Object-Oriented Programming in Kotlin

Prepared by

Amit Sen

Senior Software Engineer

Welldev Bangladesh Ltd.

Website: http://amiit.me/

Linkedin: https://www.linkedin.com/in/amit-sen-bb156428/

Contents

Object Oriented Programming

- Over time, data abstraction has become essential as programs became complicated.

- Benefits:
    1. Simplicity (OOP models the real world)
    2. Modularity (An Object is modular)
    3. Modifiability (Objects are easily modifiable)
    4. Extensibility (Objects are extensible)
    5. Reusability (Objects can be reused)

Object Oriented Programming

- Four major principles:

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Data Abstraction

Will describe these using Kotlin, because ...

More about OOP

- Everything is an object.
- Objects communicate by sending and receiving messages. How??
- Objects have their own memory.
- Every object is an instance of a class.
- The class holds the shared behavior for its instance.

** Kotlin provides full support for the points above.

Classes

- Classes are the main building blocks of any OOPL.
- The concept of a class was first studied by Aristotle.
- All objects, despite being unique, are part of a class and share common behavior
- Think of a class as a blueprint; it describes the data and the behavior of a type.
- In Kotlin, a class is declared as the following,

```
class Deposit {
}
```

Classes ...

```
class Person constructor(val firstName: String, val lastName:  String, val age:
Int?) {
}

fun main(args: Array<String>) {
    val person1 = Person("Alex", "Smith", 29)
    val person2 = Person("Jane", "Smith", null)
    println("${person1.firstName},${person1.lastName} is  ${person1.age} years old")
    println("${person2.firstName},${person2.lastName} is $
{person2.age?.toString() ?: "?"} years old")
}
```

- Where is the new keyword??!!
- what's that constructor keyword!!??
- How can I code inside the constructor??

Constructors

```
class Person(val firstName: String, val lastName: String, val age: Int?) {
    init{
        require(firstName.trim().length > 0) {
            "Invalid firstName argument."
        }
        require(lastName.trim().length > 0) {
            "Invalid lastName  argument."
        }
        if (age != null) {
            require(age >= 0 && age < 150) {
              "Invalid age argument."
            }
        }
    }
}
```

- Validates incoming parameters
- The require method will throw IllegalArgumentException

Access Levels

Kotlin comes with four different access levels:

1. **Internal**
2. **Public**
3. **Private**
4. **Protected**

*** What are the differences between all these access levels?

Nested Classes: Java

In Java, nested classes come in two flavors: static and non-static.

```
class Outer {
    static class StaticNested {} // Static class
    class Inner {} // Inner Class
}
```

- Subtle difference between static and inner nested classes.
- The inner nested classes have access to the enclosing class members even if they are declared private.
- The static nested classes can access the public members only.
- To create an instance of the inner class, you will first need an instance of an outer class.

Static Nested Classes: Kotlin

```
class BasicGraph(val name: String) {
    class Line(val x1: Int, val y1: Int, val x2: Int, val y2:  Int) {
        fun draw(): Unit {
            println("Drawing Line from ($x1, $y1) to ($x2, $y2)")
        }
    }
    fun draw(): Unit {
        println("Drawing the graph $name")
    }
}
val line = BasicGraph.Line(1, 0, -2, 0)
line.draw()
```

The example is pretty straightforward and shows you how it works.

Inner Nested Class: Kotlin

- To allow the Line class to access a private member of the outer class BasicGraph, all you need to do is make the Line class inner; just prefix the class with the inner keyword.

```
class BasicGraphWithInner(graphName: String) {
    private val name: String
    init {
        name = graphName
    }
    inner class InnerLine(val x1: Int, val y1: Int, val x2: Int,  val y2: Int) {
        fun draw(): Unit {
            println("Drawing Line from ($x1, $y1) to ($x2, $y2) for  graph $name ")
        }
    }
    fun draw(): Unit {
        println("Drawing the graph $name")
    }
}
```

Power of *this*

- Kotlin comes with a more powerful this expression than you may be accustomed with.
- You can refer the outer scope to this by using the label construct this@label.

```
class A {
    private val somefield: Int = 1
    inner class B {
        private val somefield: Int = 1
        fun foo(s: String) {
            println("Field <somefield> from B" + this.somefield)
            println("Field <somefield> from B" + this@B.somefield)
            println("Field <somefield> from A" + this@A.somefield)
        }
    }
}
```

Data Classes

```
data class User(var name: String, val age: Int)
```

- Classes to create data models which hold the data.
- The primary constructor needs to have at least one parameter.
- Data classes cannot be abstract, open, sealed or inner.
- Data classes may only implement interfaces.

