{ this is Kotlin }

OOP

Tiberiu Tofan

OOP

object oriented programming

- objects - contain data (properties) and code (functions)
- classes - blueprints for creating objects
- features:
  - encapsulation - different level of access on data depending on the context
  - inheritance - a class can extend another class, inherit it’s characteristics and add on top of that
  - polymorphism - overriding, overloading operators and functions
  - generic classes*

* will be covered on the ***Generics*** section of the course

Modules

- an IntelliJ IDEA module
- a Maven project
- a Gradle source set (with the exception that the `test` source set can access the internal declarations of `main`)
- a set of files compiled with one invocation of the `<kotlinc>` Ant task

Visibility Modifiers

- **`public`** - visible everywhere
- `private` - visible just inside the class / file containing the declaration
- `internal` - visible in the same module (a compilation unit)
- `protected` - private + visible in subclasses

Modifiers

- **`final`** - cannot be extended / overridden
- `open` - can be extended / overridden
- `abstract` - the implementation need to be provided by extension

Does having the classes and functions final by default violate the open-closed principle (SOLID)?

Classes

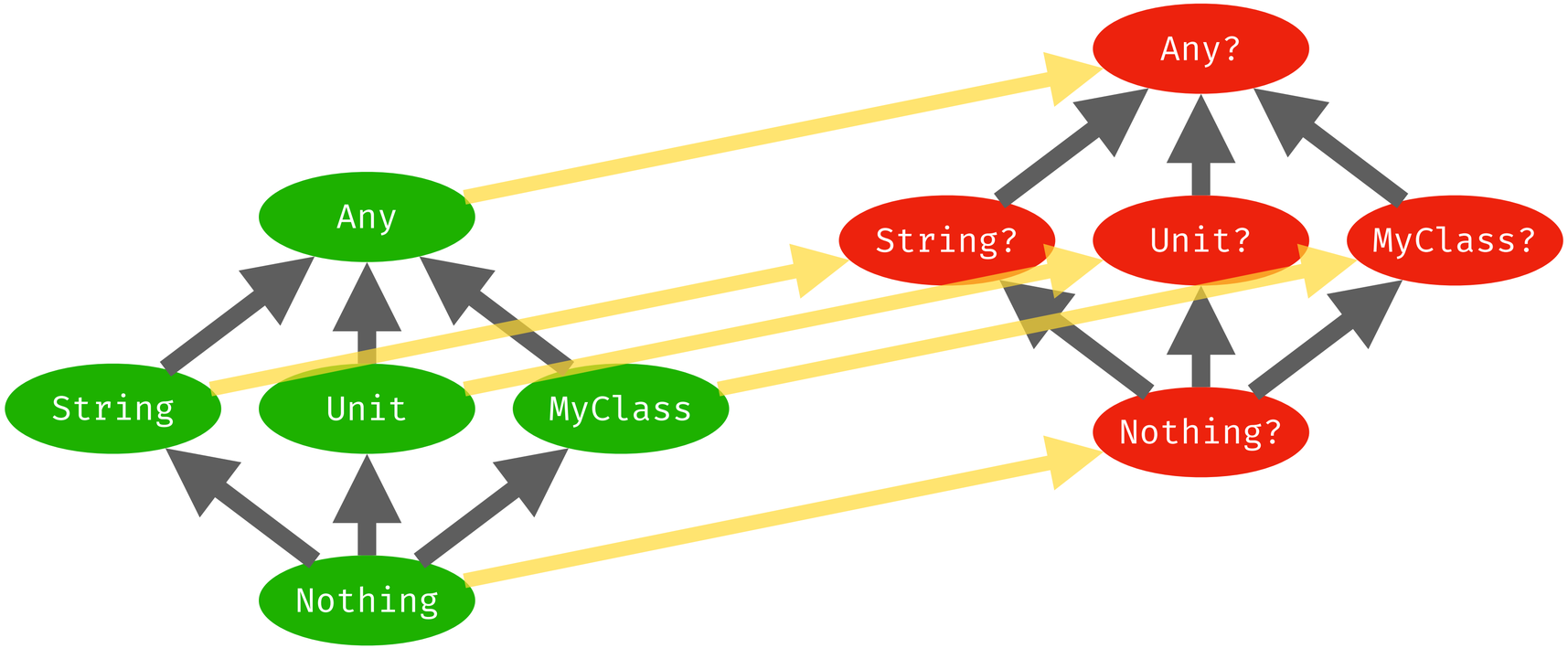
declaration & default constructor

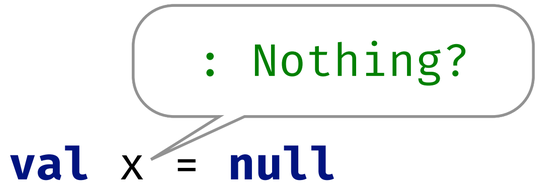
```
class MyClass

val myClassInstance = MyClass()
```

default constructor
without parameters

Type System

from `Any` to `Nothing` via `Unit`




: Nothing?
val x = null

How can a function that returns `Nothing` be implemented?

a)
```
fun iReturnNothing(): Nothing = Nothing()
```

b)
```
fun iReturnNothing(): Nothing = throw Exception()
```

c)
```
fun iReturnNothing(): Nothing {}
```

d)
```
fun iReturnNothing(): Nothing = TODO()
```

e)
```
fun iReturnNothing(): Nothing { while(true) println(".") }
```

How can a function that returns `Nothing` be implemented?

a)
```
fun iReturnNothing(): Nothing = Nothing()
```

b)
```
fun iReturnNothing(): Nothing = throw Exception()
```

c)
```
fun iReturnNothing(): Nothing {}
```

d)
```
fun iReturnNothing(): Nothing = TODO()
```

e)
```
fun iReturnNothing(): Nothing { while(true) println(".") }
```

How can a function that returns `Nothing` be implemented?

a) `fun iReturnNothing(): Nothing = Nothing()` ✗

