Android Mobile Application Development
Week 3: More Kotlin

Arrays

In Kotlin arrays are represented by the Array class. Arrays are an ordered collection of a fixed size

- Elements must be of the same type
- Arrays can contain both primitive data types as well as objects of a class
- Items are not promised to be unique
- Must specify a size when creating an array

Creating a new empty array

Example: https://pl.kotl.in/LDuHzJhqx

```
var womens_sizes = emptyArray<Int>()
womens_sizes += 0
womens_sizes += 2
```

When adding an item to an array a new array is created, all the elements are copied into it, and the new items are added. Simple syntax but an expensive operation.

You can access array items using the index and [] notation.

```
println(womens_sizes[1])
```

Creating an array with items

```
var unisex_sizes = arrayOf("small", "medium", "large")
for (size in unisex_sizes){
    println(size)
}
```

Along with the for-in loop Kotlin has a forEach loop that accepts a lambda expression.

```
unisex_sizes.forEach{size -> println(size)}
```

Kotlin also has classes that represent arrays of primitive types. The size of the array is specified in the ().

```
var mens_sizes = IntArray(2)
println(mens_sizes[0])
```

Arrays have a size property that returns the size of the array.

```
println(mens_sizes.size)
```

If you try to access an array index that is larger than its size and doesn't exist you will get an array index out of bounds exception. `java.lang.ArrayIndexOutOfBoundsException`

```
println(mens_sizes[2]) //error
```

Collections

Kotlin offers three types of collections each with a mutable and immutable interface.

- Lists are an ordered collection

    - Elements must be of the same type.
    - Lists can contain both primitive data types as well as objects of a class
    - Items are not promised to be unique
    - Very similar to arrays but lists do not have a predefined size, the size is dynamic
- Sets are very similar to Lists with two exceptions:
    - Unordered collection
    - Items are unique
- Maps store key/value pairs
    - Keys are unique, values are not
    - Similar to associated arrays or dictionaries

Create an empty List

```
var size_list = mutableListOf<String>()
```

You can add as many items as you want to a List

```
size_list.add("small")
size_list.add("medium")
for (size in size_list){
    println(size)
}
```

You can remove items using their value or index.

```
size_list.remove("medium")
size_list.add("large")
for (size in size_list){
    println(size)
}
size_list.removeAt(0)
println(size_list.size)
println(size_list[0])
```

OOP

Kotlin is an object-oriented programming language.

Classes

- A class is a template/blueprint that describes the behavior/state that the object of its type supports.
    - Properties for data (variables and constants)
    - Methods for behavior (functions)
- You can control the visibility of a class as well as its variables and methods by specifying the access level. The visibility modifiers are
    - public: accessible outside the class
        - In Kotlin public is the default
    - private: only accessible inside the class
    - protected: accessible by the class and its subclasses
    - internal: visible within the module
- The class declaration consists of
    - class keyword
    - the class name which should start with an upper-case letter

- the class header in parenthesis containing
  - parameters
  - the primary constructor
- class body surrounded by curly braces.

```
class Animal(){}
```

Constructors

Constructor methods initialize objects of that class.
A class in Kotlin can have a primary constructor and one or more secondary constructors**.** The primary constructor is part of the class header.

```
class Animal constructor(animalName: String, animalWeight:Int){}
```

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

```
class Animal (animalName: String, animalWeight:Int){}
```

Initialization code can be placed in initializer blocks**,** which are prefixed with the *init* keyword. The initializer blocks are executed in the same order as they appear in the class body.

```
      class Animal(animalName: String, animalWeight:Int){
        var weight :Int
        init{
            weight = animalWeight
        }
        var name :String
        init{
            name = animalName
        }
    }
```

You can simplify this by combining the init block and the variable declaration into one statement.

```
class Animal(animalName: String, animalWeight:Int){
      var name = animalName;
      var weight = animalWeight;
}
```

Kotlin also lets you declare class properties and initialize them all in the primary constructor.
You can also include default values for class properties but they still require the type definition when in a constructor.

```
class Animal(var name: String, var weight:Int=0){}
```

The primary constructor cannot contain any code for logic for its parameters.

