Kotlin OOP

Download the full Chapter source code from here:

https://drive.google.com/file/d/1tpnZlXR7vaBfdJFVjOEw7x4BoRubKFjk/view?usp=sharing

In Kotlin all objects inherit from class Any (it has toString, equals & hashCode)
Unlike Java in Kotlin we can combine few public classes in one file

Lets make our first class:

```
class Student()

fun main() {
    val s = Student()
}
```

Class student has an empty constructor
s has only the functions from Any
If we want the java setter we set it's properties as var if we want only getters they will be val
init function is called in any instance creation - no matter which constructor we invoked

```
class Student(first:String, age:Int) {

    var first:String
    var age:Int

    init {
        println("init called")
        this.first = first
        this.age = age
    }
}

fun main() {
    val s = Student( first: "moshe", age: 67)
    println("name ${s.first} age ${s.age}")
}
```

Please notice that when printing s.first and s.age we use {} meaning it's actually

a function invocation! Calling the get...
The . Actually invoke the getter that was auto generated
Same as s.age = 80; -> this only ok if age is var and not val because the setter is invoked
These are Properties

A better way for writing the class - Kotlin way

```
class Student(first:String, age:Int) {

    var first = first
    var age = age

    init {
        println("init called")
    }
}
```

The init is still called before the initialization

A better way then this (show with auto correction)

```
class Student(var first: String, var age: Int)

    init {
        println("init called")
    }
}
```

This is the primary constructor!

When we add the var or val in the **constructor** then we set the properties to the parameters passed to the constructor and there will be default getters and setters according to the var or val. That is why we see classes that are just one line with no body

```
class Student(var first: String, var age: Int)
```

Think of how coding you just saved!

If we want our custom getter and setter we can't declare the properties in the constructor but in the class body and provide a custom get() and set(value) functions
field - reference for the actual value

```
class Student(first:String, age:Int) {

    var first = first
        get() = "name is $field"

    var age = if(age>0) age else 0 //for the ctor initialization
    set(value) {
        if(value > 0)
            field = value
    }
}
```

Because of default parameters we usually don't need ctor overloading

```
class Student(var first: String, age: Int = 25) {
```

```
val s1 = Student( first: "Rona")
```

If age is not passed to the ctor then he will get the default value

Please note that If you supply one value, it's used for the first named parameter (you can use parameters name to overcome this) so it generally doesn't make any sense to provide a default value for an early parameter without providing a default for subsequent parameters.

If you're not going to provide default values for all parameters, you should only provide default values for the last parameters in the constructor:

```
class Student(var first: String = "moshe", age: Int) {

    fun main() {
        val s = Student( first: 25)
    }
```

The integer literal does not conform to the expected type String

In this case you need to specify the second parameter by his name

```
val student = Student(age = 23)
```

If for some reason we want the java way or you want a whole new constructor all together you can create another one it is called secondary constructor. We can do it using the **constructor** keyword (in the primary ctor the keyword is deferred). We can delegate to the primary constructor

```
class Student(val first:String, age:Int = 20) {

    var age = if(age>0) age else 0 //when passing negative to the ctor
    set(value) { //when using the setter
        if(value > 0)
            field = value
    }

    constructor(first: String) : this(first, age: 40) {
        //Do something here
    }
}

fun main() {
    val s1 = Student( first: "mika")
    println(s1.age) //40!
```

In this case if we create a Student with name alone he will be 40 and not 20! The system will always look for the exact constructor before applying default values!.

private constructor

To avoid the public constructor in cases such as singletons add the **private constructor()** after the class declaration or inside the class body - better as primary constructor (in the class declaration)

```
class Student private constructor(val first:String, age:Int = 20) {
```

```
val s  = Student( first: "Rona", age: 28)
```

Cannot access '<init>': it is private in 'Student'

lateinit var

Some variables needs to initialize later we can use the **lateinit var**

```
lateinit var last:String
```

In this case the compiler won't show the compilation error of the var not being initialize BUT be careful from accessing the variable - **UninitializedPropertyAccessException** will be thrown at runtime

Only lateinit var exist not val (the lateinit gives it some initial value) and it's not working on all the Java primitives(Int, Double, Char...).

```
lateinit var grade : Int
'lateinit' modifier is not allowed on properties of primitive types
```

Delegated properties

Another way of late initialization is using delegated properties: we will create an object that when we first access it's properties the delegate object is created and store the value computed in the object. This can also work on java primitives.

