UT1.- Introducción a Kotlin para Android

Listas

- Una lista es una colección redimensionable y ordenada que, por lo general, se implementa como un array que puede cambiar de tamaño

- List y MutableList
  - **List** es una interfaz que define las propiedades y los métodos relacionados con una colección ordenada de solo lectura de los elementos.
  - **MutableList** extiende la interfaz List con la definición de métodos para modificar una lista, como agregar o quitar elementos.
- Al igual que arrayOf(), la función listOf() toma los elementos como parámetros, pero devuelve un elemento List en lugar de un array.

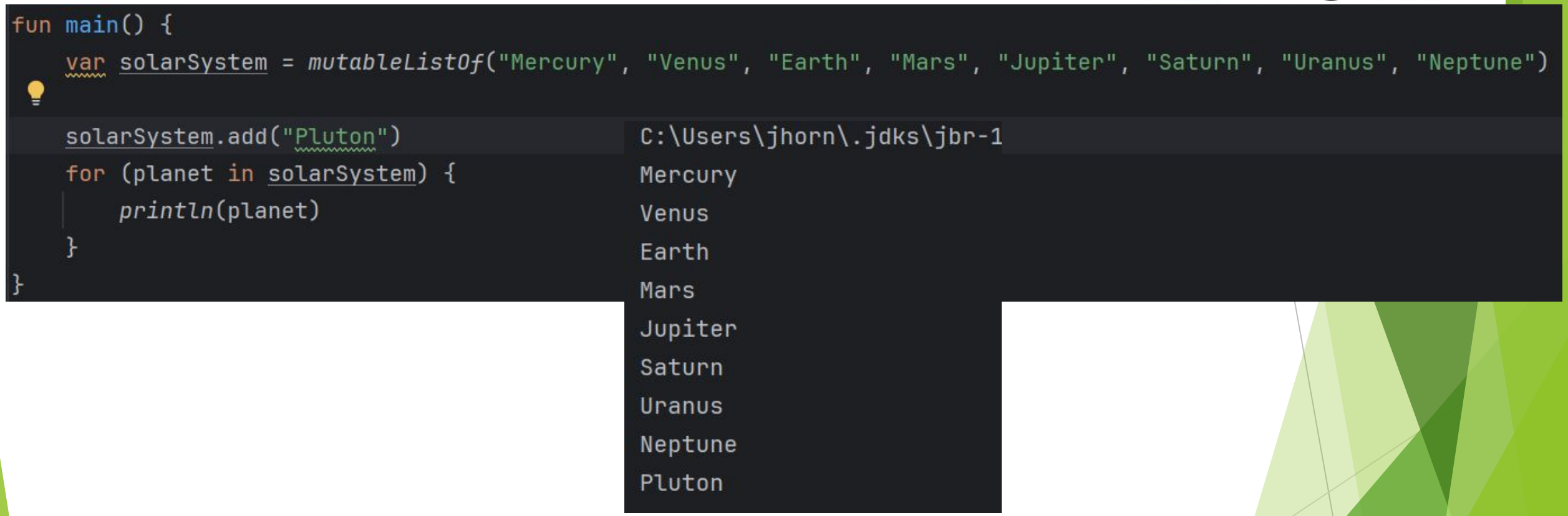
Listas

```
fun main() {
    var solarSystem = listOf("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune")

    for (planet in solarSystem) {
        println(planet)
    }
}
```

```
C:\Users\jhorn\.jdks\j
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
```

```
fun main() {
    var solarSystem = listOf("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune")

    for (planet in solarSystem) {
        println(planet)
    }
    solarSystem.add
}
```

Listas



```
fun main() {
    var solarSystem = mutableListOf("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune")

    solarSystem.add("Pluton")
    for (planet in solarSystem) {
        println(planet)
    }
}
```

```
C:\Users\jhorn\.jdks\jbr-1
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluton
```

Listas

- Para quitar un elemento, puedes pasarlo al método **remove**() o utilizar su índice mediante **removeAt**().