Secondary constructors

A Kotlin class can have at most one primary constructor which initializes the class. Kotlin can have multiple secondary constructors which allow instances of the class to be initiated with different value sets. Secondary constructors also allow initialization and often include some extra logic.

Secondary constructors are prefixed with “constructor”.

```
class Animal(var name: String, var weight: Int=0){
      var species: String = ""
      constructor(name: String, weight: Int, animalSpecies: String):
this(name, weight){
          species=animalSpecies
      }
}
```

Here the primary constructor handles the basic initialization and the secondary constructor also initializes the species variable as well.

Since the primary constructor does not initialize the species property it must be declared and initialized within the body of the class.
Although the name and weight properties are accepted as parameters to the secondary constructor, the variable declarations are still handled by the primary constructor and do not need to be declared. To associate the references to these properties in the secondary constructor with the primary constructor, we call the primary constructor using the *this* keyword.

Objects
An object is an instance, or occurrence, of a given class. An object of a given class has the structure and behavior defined by the class that is common to all objects of the same class.
Many different objects can be defined for a given class with each object made up of the data and methods defined by the class.
It’s only when an object is instantiated that memory is allocated.
In Kotlin, objects store the reference to the memory where its data is stored.
There is no “new” keyword in Kotlin as in other languages.

```
var animal1 = Animal("Hazel", 50)
println(animal1.name)
```

Methods
A method is where the logic is written to define a specific behavior. Data is manipulated and all the actions are executed.

- Method names should start with a lower-case letter

```
class Animal(var name: String, var weight:Int){
    fun speak(){
        println("$name says hi")
    }
}

animal1.speak()
```

Properties

All non-null properties must be initialized in a constructor. If that's not possible you can mark the property with the keyword lateinit.

```
class Animal(var name: String, var weight:Int){
  var guardian: String
    fun speak(){
        println("$name says hi")
    }
}
```

This results in an error: Property must be initialized or be abstract

You must specify that this property will be initialized later, after the constructor method is called, by using the keyword lateinit.

- Can be used on var class properties
  - not in the primary constructor
  - not when the property has a custom getter or setter
- type of the property or variable must be non-null
- type of the property must not be a primitive type

```
lateinit var guardian: String
```

If you access a property marked with lateinit before it has been initialized, you will get an error.

```
println(animal1.guardian)
```

This results in an error: Exception in thread "main" kotlin.UninitializedPropertyAccessException: lateinit property guardian has not been initialized

```
    fun setup(guardianName:String){
        guardian = guardianName
    }


animal1.setup("Gail")
println(animal1.guardian)
```

To check whether a `lateinit var` has already been initialized you can use `.isInitialized` on the reference to that property.

Inheritance

Inheritance allows you to define a hierarchy of classes with subclasses acquiring the properties of the superclass.

Kotlin access modifiers define the visibility of a class, object, properties, methods, constructors and interfaces.

- Final
  - A final class means you can't create a subclass
  - A final method means you can't override it
  - This is the default in Kotlin
- Open

    - Allows a class or method to be inherited
- Override
    - Overrides a method's implementation in a subclass (or a property)
- Abstract
    - An abstract class or method does not have an implementation in its class
        - An abstract class cannot be instantiated
        - It will be implemented in a subclass or a class that implements an interface
    - A class or method marked as abstract is automatically open so the open keyword is not needed

```
class Animal{}
class Dog : Animal(){}
```

This will cause an error: "This type is final, so it cannot be inherited from"
To make a class inheritable you must mark it with the keyword open.

```
open class Animal{}
class Dog : Animal(){}
```

If the subclass has a primary constructor the superclass must be initialized as well, using the parameters from the primary constructor.

```
open class Animal(var name: String, var weight:Int){}
class Dog(var breed: String, name: String, weight:Int) :
Animal(name, weight){}

var animal1 = Animal("Hazel", 50)
println(animal1.name)
var pet = Dog("lab", "Cole", 40)
println(pet.breed)
println(animal1.breed) //error, breed property in the Animal class
```

If the subclass has no primary constructor then each secondary constructor has to call the superclass constructor using the keyword super.

```
open class Animal(var name: String, var weight:Int){}
class Dog: Animal{
      var breed: String
      constructor(dogBreed: String, name: String, weight:Int)
     :super(name, weight){
            breed = dogBreed
       }
}
```