b) `fun iReturnNothing(): Nothing = throw Exception()` ✓

c) `fun iReturnNothing(): Nothing {}` ✗

d) `fun iReturnNothing(): Nothing = TODO()` ✓

e) `fun iReturnNothing(): Nothing { while(true) println(".") }` ✓

```
public inline fun TODO(): Nothing = throw NotImplementedError()
```

Properties

```
class Account {
    var currency: String = "EUR"
}
```

- property = field + accessor(s)
  - **val** = field + getter
  - **var** = field + getter + setter

Properties

Why do we need properties and not just work directly with fields?

Properties

access

```
class Account {
    var currency: String = "EUR"
}
```

the setter is used

```
account.currency = "RON"

println(account.currency)
```

the getter is used

Java equivalent

```
account.setCurrency("RON");

System.out.println(account.getCurrency());
```

Properties

overriding getters/setters

```
class Account {
    var currency: String = "EUR"
        get() {
            println("accessing property currency with value $field")
            return field
        }
        set(value) {
            println("updating currency from $field to $value")
            field = value
        }
}
```

getters/setter are the only places were it's possible to access `field` directly

```
account.currency = "RON"
```

updating currency from EUR to RON

```
account.currency
```

accessing property currency with value RON

Properties
without fields

```
class Account(val currency: String, val balance: Double) {
    val hasDebt: Boolean
        get() {
            return balance < 0
        }
}
```

No field is generated if neither getter or setter access `field`

Properties
visibility

- `public` by default
- changed for both getters and setters

```
class Account {
    private var currency: String = "EUR"
}
```

```
account.currency = "RON"

println(account.currency)
```

- changed only for setter

```
class Account {
    var currency: String = "EUR"
        private set
}
```

```
account.currency = "RON"

println(account.currency)
```

Extension Properties

```
val String.lastChar: Char
    get() = get(length - 1)


println("Kotlin".lastChar)   n
```

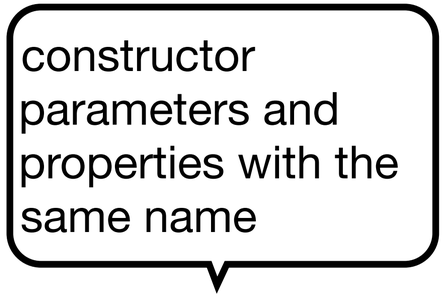
Extension Properties

mutable

```
var MutableList<Int>.last: Int
    get() = get(size - 1)
    set(value) {
        set(size - 1, value)
    }


val ns = mutableListOf(1, 2, 3)
ns.last = 4
println("Last element is ${ns.last}")4
```