The syntax is: **val/var <property name>: <Type>** by **<expression>**

The expression after **by** is a *delegate*. The get() and set() corresponding to the property that will be delegated to its getValue() and setValue() methods. Property delegates doesn't have to implement any interface, but they have to provide a getValue() function (and setValue()--- for var s).

If we are want val or one the java primitives to get a later value we can use this delegate:

```
val grade by Delegates.notNull<Int>()
```

But again be careful from accessing it before initialization
Take a look at his code:

```
public object Delegates {

    Returns a property delegate for a read/write property with a non-null value that is initialized r
    during object construction time but at a later time. Trying to read the property before the initia
    value has been assigned results in an exception.

    Samples: samples.properties.Delegates.notNullDelegate
            // Unresolved

    public fun <T : Any> notNull(): ReadWriteProperty<Any?, T> = NotNullVar()
```

```
private class NotNullVar<T : Any>() : ReadWriteProperty<Any?, T> {
    private var value: T? = null

    public override fun getValue(thisRef: Any?, property: KProperty<*>): T {
        return value ?: throw IllegalStateException("Property ${property.name} should be initialized before get.")
    }

    public override fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
        this.value = value
    }
}
```

Another option is using the lazy function
The lazy function will be invoked only when the object id accessed for the fist time

```
val last by lazy {
    println("lazy")
    "Cohen"  ^lazy
}
```

The println is used to show you that the initialization is happening only when first accessed

Another example in using observable delegates
Delegates.observable - Returns a property delegate for a read/write property that calls a specified callback function when changed.

```
class User {
    var name : String by Delegates.observable( initialValue: "moshe") {
            _,old,new ->
        println("$old -> $new")
    }
}

fun main() {
    val user = User()
    user.name = "rona" //outputs: moshe -> rona
    user.name = "dave"//outputs: rona -> dave
```

If you want to intercept assignments and *veto* them, use vetoable() instead of observable(). The handler passed to the vetoable is called *before* the assignment of a new property value.

```
var last by Delegates.vetoable( initialValue: "moshe") {
    _,old,new ->
    println("$old $new")
    false  ^vetoable
}
```

You read more about the concept of Delegates and maybe create your own here:
https://kotlinlang.org/docs/delegated-properties.html
And of course on in this course we will have our own delegate that changes the property value according the the attached fragment lifecycle.

Class extensions

We can add functions to existing classes in Kotlin using the class name before the function name.

Inside the function we have **this**

```
fun Int.isEven() = this % 2 == 0
val x = 4;
println(x.isEven())
```

```
fun Student.isOldEnough() = this.age > 25

val s = Student( first: "mush", age: 89)
println(s.isOldEnough())
```

Data class

```
data class Contact(var name:String,var email:String,var id:Long)
```

Using the keyword **data** we can use Kotlin to create everything needed for a data class - equals, hash-code, toString, copy and more (a Whole file in java is just one line)

```
val moshe = Contact( name: "Moshe Cohen", email: "moshe@moshe.com", id: 1234567)

println(moshe) //calls toString

val moshe1 = Contact( name: "Moshe Cohen", email: "moshe@moshe.com", id: 1234567)

println(moshe == moshe1) //calls equals return true
println(moshe === moshe1) //reference checking  - return false

val moshon = moshe.copy(name = "mush") //copy the object with new name

println(moshon)
```

In Data classes a componentN function is created for each the the properties. We can use this for destructuring declaration

Destructuring declarations

A destructuring declaration creates multiple variables at once(only local variables).

```
val(name,email) = moshe
// same as
val name1 = moshe.component1()
val email1 = moshe.component2()
//the component functions created because of the data class
println("$name $email")
```