- To exclude a property from the generated implementations, declare it inside the class body

```
data class Person(val name: String) {
    var age: Int = 0
}
```

Data Class Properties …

- equals()/hashCode() pair
- toString() of the form "User(name=John, age=42)”
- copy() function

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

- componentN() functions corresponding to the properties in their order of declaration

```
val (name, age) = person
```

  - This syntax is called a *destructuring declaration*. A destructuring declaration creates multiple variables at once.

Enum Classes

- To define an enumeration, you could use the enum class keywords

```
enum class Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY,  FRIDAY, SATURDAY, SUNDAY
}
```

- Enumeration, like all classes, can take a constructor parameter.

```
public enum class Planet(val mass: Double, val radius: Double) {
    MERCURY(3.303e+23, 2.4397e6), VENUS(4.869e+24, 6.0518e6), EARTH(5.976e+24,
6.37814e6), MARS(6.421e+23, 3.3972e6), JUPITER(1.9e+27, 7.1492e7), SATURN(5.688e+26,
6.0268e7), URANUS(8.686e+25, 2.5559e7), NEPTUNE(1.024e+26, 2.4746e7)
}

Planet.valueOf(“JUPITER") // Like Java you an access the value of an enum item
Planet.values() // It will return you a list of values
```

Enum with Interfaces

```
interface Printable {
    fun print(): Unit
}

public enum class Word : Printable {
    HELLO {
        override fun print() {
             println("Word is HELLO")
        }
    },
    BYE {
        override fun print() {
            println("Word is BYE")
        }
    }
}

val w = Word.HELLO
w.print()
```

Singleton using Java

```
public final class SomeSingleton {
   public static final SomeSingleton INSTANCE;

   private SomeSingleton() {
      INSTANCE = (SomeSingleton) this;
      System.out.println("init complete");
   }

   static {
      new SomeSingleton();
   }
}
```

- This is the preferred way to implement singletons on a JVM because it enables thread-safe lazy initialization without having to rely on a locking algorithm.

** What is a locking algorithm?

Singleton using Kotlin

- In Kotlin, A singleton is created by simply declaring an object.

```
object SomeSingleton
```

- Contrary to a class, an object can’t have any constructor, but init blocks are allowed if some initialization code is needed.

```
object SomeSingleton {
    init {
        println("init complete")
    }

    var b: String? = null
}

var first = SomeSingleton
first.b = "hello singleton"

var second = SomeSingleton
println(second.b)   // hello singleton
```

Interfaces

- An interface is nothing more than a contract; it contains definitions for a set of related functionalities.
- The implementer of the interface has to adhere to the interface the contract and implement the required methods.
- Just like Java 8, a Kotlin interface contains the declarations of abstract methods as well as method implementations.

Interfaces ...

```
interface Document {
    val version: Long
    val size: Long

    val name: String
        get() = “NoName" // A new thing??!!

    fun save(input: InputStream)
    fun load(stream: OutputStream)
    fun getDescription(): String {
        return "Document $name has $size byte(-s)"
    }
}
```

- This interface defines three properties and three methods; the name property and the getDescription methods provide the default implementation.

*** How would you use the interface from a Java class?

Implement Interface in Java

```
public class MyDocument implements Document {
    public long getVersion() {
        return 0;
    }

    public long getSize() {
        return 0;
    }

    public void save(@NotNull InputStream input) {
    }

    public void load(@NotNull OutputStream stream) {
    }

    public String getName() {
        return null;
    }

    public String getDescription() {
        return null;
    }
}
```

Implement Interface in Kotlin

```
class DocumentImpl : Document {
    override val size: Long
        get() = 0

    override val version: Long
        get() = 0

    override fun load(stream: OutputStream) {
    }

    override fun save(input: InputStream) {
    }
}
```

Inheritance

- All classes in Kotlin have a common superclass Any, that is the default superclass for a class with no supertypes declared.
- To declare an explicit supertype, we place the type after a colon in the class header:

```
open class Base(p: Int)

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

- 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.

Overriding Methods

- Unlike Java, Kotlin requires explicit annotations for overridable members (we call them open).