- List proporciona el método **contains**() que devuelve un Boolean si existe un elemento en una lista. Una sintaxis aún más concisa consiste en usar el operador **in**
- Si **en el momento de declarar la lista no tienes los elementos y no se puede inferir su tipo, hay que indicarlo (y luego ya se añadiran los elementos)**

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

- Se pueden hacer listas (y arrays) de objetos igual que en Java.
- https://developer.android.com/codelabs/basic-android-kotlin-training-lists?hl=es-419#1

Listas

Kotlin

```
fun main() {
    var solarSystem = mutableListOf("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune")

    solarSystem.add("Pluton")
    for (planet in solarSystem) {
        println(planet)
    }
    println("---------------------------------")
    println("Pluton" in solarSystem)
    solarSystem.remove( element: "Pluton")
    println("Pluton" in solarSystem)
    println("---------------------------------")
    solarSystem.removeAt( index: 0)
    for (planet in solarSystem) {
        println(planet)
    }
}
```

```
C:\Users\jhorn\.jdks\jbr-17.0.7\bin
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluton
---------------------------------
true
false
---------------------------------
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
```

Clases

- Declaración de clase en Kotlin mediante la palabra reservada ***class***.

- La declaración de clase consta del nombre de la clase, el encabezado de la clase (especificando sus parámetros de tipo, el constructor principal y algunas otras cosas) y el cuerpo de la clase rodeado por llaves. Tanto encabezado como cuerpo son opcionales; si la clase no tiene cuerpo, se pueden omitir las llaves.

```
class Persona { }
```

```
class Persona
```

- Una clase en Kotlin puede tener un **constructor principal** y uno o más **constructores secundarios**.

Constructor principal

- El **constructor principal** es parte del encabezado de la clase y va después del nombre de la clase:

```
class Persona constructor (nombre: String, apellidos: String, edad: Byte) { }
```

- Si el **constructor principal** no tiene ***annotations*** o modificadores de visibilidad, la palabra clave ***constructor*** se puede omitir:

```
class Persona (nombre: String, apellidos: String, edad: Byte) { }
```

- En los casos anteriores no se han creado las propiedades. Kotlin tiene una sintaxis concisa para declarar propiedades e inicializarlas desde el constructor principal:

```
class Persona (var nombre: String, var apellidos: String, var edad: Byte)
```

Constructor principal

Kotlin

```
.kt        Persona.kt ×
class Persona (var nombre:String, var apellido:String, var edad:Int){
```

```
fun main() {
    var miPersona = Persona( nombre: "Julio",  apellido: "Hornos",  edad: 51)
    println("Yo soy ${miPersona.nombre} y tengo ${miPersona.edad} años")
}
```

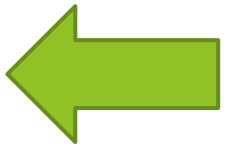
```
C:\Users\jhorn\.jdks\jbr-17.0.7
Yo soy Julio y tengo 51 años
```

Constructor principal

- Si decides hacer ll **constructor principal,** no se declara un método como en Java ni se ponen parámetros. Para llevar a cabo la inicialización se puede colocar código en bloques ***init*** que se ejecutan en el mismo orden en que aparecen en el cuerpo de la clase cuando esta se instancia.

- En ese caso, las propiedades son declaradas en el interior de la clase y los parámetros pasan a ser parámetros de constructor sin ***val*** ni ***var***.

Constructor primario



```kotlin
class Persona (nombre: String, apellidos: String, edad: Byte) {
    var nombre: String
    var apellidos: String
    var edad: Byte
    init {
        this.nombre = nombre
        this.apellidos = apellidos
        this.edad = edad
        println ("Constructor primario")
    }
}

fun main () {
    var persona1: Persona = Persona("Marta" , "Martínez", 22)
}
```