If you don't need a variable in the destructuring declaration, you can place an underscore instead of its name:

```
val(name,_,id) = moshe
//the component functions is not called when using _
println("$name $id")
```

The restructuring declaration can also help us in a variety for other things, for example when iterating on a map objects

```
val map  = mapOf(1 to "a",2 to "b")
for ((key,value) in map) {
    println("$key $value")
}
```

They are also used for function calls that we want to return more then one value and no need for the wrapper object

```
    //inside main function cause destructuring declaration only allowed on local variables
    val(result,status) = something()
}

fun something() : Result {
    return Result( result: 4, status: true)
}

data class Result(val result:Int, val status:Boolean)
```

Inheritance

The derived class doesn't need to add val or var to arguments already defined in the parent class.

Inheritance is defined by **:**

All Kotlin classes are final by default! This is mainly because almost no one wrote final in java
When we want to inherit form a class it must be declared as **open**
All the function that we want to override must be **open**

```
open class Shape1(val name:String) {
    open fun area() = 0.0
}

class Circle1(name: String, val radius:Double) : Shape1(name){

    override fun area() = Math.PI * Math.pow(radius,2.0)

    fun fill() = "Circle filled"
}

fun main(args:Array<String>) {

    val s : Shape1 = Circle1( name: "circle", radius: 7.9)
    println(s.area())
    //s.fill() can't find function fill in reference of type Shape
```

In Kotlin we can also override the **class properties** in cases where we want to add a setter for the child or write a different get and set functions.
If the property is val in the parent class in the derived class it can be either var or val - just add a setter - but if it var in parent class it can't be val in the child - we can't vanish the setter it already has one.

This is an error - we can't hide the parent name but we can override it and add a setter and a different default value

```
open class Shape (val name:String = "SHAPE") {}

class Circle(var name:String = "CIRCLE") : Shape(name) {
```

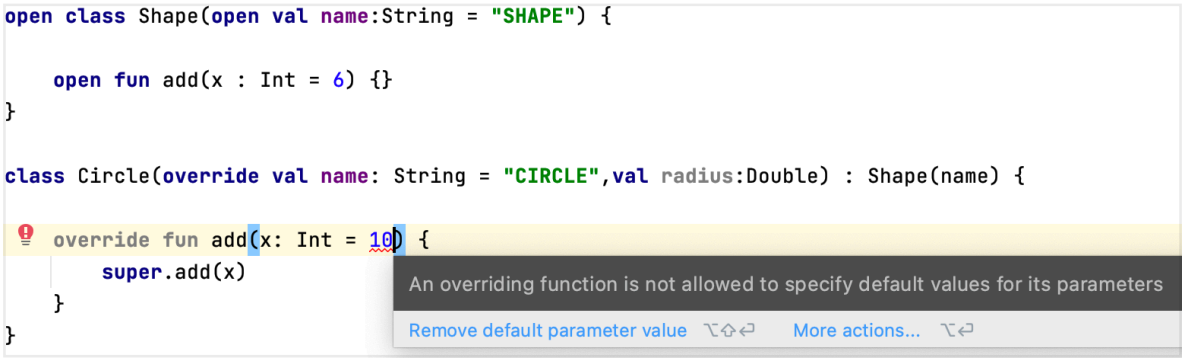
'name' hides member of supertype 'Shape' and needs 'override' modifier

```
open class Shape (open val name:String = "SHAPE") {}

class Circle(override var name:String = "CIRCLE") : Shape(name) {}
```

We can also change the parents getter function:

```
    val shape = Shape()
    println(shape.name)//Shape SHAPE
    val circle = Circle()
    println(circle.name)//Circle CIRCLE
}
open class Shape (name:String = "SHAPE") {
    open val name = name
        get() = "Shape $field"
}

class Circle(name:String = "CIRCLE") : Shape() {
    override var name = name
        get() = "Circle $field"
}
```

Overriding methods always use the same default parameter values as the base method.
When overriding a method that has default parameter values, the default parameter values must be omitted from the signature



It make more sense for Shape to be **abstract** - the area function needs to be abstract
No need for **open** when using the **abstract** keyword same in functions or in classes

```
abstract class Shape1(open val name:String = "SHAPE") {
    abstract fun area() : Double
}
```

Casting(is, as and as?)

