Universidad Nacional de La Matanza

Repaso Kotlin y Android

Repaso general

Agenda

- Funciones, variables y tipos de datos
- Estructuras de control y repetición
- Clases y objetos
- ¿Cómo seguir?

Variables

Variables

- Inferencia de tipo “poderosa”
  - Permite al compilador inferir el tipo
  - Se puede declarar el tipo explícitamente si es necesario
- Variables mutables e inmutables
  - La inmutabilidad no es obligada, pero sí recomendada

Kotlin es un lenguaje estáticamente tipado. El tipo es resuelto durante tiempo de compilación y nunca cambia.

Variables mutables e inmutables

- Mutables (Cambiables)

```
var score = 10
```

- Inmutables (No cambiables)

```
val name = "Jennifer"
```

Aunque no se pide estrictamente, usar variables inmutables es recomendado en la mayoría de los casos.

Strings

Strings

Strings son cualquier secuencia de caracteres encerrados por comillas dobles.

```
val s1 = "Hello world!"
```

String literals pueden tener ‘escape characters’

```
val s2 = "Hello world!\n"
```

O cualquier texto arbitrario delimitado por triple comillas dobles (""")

```
val text = """
    Hello world! I am very happy to be here
"""
```

Concatenación de strings

¿Como concatenamos en Java?

```
int numberOfDogs = 3
int numberOfCats = 2

"I have ” + numberOfDogs + ”dogs and” + numberOfCats + “cats"


=> I have 3 dogs and 2 cats
```

Concatenación de strings

¿Como lo hacemos en Kotlin?

```
val numberOfDogs = 3
val numberOfCats = 2

"I have $numberOfDogs dogs and $numberOfCats cats"

=> I have 3 dogs and 2 cats
```

Condicionales

Control de flujo

Kotlin posee varias formas de implementar la lógica condicional:

- If/Else statements
- When statements

Sentencias if/else

```
val numberOfCups = 30
val numberOfPlates = 50

if (numberOfCups > numberOfPlates) {
    println("Demasiadas copas!")
} else {
    println("Insuficientes copas!")
}
```

```
=> Insuficientes copas!
```

Sentencia when

```
val results = 100

when (results) {
    0  -> println("No results")
    in 1..39 -> println("Got results!")
    else -> println("That's a lot of results!")
}

⇒ That's a lot of results!
```

Estructuras de repetición

Estructuras de repetición

- for… each
- for… con índices
- for… con rangos
- while

for loops

```
val pets = arrayOf("dog", "cat", "canary")
for (element in pets) {
    print(element + " ")
}
⇒ dog cat canary
```

No tenés que definir una variable de iteración e incrementarla en cada vuelta

for loops: elementos e índices

```
for ((index, element) in pets.withIndex()) {
    println("Item at $index is $element\n")
}
⇒ Item at 0 is dog
Item at 1 is cat
Item at 2 is canary
```

for loops: step sizes and ranges

```
for (i in 1..5) print(i)
⇒ 12345

for (i in 5 downTo 1) print(i)
⇒ 54321

for (i in 3..6 step 2) print(i)
⇒ 35

for (i in 'd'..'g') print (i)
⇒ defg
```

while loops

```
var bicycles = 0
while (bicycles < 50) {
    bicycles++
}
println("$bicycles bicycles in the bicycle rack\n")
⇒ 50 bicycles in the bicycle rack

do {
    bicycles--
} while (bicycles > 50)
println("$bicycles bicycles in the bicycle rack\n")
⇒ 49 bicycles in the bicycle rack
```

repeat loops

```
repeat(2) {
    print("Hello!")
}
```

⇒ Hello!Hello!

