

Android Kotlin - Intro

<https://kotlinlang.org/docs/home.html>

Lecturer: Yehuda Rozilyo

Var declaration

val - parameter that never changes

```
var count: Int = 10
```

var - value can change

```
var count: Int = 10  
count = 15
```

Var declaration

```
val languageName = "Kotlin"  
val upperCaseName = languageName.toUpperCase()  
  
// Fails to compile  
languageName.inc()
```

If type is not provided, kotlin automatically assign a type to the argument according to the value given.

languageName is a String, we can call toUpperCase func on it

We cannot call inc() its Int function

Null value

```
val languageName: String? = null
```

The “?” states that this argument can have a null value - nullable

```
val accountName = account.name!!.trim()
```

!! op will force getting the value

If its null we will get exception

```
val accountName = account.name?.trim()
```

? op will conditionally get the value

If its null it will stop the operation and return null

?: op means - if the left is null return the value on the right

```
val accountName = account.name?.trim() ?: "Default name"
```

Classes & Constructors

A class in Kotlin can have a primary constructor and one or more secondary constructors.

The primary constructor is a part of the class header, and it goes after the class name and optional type parameters.

```
class Person constructor(firstName: String) { /*...*/ }
```

If the primary constructor does not have any annotations or visibility modifiers, the constructor keyword can be omitted:

```
class Person(firstName: String) { /*...*/ }
```

initializers

The primary constructor cannot contain any code. Initialization code can be placed in initializer blocks prefixed with the `init` keyword.

The initializer blocks are executed in the same order as they appear in the class body, interleaved with the property initializers:

```
class InitOrderDemo(name: String) {  
    val firstProperty = "First property: $name".also(::println)  
  
    init {  
        println("First initializer block that prints ${name}")  
    }  
  
    val secondProperty = "Second property: ${name.length}".also(::println)  
  
    init {  
        println("Second initializer block that prints ${name.length}")  
    }  
}
```

Constructors

If the constructor has annotations or visibility modifiers, the constructor keyword is required and the modifiers go before it:

```
class Customer public @Inject constructor(name: String) { /*...*/ }
```

```
class DontCreateMe private constructor () { /*...*/ }
```

Secondary Constructors

```
class Person(val pets: MutableList<Pet> = mutableListOf())

class Pet {
    constructor(owner: Person) {
        owner.pets.add(this) // adds this pet to the list of its owner's pets
    }
}
```

If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor, either directly or indirectly through another secondary constructor(s)

```
class Person(val name: String) {
    var children: MutableList<Person> = mutableListOf()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```


Secondary Constructor

- Code in initializer blocks effectively becomes part of the primary constructor.
- The code in all initializer blocks and property initializers is executed before the body of the secondary constructor.

Open class

open keyword means “open for extension”

The open annotation on a class is the opposite of Java's final : it allows others to inherit from this class.

By default, all classes in Kotlin are final

Abstract class

```
abstract class Polygon {  
    abstract fun draw()  
}  
  
class Rectangle : Polygon() {  
    override fun draw() {  
        // draw the rectangle  
    }  
}
```

Instances

There is no need to use “new”

```
val invoice = Invoice()
```

```
val customer = Customer("Joe Smith")
```

Inheritance

Inheritance

All classes in Kotlin have a common superclass, **Any** (same as Object in Java)

Any has three methods: equals(), hashCode(), and toString().

By default, Kotlin classes are final – they can't be inherited. To make a class inheritable, mark it with the open keyword.

If the derived class has a primary constructor, the base class must be initialized in that primary constructor.

```
open class Base(p: Int)
```

```
class Derived(p: Int) : Base(p)
```

Inheritance

If the derived class has no primary constructor, then each secondary constructor has to initialize the base type using the `super` keyword or it has to delegate to another constructor which does.

```
class MyView : View {  
    constructor(ctx: Context) : super(ctx)  
  
    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)  
}
```

Getters and Setters

The full syntax for declaring a property is as follows:

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

```
var stringRepresentation: String  
    get() = this.toString()  
    set(value) {  
        setDataFromString(value)  
    }
```

```
var setterVisibility: String = "abc"  
    private set // the setter is private and has the default implementation
```


Backing Field

```
var counter = 0 // the initializer assigns the backing field directly
set(value) {
    if (value >= 0)
        field = value
        // counter = value // ERROR StackOverflow: Using actual name
}
```

field is only used as a part of a property to hold its value in memory.


fields cannot be declared directly.

when a property needs a backing field, Kotlin provides it automatically.

This backing field can be referenced in the accessors using the field identifier

field - same as doing....

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null) {
            _table = HashMap() // Type parameters are inferred
        }
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```



Scope Functions

— — —

- The Kotlin standard library contains several functions whose sole purpose is to execute a block of code within the context of an object.
- When you call such a function on an object with a **lambda expression** provided, it forms a temporary scope.
- In this scope, you can access the object without its name.
- Such functions are called scope functions. There are five of them: `let`, `run`, `with`, `apply`, and `also`.

Person("Alice", 20, "Amsterdam").let {
 println(it)
 it.moveTo("London")
 it.incrementAge()
 println(it)
}

Same as:

```
val alice = Person("Alice", 20, "Amsterdam")  
println(alice)  
alice.moveTo("London")  
alice.incrementAge()  
println(alice)
```

let, run, with, apply, also

— — —

Function	Object reference	Return value	Is extension function
<code>let</code>	<code>it</code>	Lambda result	Yes
<code>run</code>	<code>this</code>	Lambda result	Yes
<code>run</code>	-	Lambda result	No: called without the context object
<code>with</code>	<code>this</code>	Lambda result	No: takes the context object as an argument.
<code>apply</code>	<code>this</code>	Context object	Yes
<code>also</code>	<code>it</code>	Context object	Yes

Extensions

Extensions are the ability to extend a class with new functionality without having to inherit.

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {  
    val tmp = this[index1] // 'this' corresponds to the list  
    this[index1] = this[index2]  
    this[index2] = tmp  
}
```

Extensions

- — — ● Extensions do not actually modify the classes they extend.
- Extension adds a new functions callable with the dot-notation on variables of this type.
- Extensions are not virtual
- An extension function being called is determined by the type of the expression on which the function is invoked.

```
open class Shape
class Rectangle: Shape()

fun Shape.getName() = "Shape"
fun Rectangle.getName() = "Rectangle"

fun printClassName(s: Shape) {
    println(s.getName())
}

printClassName(Rectangle())
```

Object Expressions

Object expressions create objects of anonymous classes.

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.

Creating anonymous objects from scratch

```
val helloWorld = object {  
    val hello = "Hello"  
    val world = "World"  
    // object expressions extend Any, so `override` is not needed  
    override fun toString() = "$hello $world"  
}
```

Singleton

Companion Object

Object that is declared as part of a class

It is created once

it is shared with all other objects of that class

Similar to static member

Functions of companion object are also similar to static functions

Singleton using Companion Object

Private constructor

Companion object holds the single instance of the class

```
//Singleton
class Model private constructor(){
    companion object {
        val instance = Model()
    }
}
```