Android Development with Kotlin

Classes in Kotlin

What is a class?

Classes can be thought of as the foundations of the Object-Oriented Programming Languages. They are the blueprints upon which the objects are built. Classes without objects are useless. It's the objects of a class that make it useful by mirroring the real-life entities in a program. A class consists of properties i.e. variables and values, and member functions which are synonymous to the characteristics and behaviours of the objects they are the blueprints of.
Classes in Kotlin can be defined using the "class" keyword, followed by the name of the class. A class in Kotlin can be defined as:

```
class className {    // class header
      // property
      // member function
}
```

Kotlin classes are default by public and can be accessed throughout the project, to modify the access of a Kotlin class, an access specifier can be added before the class. Kotlin offers four access specifiers, namely public, internal, protected and private. Unlike Java, a class in Kotlin cannot be protected and the keyword protected is reserved to specify the access of the contents of the class and not the class itself. Using a internal keyword means that the class and contents are accessible throughout the module, whereas the protected keyword ensures that the contents of the class are only available to its inherited child classed, or sub-classes. An internal class is declared as follows:

```
internal class internalClassName {    // class header
/* the contents of this class are only available
to the Module it is a part of. */
}
```

The constructors of a Kotlin class can be defined in two ways, explicitly by the use of the constructor keyword or within the class declaration itself.

```
class className {
      var name: String?
      //Explicit constructor definition
      constructor (name: String?) {…}
}
```

In Kotlin the implicit constructor is used with an init function, as demonstrated by the code below.

```
class className(name: String?) {    //Implicit constructor definition
      var name: String?
      init {…}
}
```

If the only purpose of the constructor is to initialize a property, the declaration of the property can be done within the constructor itself, which reduces the line of code significantly.

```
class className(var name: String?) {
        …
}
```

One of the best features of Kotlin is a portable code under which Kotlin allows easy definition of getters and setters for the properties (or variables) defined within a class.

```
class Company {
         var name: String = "defaultValue"
         get() = field                                   // getter
         set(value) { field = value }                    // setter
}
```

The getters and setters in Kotlin, use "field" instead of the property itself as doing so will reinvoke the getter/setter for the property creating an endless recursive loop. The keyword "field" is used to refer the backing field which temporarily stores the value of the property.

Inheritance and Polymorphism

Kotlin, much like Java, allows inheritance of class properties, wherein the derived class inherits the characteristics and behaviours of its parent class. Kotlin also allows a single class inheritance, just like Java but not every class is inheritable. To make a class inheritable in Kotlin, an "open" keyword is to be used before the class specifier. The following code demonstrates an example of class inheritance in Kotlin

```
open class classA {…}
class classB : classA {…}
```

The constructor of the parent class is called before the sub-class and is explicitly mentioned as follows:

```
open class classA(val name:String?) {…}
class classB(val name:String?) : classA(name) {
}
```

Polymorphism allows the code to be contextual, i.e. the same code segment can result in different outcomes based on the context it is being used in. Kotlin offers two types of Polymorphism; compile-time polymorphism and run-time polymorphism. Compile-time polymorphism can be achieved by overloading where more than one methods share the same name with different parameters or signature and different return type. An example of compile-time polymorphism in Kotlin is given as follows:

```
fun printNumber(n : Number){
      println("Using printNumber(n : Number)")
      println(n.toString() + "\n")
}
```

```
fun printNumber(n : Int){
      println("Using printNumber(n : Int)")
      println(n.toString() + "\n")
}
fun main{
      val a : Number = 99
      val b = 1
      printNumber(a) //First version of printNumber is getting used
      printNumber(b) //Second version of printNumber is getting used

}
```

Runtime-polymorphism allows us to use the same interface for different classes that inherit the same parent class. The essence of run-time polymorphism function overriding i.e. a function (or method) that is re-defined in the inherited classes, and essentially has the same name and signature as the parent class's function but a different functional body. Therefore, the same function when called, will give different results for different instances. The definition of overridden functions is initiated with an "open" keyword in parent class and the "override" keyword within the subclass respectively. An example of run-time polymorphism in Kotlin is given as follows:

```
open class Animal {
      open fun makeSound(){
            println("Animal Sound")
      }
}
class Dog: Animal {
      override fun makeSound(){
            println("Bark")
      }
}
class Cat: Animal{
      override fun makeSound(){
            println("Meow")
      }
}
fun main{
      val animal1: Animal = Dog()
      val animal2: Animal = Cat()
      animal1.makeSound()            //Prints "Bark"
      animal2.makeSound()            //Prints "Meow"
}
```