```
open class Base {
    open fun v() {}
    fun nv() {}
}

class Derived(): Base() {
    override fun v() {}
}
```

- The override annotation is required for Derived.v().
- open annotation on a function to be overridden is required.
- A member marked override is itself open, i.e. it may be overridden in subclasses. If you want to prohibit re-overriding, use final:

```
open class AnotherDerived() : Base() {
    final override fun v() {}
}
```

Overriding Properties

```
open class Foo {
    open val x: Int get() { ... }
}

class Bar1 : Foo() {
    override val x: Int = ...
}
```

- You can override a val property with a var property, but not vice versa.
  - This is allowed because a val property essentially declares a getter method, and overriding it as a var additionally declares a setter method in the derived class.

```
interface Foo {
    val count: Int
}

class Bar1(override val count: Int) : Foo

class Bar2 : Foo {
    override var count: Int = 0
}
```

Superclass Implementations

```
open class Foo {
    open fun f() {
        println("Foo.f()")
    }
    open val x: Int get() = 1
}

class Bar : Foo() {
    override fun f() {
        super.f()
        println("Bar.f()")
    }

    override val x: Int get() = super.x + 1
}
```

Companion Objects

- An object declaration inside a class can be marked with the companion keyword:

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

- Members of the companion object can be called by using simply the class name as the qualifier:

```
val instance = MyClass.create()
```

- The name of the companion object can be omitted, in which case the name Companion will be used:

```
class MyClass {
    companion object {
    }
}

val x = MyClass.Companion
```

Association

- One of the compelling features of an OOP language is code reuse.
- The concept of building up a brand new class by reusing existing ones is called association.
- This term is referred to as a has-a relationship.

- Association comes in two flavors. This detail is most of the time overlooked.
  1. Aggregation
  2. Composition

Aggregation

- A relationship between two or more objects.
- Each object has its own life cycle
- Basically, the objects can be created and destroyed independently.
- Example:
  - A desktop has a hard disk, motherboard, and so on.
  - The computer can stop working through no fault of the hard drive.
  - The hard disk can still work on other desktops, even if the current desktop is thrown away.

Composition

- A specialized type of aggregation.
- Once the container object is destroyed, the contained objects will cease to exist as well.
- The container will be responsible for creating the object instances.
- You can think of composition in terms of “part of”.
- Example:
  - Human beings have Legs and Hands.
  - Legs and Hands are part of Human beings.
  - If a Human being dies, his Legs or Hands won’t work on other Human beings.

Association over Inheritance

- Inheritance gets so much focus.
- A new developer uses Inheritance everywhere.
- This can result in awkward and over-complicated class hierarchies.
- You should first consider composition when you are about to create a new class, and only if applicable should you make use of inheritance.

- Association has a great deal of flexibility.
- Since the instantiated objects are not accessible by the client of your class, you have the liberty of changing them, without impacting the client code at all.

Class Delegation

- It allows a type to forward one or more of its methods call to a different type.

- Therefore, you need two types to achieve this:
  1. Delegate
  2. Delegator

Class Delegation: Example

```
interface UIElement {
    fun getHeight(): Int
    fun getWidth(): Int
}
class Rectangle(val x1: Int, val x2: Int, val y1: Int, val y2: Int): UIElement {
    override fun getHeight() = y2 - y1
    override fun getWidth() = x2 - x1
}
class Panel(val rectangle: Rectangle) : UIElement by rectangle

val panel = Panel(Rectangle(10,100,30,100))
println("Panel height:” + panel.getHeight())
println("Panel witdh:" + panel.getWidth())
```

- by keyword forwards the calls for the methods exposed by the interface UIElement to the underlying Rectangle object.
- Through this pattern, you replace inheritance with composition.
- You should always favor composition over inheritance for the sake of simplicity and flexibility.

Questions?

Thank You!