Constructors

primary constructor




```
class Client(val name: String)

val wade = Client("Wade Watts")
```

Constructors

primary constructor

```
class Client(val name: String)
```

constructor parameters and properties with the same name

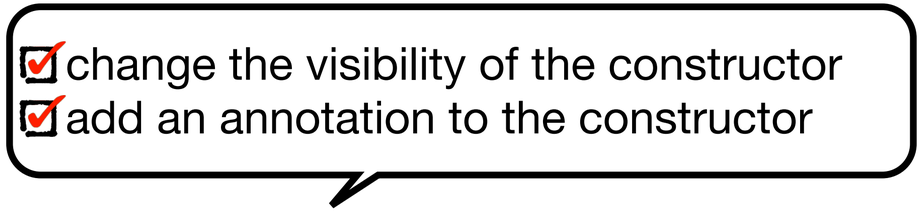
just constructor parameter (no property by default)

```
class Client(name: String) {
    val name: String

    init {
        this.name = name
    }
}
```

```
val wade = Client("Wade Watts")
```

Constructors

explicit primary constructor




```
class Client internal @ConstructorAnnotation constructor(name: String)
```

Constructors

secondary constructors

```
class Client(val name: String) {
    var birthdate: LocalDate? = null

    constructor(name: String, birthdate: LocalDate) : this(name) {
        this.birthdate = birthdate
    }
}
```

cannot declare properties

has to call another constructor

Interfaces

only behaviour

```
interface Identifiable {
    val id: String

    fun isSameAs(other: Identifiable) = id == other.id
}
```

id is not a field, it's just an abstract getter

a concrete function

- cannot have fields
- cannot have constructors
- can have abstract functions - that must be overridden
- can have concrete functions

Inheritance

```
interface Identifiable {
    val id: String

    fun isSameAs(other: Identifiable) = id == other.id
}

open class Person(val name: String)

class Client(override val id: String, name: String) : Person(name), Identifiable
```

same syntax for `extends` and `implements`

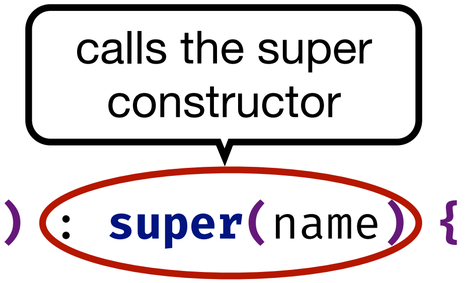
mandatory for any overridden fields or functions

calls the super constructor

Inheritance

explicit constructor

```
class Client : Person, Identifiable {
    override val id: String

    constructor(id: String, name: String) : super(name) {
        this.id = id
    }

    override fun isSameAs(other: Identifiable): Boolean =
        TODO("alternative implementation")
}
```



Inheritance

abstract classes

```
abstract class HotDrinkMaker(val size: Int) {
    fun prepare(): HotDrink {
        boilWater()
        addIngredients()
        return TODO("infuse hot drink")
    }

    abstract fun addIngredients()
    private fun boilWater(): Unit = TODO()
}

class TeaMaker(size: Int) : HotDrinkMaker(size) {
    override fun addIngredients(): Unit = TODO("add plants")
}
```

Could an interface have private functions?

Properties (2)

lateinit

```
class AccountDepositTest {
    lateinit var account: Account

    @BeforeEach
    fun setup() {
        account = Account(“NL77..”, "Current", "EUR", "100".toBigDecimal())
    }

    @Test
    fun `should add the deposited amount to the balance`() {
        assertEquals(account.deposit("3.5".toBigDecimal()).balance,
            "103.5".toBigDecimal())
    }
}
```

Properties (2)
lateinit

- cannot be initalized in constructor, but we don’t want to deal with null
- e.g. the initialization depends on the lifecycle of a framework (usually DI)
- throws an exception if the property is accessed before being initialized
- test for safe access, using property reference:

```
if (this::account.isInitialized) {
    //can be safely accessed here
}
```

Properties (2)

lazy

```
data class Client(val firstName: String, val lastName: String) {
    val fullName: String by lazy {
        "$firstName $lastName"
    }
}
```

- the function is evaluated the first time the property is accessed
- the result is cached
- subsequent access returns the cached result

Could a lazy property be mutable (`var`)?