Abstract Classes v/s Interfaces

An abstract class is a class that cannot be instantiated i.e. an instance of this class cannot be created. The members of an abstract class are non-abstract by default and have to be specified as abstract using an "abstract" keyword. Abstract classes are used as basis for the subclasses to extend and improve upon, wherein the abstract members are defined in the subclass instead of the parent class. The function in the subclass does initiate with the "override" keyword but the function in the parent class uses the “abstract” keyword instead of the “open” keyword. An example of the abstract class is given below:

```
abstract class Animal {
      abstract fun makeSound()


}
class Dog: Animal {
      override fun makeSound(){
            println(“Bark”)
      }
}
```

Interfaces are another way to achieve abstraction in Kotlin, but unlike an abstract class the members of an interface are abstract by default. The key differences between an Abstract class and interface are that

- An Abstract class can have a state but an interface cannot i.e. there are no properties with backing fields in an interface and as a result, an interface cannot have a constructor.
- Classes are used to implement inheritance whereas interfaces are used to implement behaviours, therefore a class can extend or inherit multiple interfaces in Kotlin, but can have only one parent class (abstract or not).

Interfaces in Kotlin are declared as follows:

```
interface Flyable {
      fun fly()
}
class Bird: Flyable {
      override fun fly(){
            println(“Flying”)
      }
}
```

What are Data Classes?

A data class in a class is Kotlin that only has properties and usually does not perform functionalities. Data classes are used in Kotlin as Kotlin itself provides an immense amount of self-generated code such as, equals to functions, deconstructing options etc. A data class can have properties other than the properties in its constructor, but it is advised to only use the constructor bound properties as the equal to function overlooks any properties that aren’t mentioned in the constructor of a data class. Data classes in Kotlin are used as follows:

```
data class User(val name: String, val age: Int){
      val college: String
}
fun main(){
      val user1 = User(“John”, 20)
      user1.college = “IIT”
      val user2 = User(“John”, 20)
      user2.college = “NIT”
```

```
        val bool = user1.equals(user2)
        /* The value of bool is true, as the property "college"
        Isn't mentioned in the constructor of the class*/
    }
```

Enum v/s Sealed Classes

Enum classes are user-defined data types that can hold certain constant values. An enum in Kotlin has a constructor that defines a set of properties that each enum constant can hold, and the class itself contains a set of pre-defined values that an enum object can be associated to. A sealed class on the other hand is a class that limits its subclasses to the ones nested within itself. The key difference between an enum and a sealed class is that all enum objects have similar properties whereas, the objects that can be initialized for a sealed class may have different properties depending on its subclass. Multiple instances can be initialized for the objects of a sealed class but an enum object has to choose from a set of pre-defined instances. The following code segments depict the use of both an enum and a sealed class in Kotlin:

```
    enum class Color(val rgb: Int) {
        RED(0xFF0000),
        GREEN(0x00FF00),
        BLUE(0x0000FF)
        //The enum can have only 3 values, each with the same structure
    }
    sealed class Expr {
        data class Const(val number: Double) : Expr()
        data class Sum(val e1: Expr, val e2: Expr) : Expr()
        object NotANumber : Expr()
        /*The sealed class can have a number of instances and objects
        but the instances can only be limited to the ones mentioned in
        the class itself. No other class can extend Expr*/
    }
    fun eval(expr: Expr): Double = when(expr) {
        is Const -> expr.number
        is Sum -> eval(expr.e1) + eval(expr.e2)
        NotANumber -> Double.NaN
        /*the `else` clause is not required because we've covered
        all the cases*/
    }
```

Singletons and Companion Objects

A Singleton pattern is a software design pattern wherein only a single instance of a class exists throughout the project and the class itself provides a global access point to that particular instance.
Kotlin, unlike Java, does not contain a "static" keyword and the companion object provides Kotlin with static functionalities. A companion object is an object whose single instance is shared by the entire class, and its properties and functions can only be accessed through the class itself. A companion object is initialized with the initialization of the class and has access to all its members, including the private members of that class. A companion object cannot exist without a class.
Singleton patterns in Kotlin are implemented with help of companion objects, which consist of an instance of the class itself along with a private constructor which restricts construction of

objects outside the class and its companion object. The implementation of companion objects in Kotlin is defined as below:

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

- or –

```
class MyClass {
	companion object {			//the name can be omitted
			fun create(): MyClass = MyClass()
	}
}
```