To override a method in a subclass you use keyword override before the method definition.

```
open class Animal(var name: String, var weight:Int){
      fun speak(){
            println("$name says hi")
       }
```

```
}

class Dog(var breed: String, name: String, weight:Int) : Animal(name,
weight){
      override fun speak(){
            println("$name says woof")
       }
}
```

This will cause an error: 'speak' in 'Animal' is final and cannot be overridden.
Methods in a class are also final by default so we need to use the keyword open so they can be overridden.

```
open class Animal(var name: String, var weight:Int){
      open fun speak(){
            println("$name says hi")
       }
}

var animal1 = Animal("Hazel", 50)
println(animal1.name)
animal1.speak()
var pet = Dog("lab", "Cole", 40)
println(pet.breed)
pet.speak()
```

All classes in Kotlin have a common superclass called Any. Any is the default superclass for a class with no superclass specified. Any has three methods: equals(), hashCode() and toString(). Thus, they are defined for all Kotlin classes.

Data classes

We often create classes whose main purpose is to hold data. Kotlin has the ability to declare any class as a data class by using the keyword data.

- The Kotlin compiler will automatically override the `toString()`, `equals()`, `hashCode()`, and `copy()` methods from the Any class and provide implementations for the data class
    - Derived from the properties declared in the primary constructor
- Eliminates the need for you to manually override these methods and implement them for your data class

Data classes must meet these requirements

- The primary constructor needs to have at least one parameter
- All primary constructor parameters need to be marked as val or var
- Data classes cannot be abstract, open, sealed or inner

```
class Person(val name:String){}
val person1 = Person("Diane")
val person2 = Person("Diane")
println(person1)
println(person2)
```

```
println(person1 == person2)
```

person1 == person2 evaluates to false because the `toString()` method and the `equals()` method have not been implemented. We would need to override and implement the `toString()` method and the `equals()` method if we wanted to test for name equality. Adding the keyword data to make this a data class does this for us.

```
data class Person(val name:String){}
```

Since both names are the same equals() now returns true.

Class properties not defined in the constructor are not used in the override method implementations. So let's add an age class property, assign the objects different ages, and then test for equality.

```
data class Person(val name:String){
    var age:Int = 0
}
val person1 = Person("Maggie")
val person2 = Person("Maggie")
println(person1)
println(person2)
println(person1 == person2)
person1.age = 21
person2.age = 20
println(person1 == person2)
```

It will still evaluate to true because age is not defined in the constructor so it is not used in the override implementation of equals().

Singleton objects
Singletons are classes that have only one instance declared. In Kotlin these are defined with the `object` keyword. Singletons are an easy way to access an object throughout a project. We've already used one in Compose when we accessed MaterialTheme, which is defined as a singleton object, making it easy to access all theme properties in your project.

Interfaces
In Kotlin an interface can be defined as a contract between objects on how to communicate with each other.
An interface defines the methods, a deriving class (subclass) should use. But the implementation of the methods is totally up to the subclass.
A subclass that implements an interface must implement the required methods and has the choice to implement any optional ones.

Extension Functions
Kotlin provides the ability to add more functionality to an existing class or interface, without inheriting them, through extension functions. So new functions can be written for a class that you can't modify and they can be used just as if they were methods of the original class.
We'll be using some extension functions in the SDK and we use them no differently than methods that are part of the class.

Packages

A package groups related classes together in one directory whose name is the same as the package name

- The android.app package contains high-level classes encapsulating the overall Android application model

In Java the default visibility is private within the package.
Kotlin uses packages differently. Packages are used to organize files but not as a way to manage visibility.
Kotlin has modules which is a collection of files compiled together.

- The keyword internal means it's visible in a module

Debugging

Debug messages

- To log messages to the console you can use **println("string");**
- Print messages to the LogCat
  - DEBUG: Log.d(tag, message);
  - ERROR: Log.e(tag, message);
  - INFO: Log.i(tag, message);
  - VERBOSE: Log.v(tag, message);
  - WARN: Log.w(tag, message);
- Log.i("TAG", "in start of count method");
- Filter to just see the logs for your tags

Debugging https://developer.android.com/studio/debug/index.html
More info on Logcat https://developer.android.com/studio/debug/am-logcat.html