Data Classes

declaration

```
data class Client(val firstName: String, val lastName: String)
```

- `equals()` / `hashcode()`

  `Client("James", "Halliday") == Client("James", "Halliday")` *true*

- `toString()`

  `Client("James", “Halliday”).toString()` *Client(firstName=James, lastName=Halliday)*

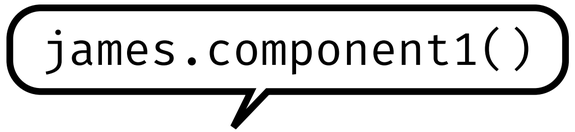
- `componentN()` functions

- `copy()` functions

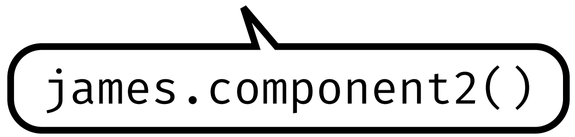
Data Classes

destructuring

```
data class Client(val firstName: String, val lastName: String)


val james = Client("James", "Halliday")
```




```
val (first, last) = james
```




```
println("First name: $first; Last name: $last")
```

First name: James; Last name: Halliday

Data Classes

copy

```
data class Client(val firstName: String, val lastName: String)
```

- named params are very useful
- a new instance is created
- the new instance has the original values replaced with the params of copy, if provided

```
val enrico = Client("Enrico", "Chiesa")
val frederico = enrico.copy(firstName = "Frederico")
println(enrico)
println(frederico)
```

Client(firstName=Enrico, lastName=Chiesa)

Client(firstName=Frederico, lastName=Chiesa)

Data Classes

properties in class body

```
data class Client(val firstName: String, val lastName: String) {
    var birthdate: LocalDate? = null
}
```

- `birthdate` is not used in `equals()` / `hashcode()`
- `birthdate` is not used in `toString()`
- `birthdate` doesn't have a `componentN()` function
- `birthdate` is not a `copy()` function parameter

Data Classes

restrictions

- the primary constructor should have at least one parameter
- all primary constructor params need to be marked as properties (`var` or `val`)
- cannot be
  - abstract
  - open
  - sealed
  - inner

Objects
singleton

```
object Config {
    val port: Int = 80
    val protocol: String = "https"
}
```

Objects
singleton

```
public object Unit {
    override fun toString() = "kotlin.Unit"
}
```

Objects

anonymous interface implementation

```
val runnable: Runnable = object : Runnable {
    override fun run() {
        println("run called")
    }
}
```

Objects

anonymous objects

```
val circle = object {
    val radius: Double = 10.0
    fun area() = Math.PI * radius * radius
}

println(circle.area())
```

Objects

companion object

```
data class Client(val firstName: String, val lastName: String) {
    companion object {
        fun reduplicatedName(name: String): Client =
                Client(firstName = name, lastName = name)
    }
}

val thomas = Client.Companion.reduplicatedName("Thomas")
```

default name of the companion object

```
val thomas = Client.reduplicatedName("Thomas")
```

can be accessed directly from the class name

Objects
companion object

```
class IntegrationTest {
    companion object {
        @BeforeAll
        @JvmStatic
        fun setup() {
            //start up in memory database
        }
    }
}
```

setup is not called if @JvmStatic is missing

if not explicitly specified, companion object functions or properties are not actually static

Sealed Classes

starting with Kotlin 1.5 there are also **sealed interfaces**

```
sealed interface Account {
    val iban: String
    val balance: BigDecimal
}
```

the sealed class/interface and all implementations have to be in the **same package and module**

```
data class CurrentAccount(
    override val iban: String,
    override val balance: BigDecimal
) : Account

data class CreditAccount(
    override val iban: String,
    override val balance: BigDecimal,
    val creditLimit: BigDecimal,
    val interest: BigDecimal
) : Account
```