Constructores secundarios

- Una clase también puede declarar **constructores secundarios**, los cuales tienen el prefijo ***constructor***.

```
class Persona ( var nombre:String, var apellido:String,  var edad:Int){

    constructor( nombre: String, apellido: String): this(nombre, apellido, edad: 51) {
        kotlin.io.println("Estoy inicializando con el constructor secundario")
    }
}

fun main() {
    var miPersona = Persona( nombre: "Julio", apellido: "Hornos")
    println("Yo soy ${miPersona.nombre} y tengo ${miPersona.edad} años")
}
```

```
C:\Users\jhorn\.jdks\jdk-17.0.7\bin\java.exe "-javaa
Estoy inicializando con el constructor secundario
Yo soy Julio y tengo 51 años
```

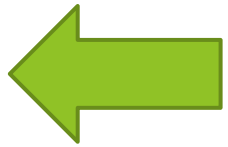
Constructores secundarios

- Si la clase tiene un constructor principal, cada constructor secundario delegará en él directa o indirectamente a través de otro(s) constructor(es) secundario(s).

- En este caso se invocará al constructor principal con la palabra ***this*** seguida de los dos puntos (:).

- Esta delegación es la primera declaración de un constructor secundario, por lo que el código de los bloques inicializadores se ejecutará antes que el cuerpo del constructor secundario.

```
Constructor primario para Marta
Constructor primario para David
Constructor secundario para David
```



```kotlin
class Persona (nombre: String, apellidos: String, edad: Byte) {
    var nombre: String
    var apellidos: String
    var edad: Byte
    init {
        this.nombre = nombre
        this.apellidos = apellidos
        this.edad = edad
        println ("Constructor primario para ${this.nombre}")
    }
    constructor (nombre: String, apellidos: String): this(nombre, apellidos, 0) {
        // otras posibles inicializaciones
        println ("Constructor secundario para ${this.nombre}")
    }
}

fun main () {
    var persona1: Persona = Persona ("Marta", "Martínez", 22)
    var persona2: Persona = Persona ("David", "Ortega")
}
```

Propiedades

```
class User {
    var firstName: String? = ...     // mutable (getter/setter)
    var lastName: String = ...       // mutable
    val age: Int = ...               // read-only (getter only)
}


// Use them as fields
fun test() {
    val user = User()
    user.firstName = "Grzegorz"                // setter is called
    print("firstName = ${user.firstName}")     // getter is called
}
```

En Kotlin no se usan getters ni setters. Se accede directamente a las propiedades con nombre de la clase.propiedad

Propiedades

```
class Persona ( var nombre:String, var apellido:String,  var edad:Int?=51){}
```

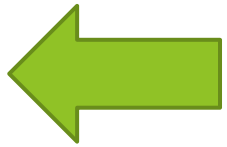
```
fun main() {
    var miPersona = Persona( nombre: "Julio", apellido: "Hornos")
    println("Yo soy ${miPersona.nombre} y tengo ${miPersona.edad} años")
}
```

```
C:\Users\jhorn\.jdks\jbr-17.0.7\bi
Yo soy Julio y tengo 51 años
```

Clases enumeración

- Se utilizan para modelar tipos que representan un conjunto finito de valores distintos (como direcciones, estados, modos, ...).
- Se declaran con palabra reservada ***enum*** .

```
Está casado
```



```kotlin
enum class EstadoCivil {
    SOLTERO, CASADO, VIUDO, DIVORCIADO
}

fun main() {
    val estado = EstadoCivil.CASADO
    val mensaje = when (estado) {
        EstadoCivil.SOLTERO -> "Está soltero"
        EstadoCivil.CASADO -> "Está casado"
        EstadoCivil.VIUDO -> "Está viudo"
        EstadoCivil.DIVORCIADO -> "Está divorciado"
    }
    println(mensaje)
}
```

Clases de datos

- Las clases de datos facilitan la creación de clases que se utilizan para almacenar valores.
- No realizan ninguna operación.
- Se proporcionan automáticamente con métodos para copiar, obtener una representación como String y usar instancias en colecciones.
- Métodos get/set construidos automáticamente.
- Se pueden sobrescribir estos métodos con implementaciones propias al declarar la clase.
- Se declaran con palabra reservada ***data*** .
- class vs data class

