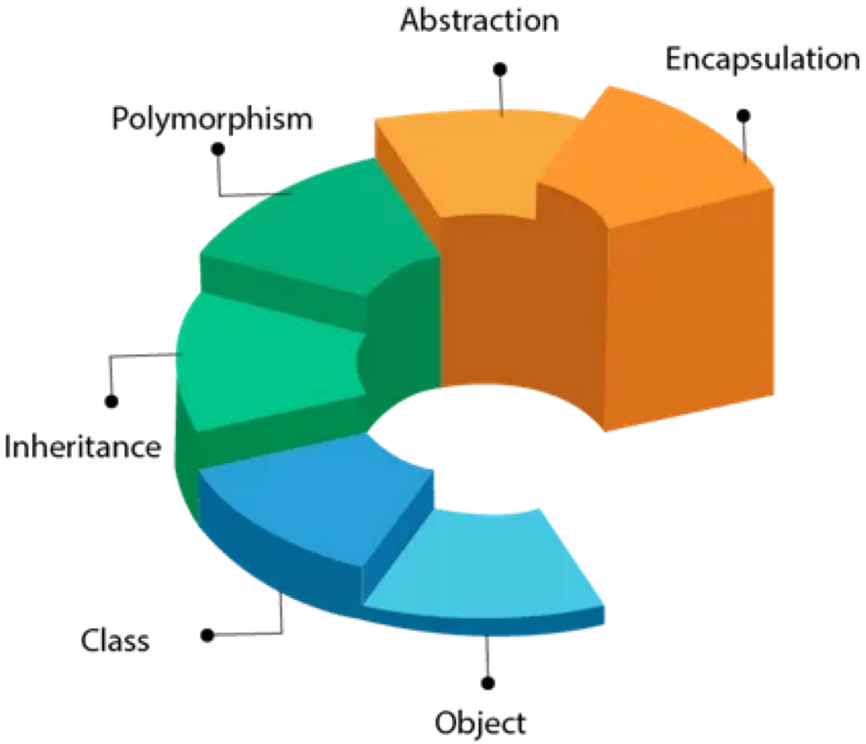
Kotlin Part 2: Object-Oriented Kotlin

CS 346 Application Development

OOPs (Object-Oriented Programming System)



As an object-oriented language, Kotlin supports these foundational principles. Kotlin is most like Java but has enhancements compared to that language.

Classes

Kotlin is a class-based object-oriented language.

- The class keyword denote a class definition:

```
class Person
val p = Person() // there is no `new` keyword required
```

- val or var can be applied to a variable to indicate whether the variable can be reassigned (i.e., the object it points to can still mutate!).
- Classes contain **methods** (aka class functions), and **properties** (aka fields to store data).

Sidebar: Visibility Modifiers

- Annotations or visibility modifiers go before the constructor, property 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 (default)	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

Constructors

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

Different but equivalent ways to define a constructor:

```
class Person1
class Person1()
class Person1() { }
class Person1 { }
```

These all define a single no-arg constructor as part of the class definition.

```
class Person1
fun main() {
  val student1 = Person1() // no properties or methods :/
}
```

Properties

A property is a variable that is declared at the class level. They are analogous to class `members`, or `fields` in other languages.

```
class Person {
    var firstName = "Sally"
    var lastName = "Fields"
 }

 fun main() {
     val p = Person()
     println("Full name is ${p.firstName} ${p.lastName}")
 }
```

https://pl.kotl.in/5vPXtuia8

Creating Properties from Parameters

```
fun main() {
  class Person2(name: String, age: Int)
  val student2 = Person2("Sally", 25) // parameters passed in but not saved
  println("${student3.name}") // unresolved reference!

  // val or var will create the associated properties for these parameters!
  class Person3(val name: String, var age: Int)
  val student3 = Person3("Sally", 25) // parameters turn into public properties
  println("${student3.name} is ${student3.age}") // “Sally is 25”
}
```

https://pl.kotl.in/x3V0tNzQv

Methods

Methods are just functions associated with a class.
Invoke then using the dot operator on an object (i.e. class instance).

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

fun main() {
    val p = Person()
    p.talk() // “I am a human being!”
}
```

https://pl.kotl.in/n0Hm9jVLP

Primary Constructor

Note that you only have the constructors you define.

- There are no automatically created constructors!
- Adding parameters here replaces the no-arg primary constructor with a parameterized constructor

```
class Person1(val name: String, val age: Int)

fun main() {
  // will fail to compile, we do not have a no-arg constructor
  // val p1 = Person1()

  val p2 = Person1("Jeff", 35) // this works
  println("${p2.name} is ${p2.age} years old")
}
```

https://pl.kotl.in/UWnr3AORH

Multiple Constructors

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

fun main()  {
    val parent = Person("Mary") // primary constructor invoked
    val child1 = Person("Cameron", parent) // secondary constructor invoked
    val child2 = Person("Sally", parent)
    for (child in parent.children) {
        println(child.name)
    }
}
```

https://pl.kotl.in/Yj9mDih8h

What can constructors do?

Primary constructors are only meant to initialize properties!

Use `init` blocks to contain other code that must run.

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

val second = "${name.length}"
  init {
    println("Second: $second")
  }
}
val a = InitOrderDemo("Jeff") // First: Jeff, Second: 4
```

https://pl.kotl.in/cqKy6V1WL

Constructor Delegation

```
class InitOrderDemo() { // 3

  var name:String = "Default"
  val first = "$name".uppercase()

  init { // 4
    println("First init: $first")
  }

  constructor(name: String) : this() { // 2
    this.name = name // 5
    println("Second constructor: $name") // can do in a secondary constructor
  }
}
val a = InitOrderDemo("Jeff") // 1
```

https://pl.kotl.in/M8kJljKX6

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.

Why this design?

```
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. ***Classes that implement an interface promise to implement these methods.***

```
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 }
}
```

https://pl.kotl.in/wdHN9HHqq

Abstract Classes

An abstract class is meant to be a base or parent class in a class hierarchy. Unlike an interface, **abstract classes can have constructors, fields and default implementations**.

```
abstract class Shape(var width: Double, var height: Double) {
    fun dimensions(w: Double, h: Double) { width = w; height = h; // more code … }
    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
}
```

We don't have to override the dimensions() function, but we do have to override the area which is abstract. We also override the getter to calculate the area!

https://pl.kotl.in/OGdn3CsGX

Data Classes

A data class is a special type of class which primarily exists to hold data. 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 provide 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 properties (value equality by default!)
print(mike == Person("Mike", 23)) // True
print(mike == Person("Mike", 21)) // False
```

Data Classes

```
// 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")
```

https://pl.kotl.in/Yx5YTC2k_

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
```

https://pl.kotl.in/1nCOM_D59

Cool Stuff

Extension functions

Add a method to an already existing class.

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

Operator overloading

We can overload standard operators (but not arbitrary ones!)

```
data class Point(val x: Int, val y: Int)
operator fun Point.unaryMinus() = Point(-x, -y)

val point = Point(10, 20)
println(-point) // "Point(x=-10, y=-20)"
```

https://pl.kotl.in/PZ_uWl8KI

Reference

- Dave Leeds. 2025. Dave Leeds on Kotlin. Online.
- Dave Leeds. 2025. **Kotlin: An Illustrated Guide**. TypeAlias Studios LLC. ISBN 979-8992796605.
- JetBrains. 2025. Kotlin Documentation. Online.
- Roman Elizarov, et al. 2024. **Kotlin in Action**. 2nd edition. Manning Publications. ISBN 9781617299605.