4)}")

> 2 x 4 = 8
```

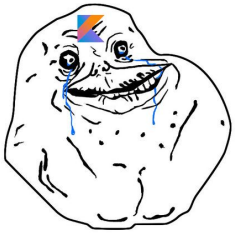
“Scope Functions”

- Fem typer: *let*, *with, run*, *apply*, *also*

- *run*, *with* og *apply* er lambda *receivers*
  - Keyword: this
- *let* og *also* bruker lambda *arguments*
  - Keyword: it

- *apply* og *also* returnerer objektet
- *let*, *run* og *with* returnerer resultatet av lambdaen

Object / Singleton

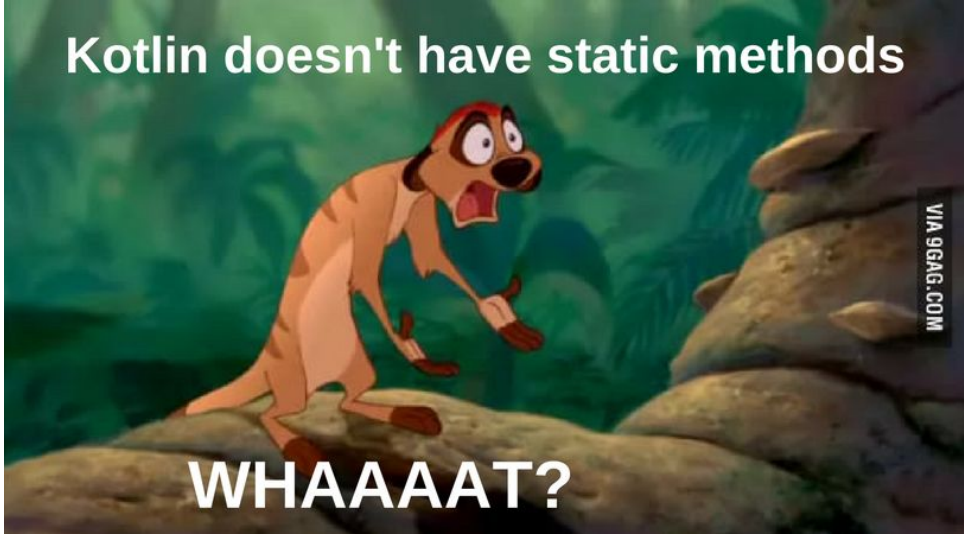


- Kan arve klasser
- Kan implementere interfaces
- Har ingen konstruktør

```
object Utilities {
    fun toUpper(text: String) = text.toUpperCase()
}

val upperCase = Utilities.toUpper("Simen")
```

Companion object

- Objekt inni en klasse
- Delt mellom instanser av en klasse
- F.eks en fabrikk




```kotlin
class MyClass private constructor() {
    companion object {
        fun create(): MyClass = MyClass()
    }
}