Kotlin can smart-cast our objects if we check them before using **is** -Note that smart casts work only when the compiler can guarantee that the variable won't

change between the check and the usage so it always works on val but only on var **local** properties(the compiler can track the local variables)

```
abstract class Shape(open val name:String = "SHAPE") {}

class Circle(override val name: String = "CIRCLE",val radius:Double) : Shape(name){
    fun roll() = println("I roll the circle")
}

fun main(args: Array<String>) {

    val shape:Shape = Circle( name: "Circle 1", radius: 5.6)
    if(shape is Circle) shape.roll()
}
```

If we want to cast the object ourselves we can use the unsafe **as** operator - Usually, the cast operator throws an exception if the cast isn't possible. And so, it's called *unsafe*. The unsafe cast in Kotlin is done by the infix operator **as**.

```
val c = shape as Circle
```

Or if you want to avoid exceptions, use the *safe* cast operator as?, which returns null on failure.

```
val x: String? = y as? String
```

For more reasons adding on Type check and castings https://kotlinlang.org/docs/typecasts.html#type-erasure-and-generic-type-checks

Interfaces

```
class Circle1(val radius:Double,override val name: String = "CIRCLE") : Shape1(name),Fillable1 {

    override fun fill() {} //must
    override fun area() = Math.PI * Math.pow(radius,2.0)
}

interface Fillable1 {
    fun fill()
}
```

Until now it was the same as Java but:
Interfaces in Kotlin can contain properties (without initialization) but when implementing it we must override it

```
class Circle1(val radius:Double,override val name: String = "CIRCLE") : Shape1(name),Fillable1 {
    override val color: String = "Black"
    override fun fill() = "Filling the circle with color $color"
    override fun area() = Math.PI * Math.pow(radius,2.0)
}

interface Fillable1 {
    val color:String
    fun fill():String
}
```

We can also define a default function implementation and in that case we don't have to override it. But that can cause the diamond problem and we solve it using the <[which]>

This is ok

```
class Circle1(val radius:Double,override val name: String = "CIRCLE") : Shape1(name),Fillable1 {
    override val color: String = "Black"
   // override fun fill() = "Filling the circle with color $color"
    override fun area() = Math.PI * Math.pow(radius,2.0)
}

interface Fillable1 {
    val color:String
    fun fill() = "Filling the fillable"
}
```

But if Shape has fill and also there is a default implementation in the interface which fill will be called and we solve it using <[parent]>

```
abstract class Shape1(open val name:String = "SHAPE") {
    abstract fun area() : Double
    open fun fill() = "Filling the shape"
}

class Circle1(val radius:Double,override val name: String = "CIRCLE") : Shape1(name),Fillable1 {
    override val color: String = "Black"
    override fun fill() : String {
        //super.fill() - error!
        println(super<Shape1>.fill())
        println(super<Fillable1>.fill())
        return "Filling the circle with color $color"
    }
    override fun area() = Math.PI * Math.pow(radius,2.0)
}

interface Fillable1 {
    val color:String
    fun fill() = "Filling the fillable"
}

fun main(args:Array<String>) {

    val c = Circle1( radius: 6.7)
    println(c.fill())
```

Object Expressions and Declarations

Object expressions create objects of anonymous classes, that is, classes that aren't explicitly declared with the class declaration - with a specific name. Such classes are useful for one-time use. You can define them from scratch, inherit from existing classes, or implement interfaces. Instances of anonymous classes are also called *anonymous objects* because they are defined by an expression, not a name - **Anonymous inner classes**

```
val w = Window.getWindows()[0]

w.addMouseListener(object : MouseListener {
   //need to implement the mouse listener functions
})
```

Like in java Anonymous class can access outside class members

```
var x = 0;
val w = Window.getWindows()[0]

w.addMouseListener(object : MouseListener {
   //need to implement the mouse listener functions

    override fun mouseClicked(e: MouseEvent?) {
        x++
    }
```

N
When only one function exist in the interface (SAM - Single Abstract Method) we will use Lambda and not object expression(will be discussed later on).