Clases de datos

n.kt PersonaData.kt

```kotlin
data class PersonaData(var nombre:String, var apellido:String, var edad:Int){
}

fun main() {
    var miPersona = PersonaData( nombre: "Julio", apellido: "Hornos", edad: 51)
    println("Yo soy ${miPersona.toString()}")
    println("O mejor mi nombre es ${miPersona.nombre}")
    miPersona.edad = 52
    println("Pronto mi edad será ${miPersona.edad} años")
}
```

```
C:\Users\jhorn\.jdks\jbr-17.0.7\bin\java.exe "-javaagent:C:\Pr
Yo soy PersonaData(nombre=Julio, apellido=Hornos, edad=51)
O mejor mi nombre es Julio
Pronto mi edad será 52 años
```

Clases de datos

- Es posible declarar varios ítems en el mismo fichero:

```kotlin
// Models.kt
data class User(val firstName: String, val lastName: String, val age: Int)
data class Address(val street: String, val zipCode: ZipCode)
data class ZipCode(val prefix: String, val postfix: String)
```

Métodos

- El método es una función asociada a un objeto. Por tanto se hace aplicable todo lo especificado anteriormente sobre funciones y argumentos.

```
Hola, soy Marta soy Martínez.
```

```kotlin
class Persona  {
    var nombre: String
    var apellidos: String
    var edad: Byte
    constructor (nombre: String, apellidos: String, edad: Byte) {
        this.nombre = nombre
        this.apellidos = apellidos
        this.edad = edad
    }
    fun saludar (){
        println ("Hola, soy ${this.nombre} soy ${this.apellidos}.")
    }
}

fun main () {
    var persona1: Persona = Persona ("Marta", "Martínez", 22)
    persona1.saludar()
}
```

Herencia

- Todas las clases en Kotlin tienen una superclase común, ***Any***, que es la superclase predeterminada para una clase sin supertipos declarados (como ***Object*** en Java).
- ***Any*** tiene los métodos *equals ()*, *hashCode ()* y *toString ()*. Por tanto, estos métodos se definen para todas las clases de Kotlin.
- La herencia entre clases se declara con dos puntos (*:*).
- Las clases son definitivas (***final***) por defecto. Para que una clase sea heredable se debe marcar con ***open***.
- Para sobrescribir un método o una propiedad, éste debe marcarse con ***open*** en la clase base. El método que lo sobrescriba en una clase derivada se marcará con ***override***.
- Si la clase derivada tiene un constructor primario, la clase base puede (y debe) inicializarse en ese constructor primario de acuerdo con sus parámetros.
- Si la clase derivada no tiene un constructor primario, entonces cada constructor secundario tiene que inicializar el tipo base usando la palabra clave ***super*** o tiene que delegar en otro constructor que lo tenga (diferentes constructores secundarios pueden llamar a diferentes constructores del tipo base).

Herencia

```kotlin
open class Forma(val nombre: String) {
    open fun area() = 0.0
}

class Circulo(nombre: String, val radio: Double): Forma(nombre) {
    override fun area() = Math.PI * Math.pow(radio, 2.0)
}

fun main(args: Array<String>) {
    val circulo = Circulo("Círculo", 2.0)
    println(circulo.nombre)
    println(circulo.radio)
    println(circulo.area())
}
```

```
Círculo
2.0
12.566370614359172
```

Interfaces

- El uso de interfaces en Kotlin es muy similar a Java.
- Al igual que la herencia se declara con dos puntos (:). Te obligará a implementar los métodos abstractos que haya en la interfaz.
- No es necesario que la interfaz sea “open” para poder implementarla

https://developer.android.com/courses/kotlin-android-fundamentals/overview?hl=es-419

https://kotlinlang.org/

https://play.kotlinlang.org/byExample/02_control_flow/02_Loops

https://kotlinlang.org/docs/basic-syntax.html#collections

https://developer.android.com/kotlin/getting-started-resources?hl=es

https://www.tutorialesprogramacionya.com/kotlinya/index.php?inicio=45