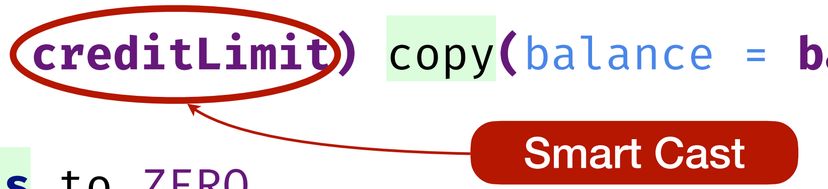
Objects as implementation

```
data class SavingsAccount(
    override val iban: String,
    override val balance: BigDecimal,
    val interest: BigDecimal
) : Account

object TechnicalAccount : Account {
    override val iban: String =
        "RO99TECH1234567812345678"
    override var balance: BigDecimal = ZERO
}
```

Sealed Classes

when is exhaustive without else

```
fun Account.withdraw(amount: BigDecimal): Pair<Account, BigDecimal> = when (this) {
    is CurrentAccount -> copy(balance = balance - amount) to amount
    is SavingsAccount -> copy(balance = balance - amount) to amount
    is CreditAccount ->
        if (balance - amount > creditLimit) copy(balance = balance - amount) to amount
        else this to ZERO
    is TechnicalAccount -> this to ZERO
}
```



Enums

```
enum class Currency(val info: String) {
    EUR("Euro"),
    RON("Romanian leu"),
    USD("United States Dollar"),
    GBP("British Pound Sterling")
}

fun convertToEur(currency: String, amount: Double): Double = when (currency) {
    "RON" -> amount * 0.2
    "USD" -> amount * 0.8
    "GBP" -> amount * 1.2
    "EUR" -> amount
    else -> throw IllegalArgumentException("Unrecognized currency!")
}
```

Enums

```
enum class Currency(val info: String) {
    EUR("Euro"),
    RON("Romanian leu"),
    USD("United States Dollar"),
    GBP("British Pound Sterling")
}

fun convertToEur(currency: Currency, amount: Double): Double = when (currency) {
    RON -> amount * 0.2
    USD -> amount * 0.8
    GBP -> amount * 1.2
    EUR -> amount
}
```

Sealed Classes vs Enums

- both work great with `when` - exhaustive without else if all the branches are covered
- sealed classes may allow custom instances
- enums are multiton - all instances predefined
- enums can be seen as a sealed class that has only objects as implementations, but…
  - enums have direct support on the JVM
  - there are optimized data structures for enums (e.g. `EnumMap`, `EnumSet`)

Delegation

```
class CensoredList(private val delegate: List<String>) : List<String> {
    companion object {
        fun censoredListOf(vararg args: String) = CensoredList(listOf(*args))
    }
    override fun contains(element: String): Boolean = delegate.contains(element)
    override fun listIterator(index: Int): ListIterator<String> =
              delegate.listIterator(index)
    /* 10 other override fun / val */

    override fun toString(): String =
        delegate.map { if (it == "spaghetti") "***" else it }.toString()
} //~30 lines of code
fun main() {
    val words = censoredListOf("giant", "spaghetti", "monster")
    println(words) [giant, ***, monster]
}
```

Delegation
built-in support

```
class CensoredList(private val delegate: List<String>) : List<String> by delegate {
    companion object {
        fun censoredListOf(vararg args: String) = CensoredList(listOf(*args))
    }

    override fun toString(): String =
        delegate.map { if (it == "spaghetti") "***" else it }.toString()
} //~7 lines of code

fun main() {
    val words = censoredListOf("giant", "spaghetti", "monster")
    println(words) [giant, ***, monster]
}
```

spread operator

Delegated Properties

```
class User(val name: String) {
    val password: String by Mask()
}

class Mask {
    operator fun getValue(auditedClass: Any, property: KProperty<*>): String = "***"
}

fun main() {
    val user = User("parzival")
    println(user.password) ***
}
```

- ☑ getValue operator for val
- ☑ both getValue and setValue operators for var

Delegated Properties

standard delegates

- lazy properties: `by lazy`
- observable properties: `by Delegates.observable`
- have a look at `kotlin.properties.Delegates`

❗Favour composition over inheritance