Another simple example :

```
val hello = object  {
    val hello = "Hello"
    val world = "World"
    //object expression also extends Any
    override fun toString(): String {
        return "$hello $world"
    }
}

fun main() {
    println(hello)
}
```

Object declarations
Object declaration always has a name following the **object** keyword. Just like a variable declaration, an object declaration is not an expression, and it cannot be used on the right-hand side of an assignment statement.

The initialization of an object declaration is thread-safe and done on **first access.** To refer to the object, use its name directly.

Kotlin makes it easy to declare singletons using object declaration:

```
    val car  = CarFactory.create("Mazda")
    val car1  = CarFactory.create("Toyota")
    println(car)
    println(car1)
    println(CarFactory.numOfCars)
}

data class Car(val name:String, val price:Int)

object CarFactory {

    var numOfCars = 0

    fun create(name: String) : Car {
        numOfCars++
        return Car(name, price: 10000)
    }
}
```

Please note the because this is an object declaration it has no constructor and there is only one instance so it already a singleton!

```
val c  = CarFactory()
         Unresolved reference.
```

However, this is ok (this is just giving the object a different reference):

```
val fact = CarFactory
println(fact.numOfCars)
```

Such objects can have super-types and we can create a non anonymous single implementation

```
fun main(args: Array<String>) {

    // val c = CarFactory.createCar("Zoe")
    val w = Window.getWindows()[0]
    w.addMouseListener(MyMouseListener)
}

object MyMouseListener : MouseListener {
    override fun mouseClicked(e: MouseEvent?) {
        TODO( reason: "Not yet implemented")
    }
```

Companion objects
An object declaration inside a class can be marked with the companion keyword
This replaces the java statics.
Only one instance of the companion object is created for all instances of the class - it is an object - one per class like the static initializer - when we load the class to the memory for the first time then the companion object is created for all the upcoming instances that will share it.

Please note that it must be an object and inside it we will declare both properties and functions

```
class Student(var first: String = "moshe", age: Int) {

    companion object {
        var numOfStuednts = 0
        fun getAvgStudent()  = Student( first: "dave", age: 25)
    }

    init {
        numOfStuednts++
    }
```

```
println("The average student is ${Student.getAvgStudent()} " +
        "and the num of students is ${Student.numOfStudent}")
```

The default companion object name is Companion but we don't need to specify

it just use the class name, you can also give the companion object a name but that is truly unnecessary

```
companion object Factory{
    var numOfStuednts = 0
    fun getAvgStudent()  = Student( first: "dave", age: 25)
}
```

If you have only one companion object you can still access it with the class name. And since each class is allowed only one companion object the naming is quite unnecessary and use it only if it makes your code more organized.

If you need the companion object itself just use the class name (or if it has a name - his name)

```
val comp = Student
comp.numOfStuednts
```

Note that even though the members of companion objects look like static members in other languages, at runtime those are still instance members of real objects, and can, for example, implement interfaces:

```
interface Factory<T> {
    fun create() : T
}

companion object : Factory<Student>{
    var numOfStuednts = 0
    fun getAvgStudent()  = Student( first: "dave", age: 25)
    override fun create(): Student {
        TODO( reason: "Not yet implemented")
    }
}
```

However, on the JVM you can have members of companion objects generated as real static methods and fields if you use the @JvmStatic annotation. See the Java interoperability section for more detail.

There is one important semantic difference between object expressions and object declarations:

- Object expressions are executed (and initialized) *immediately*, where they are used.
- Object declarations are initialized *lazily*, when accessed for the first

time.
- A companion object is initialized when the corresponding class is loaded (resolved) that matches the semantics of a Java static initializer.

