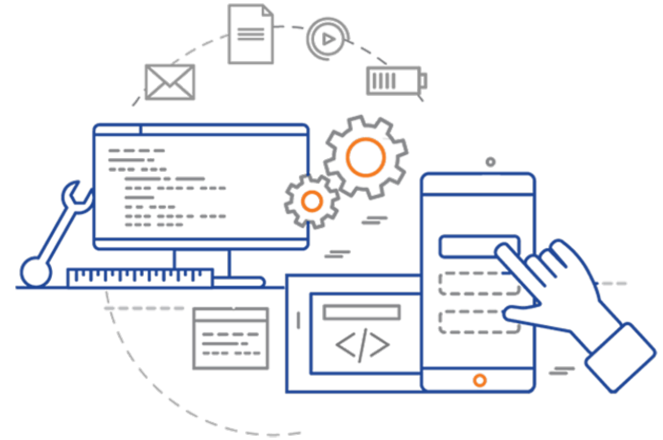
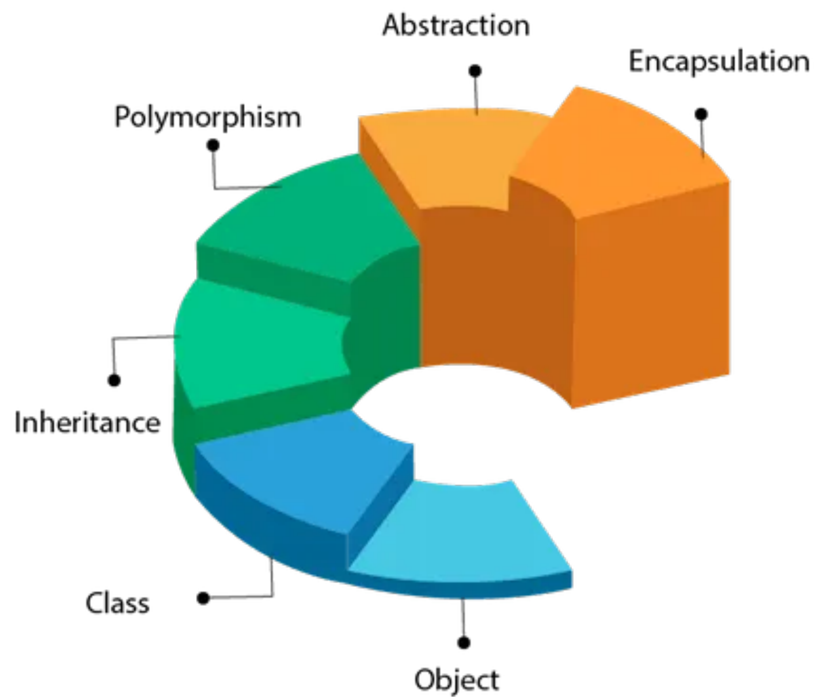


CS 398: Application Development

Week 04 Video: OO Kotlin

OOPs (Object-Oriented Programming System)



As an object-oriented language, Kotlin supports these foundational principles. Kotlin is most similar to Java, but has a number of enhancements over that language.

Classes & Objects

Kotlin is a class-based object-oriented language, with some advanced features that it shares with some other modern languages.

The 'class' keyword denote a class.

```
class Person
val p = Person()
```

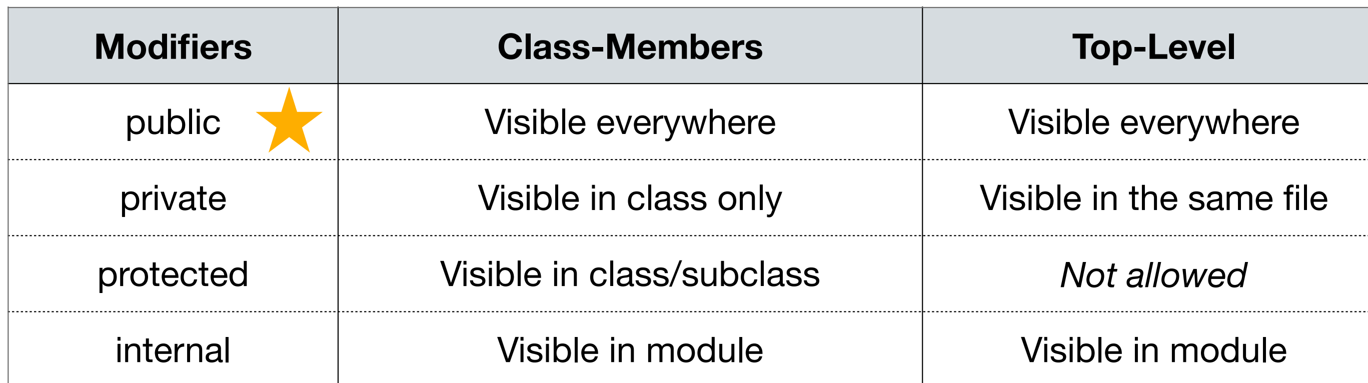
val or var can be applied to classes as well (which indicates whether p is read-only or can be reassigned).