val instance = MyClass.create()
```

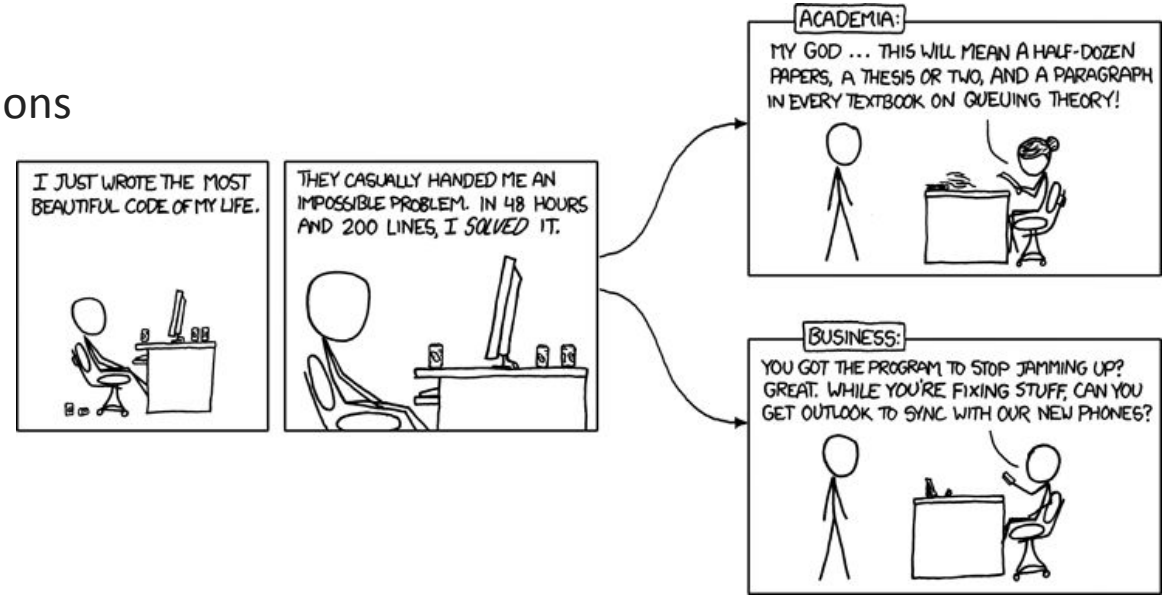
Praktisk del 2

1. Gjør oppgaver under **exercises/part2**
   a. **Classes**
   b. **Collections**
   c. **Lambdas**
2. Nyt resultatet og drikk kaffe

bouvet

Verdt å sjekke ut

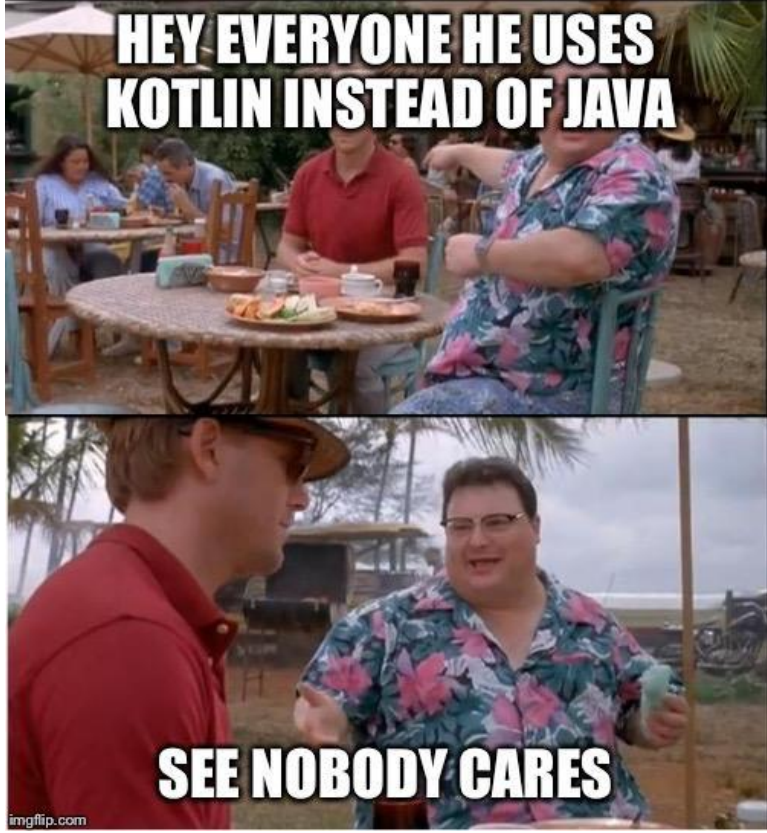
- Extensions - everything is awesome!
  - Kan bytte ut de fleste utility klasser og “bakes” inn i final objekter
- Kotlin functions
- Coroutines
- Ranges
- Folding and reducing Collections



Referanser

Referanser:

- https://kotlinlang.org/docs/reference/
- https://play.kotlinlang.org/koans/overview
- https://www.coursera.org/learn/kotlin-for-java-developers



Kahoot!

bouvet