For more reading:
https://kotlinlang.org/docs/object-declarations.html#using-anonymous-objects-as-return-and-value-types

Functional (SAM) interfaces

An interface with only one abstract method is called a *functional interface*, or a *Single Abstract Method (SAM) interface*. The SAM interface can only have one abstract method. To declare SAM interface use the keyword fun before the interface

The main advantage:
Instead of creating a class that implements a functional interface manually, you can use a lambda expression. With a SAM conversion, Kotlin can convert any lambda expression whose signature matches the signature of the interface's single method into the code, o
For example
Take the following interface:

```
fun interface IntPredicate {
    fun accept(i: Int): Boolean
}
```

If you don't use a SAM conversion, you will need to write code like this:
// Creating an instance of a class

```
val isEven = object : IntPredicate {
    override fun accept(i: Int): Boolean {
        return i % 2 == 0
    }
}
```

By leveraging Kotlin's SAM conversion, you can write the following equivalent code instead:
// Creating an instance using lambda

```
val isEvenSAM = IntPredicate { it % 2 == 0 }
```

And in main function:

```
println(isEvenSAM.accept( i: 7))
```

You will use this allot in Android programming just think of the OnClickListener interface, isn't it SAM?

Nested and inner classes

In Kotlin like in Java we can nest a class within another class (this is simply a structural thing):

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

Note that Nested can't access bar.

You can make a nested class Inner using the **inner** keyword before the class declaration.
A nested class marked as inner can access the members of its outer class!

Inner classes carry a reference to an object of an outer class. In nested class we don't need to create an instance of the outer class just use it's name (like we said before it's a structural thing). But if it is an inner class we must create an instance of the outer class to get and instance of the Inner class:

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}
val demo = Outer().Inner().foo() // == 1
```

Access Modifiers

private - same as java
public - the default in Kotlin!
protected - same as java
internal - equivalent to java package level - same Module - a set of Kotlin files compiling together - In android same Gradle or Maven

We can use **as** keyword for direct name for imported classes

```
import java.awt.Window as W
import java.awt.event.MouseEvent
import java.awt.event.MouseListener

fun main(args: Array<String>) {

    // val c = CarFactory.createCar("Zoe")
     val w = W.getWindows()[0]
```

Generics - covariance and contravarince

In case we want to define a generic class and later narrow the generic type (we used ? Extend Object in java) this is a problem since we set a more specific type of the generic, and we can add things at compile time that will crash at runtime!

```
class Source<T>()

fun main(args: Array<String>) {

    val source:Source<Any> = Source<String>()

    // val sources1:Source<String> = S
}
```

Type mismatch.
Required: Source<Any>
Found: Source<String>

The problem is that in the reference he accepts wider objects then in runtime

We can use the **out** keyword next to the generic to specify the T will only be used as return value - and the problem solved - covariance - because if it will be used only as a return value then the user won't be able to cause the problem mentioned before.

```
class Source<out T>()

fun main(args: Array<String>) {

    val source:Source<Any> = Source<String>()
```

Same in the opposite direction - we can specify the keyword **in** for generics that will only we used as parameter - think of a getter that suppose to return String but actually return Any - this is a problem! but think of a setter that suppose to get String and get Any - this is no problem - contra variance

```
class Source<in T>()

fun main(args: Array<String>) {

    val source:Source<String> = Source<Any>()
```

Its also possible to define both:

```
class Source<in T, out E>()

fun main(args: Array<String>) {

    val source:Source<String,Any> = Source<Any,String>()
```

sealed

Sealed classes and interfaces represent restricted class hierarchies that provide more control over inheritance. All direct subclasses of a sealed class are known at compile time. No other subclasses may appear after a module with the sealed class is compiled. For example, third-party clients can't extend your sealed class in their code. Thus, each instance of a sealed class has a type from a limited set that is known when this class is compiled.
This is very useful when checking instances of a curtain class with when() because first if we choked all known subclasses then we don't need else and more then that the compiler warnings that tells us that we forget to check a curtain subclass can save us allot debugging time.

A sealed class is abstract by itself, it cannot be instantiated directly and can have abstract members.
Direct subclasses of sealed classes and interfaces must be declared in the same package.

https://kotlinlang.org/docs/sealed-classes.html#sealed-classes-and-when-expression