Classes can contains **methods** (aka class functions), and **properties** (to store data).

Sidebar: Visibility Modifiers

Annotations or visibility modifiers go before the constructor or function name.

Kotlin defaults to “public” visibility if you omit the modifier (which we will often do in examples).



Modifiers	Class-Members	Top-Level
public ★	Visible everywhere	Visible everywhere
private	Visible in class only	Visible in the same file
protected	Visible in class/subclass	*Not allowed*
internal	Visible in module	Visible in module

Properties

A property is a variable that is declared in a class, but outside of methods or functions. They are analogous to class members, or fields in other languages.

```
class Person() {
   // string properties
   var lastName = "Vanilla"
   var firstName = "Ice"
 }

val p = Person()

// we can access properties directly
// this calls an implicit get() method; by default this just returns the value
println("${p.firstName} ${p.lastName}”)

> Vanilla Ice
```

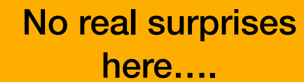
Methods

Methods are just functions contained within a class, that are associated with an instance of that class. You invoke then using the dot operator on an object (class instance).

```
class Person() {
    fun talk() {
        println("I am a human being!")
    }
}

val p = Person()
p.talk()

> I am a human being!
```



Constructors

The **primary constructor** is part of the class declaration.

```
// class with a no-arg constructor (all these are equivalent)
class Person1
class Person1 { }
class Person1 constructor () { }

// parameters can be passed in, but will not persist unless saved
class Person2(firstName:String, lastName:String)

// val or var can be used to denote mutability
// these will also force these parameters to be exposed as properties
class Person3(val firstName: String, val lastName: String, var age: Int)

val student = Person3("Sally", "Zhang")
println("${student3.firstName} ${student3.lastName}")
// Sally Zhang
```

Classes may have secondary constructors, that delegate to the primary constructor:

```
// primary constructor
class Person(val name: String) {

  var children: MutableList<Person> = mutableListOf<>()

  // secondary constructor
  constructor(name: String, parent: Person) : this(name) {
    parent.children.add(this)
  }
}
```

The primary constructor cannot do anything more than initialize properties!

```
class InitOrderDemo(name: String) {
    val first = "$name".uppercase()
    init {
        println("First init: $first")
    }

    val second = "${name.length}"
    init {
        println("Second init: $second")
    }
}

fun main() {
    val a = InitOrderDemo("Jeff")
}

// First init: JEFF
// Second init: 4
```

Any more complex code here will not compile.

Initialization code is considered part of the primary constructor. The order of initialization is (1) primary constructor, (2) init blocks in listed order, and then (3) secondary constructor (if appropriate).

```
class InitOrderDemo(name: String) {
    val first = "$name".uppercase()
    init {
        println("First init: $first")
    }

    val second = "${name.length}"
    init {
        println("Second init: $second")
    }
}

fun main() {
    val a = InitOrderDemo("Jeff")
}

// First init: JEFF
// Second init: 4
```

primary constructor, property assignment — 1

init block called — 2

init block called — 3

start here — 0

With a no-arg constructor, I’d recommend using *that* as the primary.

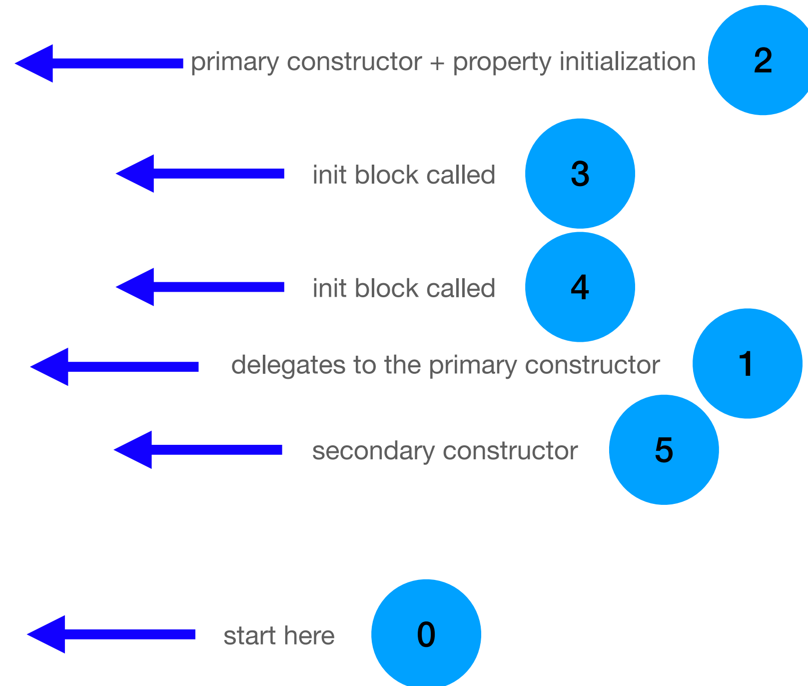
```
class InitOrderDemo() {
    var name:String = "Default"
    val first = "$name".uppercase()
    init { println("First init: $first”) }

    val second = "${name.length}"
    init { println("Second init: $second”) }

    constructor(name:String) : this() {
        this.name = name
        println("Second constructor: $name”)
    }
}

fun main() {
    val a = InitOrderDemo("Jeff")
}

// First init: DEFAULT
// Second init: 7
// Second constructor: Jeff
```



Arguments to the primary constructor are **properties**: internal fields with the same name as the arguments. Accessor methods are automatically created for them.

```
// name and age will be created as properties b/c of the val and var keywords.
class Person(val name: String, var age: Int)

val jeff = Person("Jeff", 50)
println("${jeff.name} is ${jeff.age}") // ok
jeff.name = "Jeffery" // error, since its immutable
jeff.age = 49 // ok, since it’s mutable

// we can override implicit getters and setters as well
class Person(var name: String, var surname: String) {
  var fullName: String
  get() = "$name $surname"
  set(value) {
   (name, surname) = value.split(" ")
  }
}
```

Inheritance

Kotlin supports a single-inheritance model. Although you can inherit from any number of interfaces, you cannot inherit from more than one implementation class.

By default, classes and methods are ‘closed‘ to inheritance. If you want to extend a class or method, you need to mark it as ‘open‘ for inheritance.

```
open class Person(val name: String) {
 open fun hello() = "Hello, I am $name"
}

class PolishPerson(name: String) : Person(name) {
 override fun hello() = "Dzien dobry, jestem $name"
}
```

Interfaces

An interface is a list of methods that together describe a set of expected behaviours for a class, without any specific implementation being mandated.

It's like an abstract class, that cannot be instantiated (and has no implementation). This means that you **must** override interface functions.

```
interface Shape {
    fun dimensions(w: Double, h: Double)
    fun area(): Double
}

class Rectangle : Shape {
    var width: Double = 0.0
    var height: Double = 0.0

    override fun dimensions(w: Double, h: Double) { width = w; height = h }
    override fun area(): Double { return width * height }
}
```

Abstract Class

An abstract class is meant to be a base or parent class in a class hierarchy. It can contain implementation code, but cannot be instantiated directly: it can only serve as a base class.

```
abstract class Shape(var width: Double, var height: Double) {
    fun dimensions(w: Double, h: Double) { width = w; height = h }
    abstract val area: Double
}

class Rectangle(width: Double, height: Double) : Shape(width, height) {
    override val area: Double
        get() = width * height
}

fun main() {
    val rect = Rectangle(10.0,20.0)
    print(rect.area) // 200.0
}
```

Note that we don't have to override the dimensions() function, but we <u>do</u> have to override the area which is abstract.

Data Class

A data class is a special type of class, which primarily exists to hold data, and doesn't have custom methods. Classes like this are more common than you expect – we often create trivial classes to just hold data, and Kotlin makes it very easy.

Data classes have a number of built-in features:

```
data class Person(val name: String, var age: Int)
val mike = Person("Mike", 23)

// toString() displays all properties
print(mike.toString()) // Person(name=Mike, age=23)

// equals that compares all properties
print(mike == Person("Mike", 23)) // True
print(mike == Person("Mike", 21)) // False
```

```kotlin
// hashCode based on primary constructor properties
val hash = mike.hashCode()
print(hash == Person("Mike", 23).hashCode()) // T
print(hash == Person("Mike", 21).hashCode()) // F

// deconstruction based on properties
val (name, age) = mike
print("$name $age") // Mike 23

// copy that returns a copy of the object
// with concrete properties changed
val jake = mike.copy(name = "Jake") // copy
```

Enum classes

Enums in Kotlin are classes, so enum classes support type safety. This means that we can use them as expected, but we can also use them in new ways, like in a ‘when’ expression.

```
enum class Suits {
    HEARTS, SPADES, DIAMONDS, CLUBS
}

val suit = Suits.SPADES
val color = when(suit) {
  Suits.HEARTS, Suits.DIAMONDS -> "red"
  Suits.SPADES, Suits.CLUBS -> "black"
}
println(color)
// black
```

Advanced Topics

Extension functions

Add a method to an already existing class.

```
fun Int.isEven() = this % 2 == 0
> 5.isEven()
```

Infix functions

Special functions that can be called using the infix notation (omitting the dot and the parentheses for the call).

```
infix fun Int.shl(x: Int): Int { ... }
> 1 shl 2
```