Funciones

Sobre funciones

- Son un bloque de código que ejecuta una tarea específica
- Rompe el programa en pedazos modulares más pequeños
- Se declaran usando la palabra clave `fun`
- Pueden recibir argumentos que sean nombrados o con valores por defecto

Partes de una función

¿Cómo escribimos en Java una función que imprima un “Hello World”?

```
public void printHello() {
    System.out.println("Hello World");
}
```

```
fun printHello() {
    println("Hello World")
}
```

Partes de una función

¿Cómo escribimos en Java una función que retorne un “Hello World”?

```
public String printHello() {
    return "Hello World“;
}
```

```
fun printHello() : String {
    return "Hello World"
}
```

Partes de una función

¿Cómo escribimos en Java una función que recibe un numero y retorna su doble?

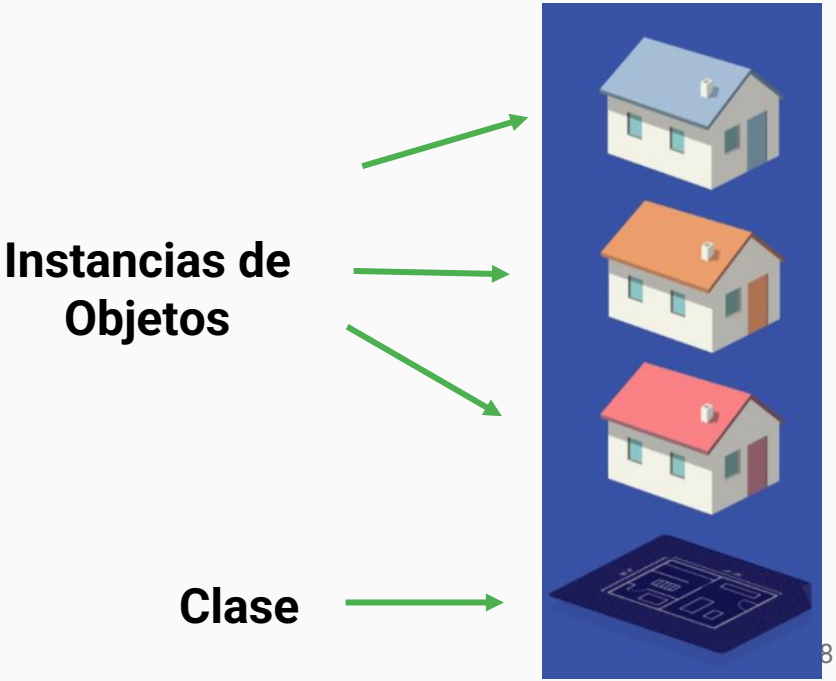
```java
public int multiplicar(int numero) {
    return numero * 2;
}
```

```kotlin
fun multiplicar(numero: Int) : Int {
    return numero.times(2)
}
```

Clases y objetos

Clase

- Las clases son planos para los objetos
- Las clases definen métodos que pueden operar en sus objetos



Clase

Clase: `Casa`

Atributos (Datos)

- `Color (String)`
- `Cantidad de ventanas (Int)`
- `En venta (Boolean)`

Comportamiento

- `actualizarColor()`
- `ponerEnVenta()`

Clases data

- Clase especial que existe solo para almacenar datos
- Se marcan con la palabra clave `data`
- Genera los getters para cada propiedad (y setters para `var` también)
- Genera los métodos `toString()`, `equals()`, `hashCode()`, `copy()`, y operadores de desestructuración

  Formato: `data class <NameOfClass>( parameterList )`

Ejemplo de clase data

Definimos nuestra clase data:

```
data class Player(val name: String, val score: Int)
```

Usamos nuestra clase data:

```
val firstPlayer = Player("Lauren", 10)
println(firstPlayer)
=> Player(name=Lauren, score=10)
```

La clase data hace nuestro código más conciso!

Companion objects

- En Kotlin todo es un objeto: no existe el concepto de funciones/métodos estáticos, o de constantes estáticas (que pertenezcan a clases)
- Si se quiere lograr lo mismo, lo que se puede hacer es definir un *companion object* para cada clase

Companion object example

```
class PhysicsSystem {
    companion object WorldConstants {
        val gravity = 9.8
        val unit = "metric"
        fun computeForce(mass: Double, accel: Double): Double {
            return mass * accel
        }
    }
}
println(PhysicsSystem.WorldConstants.gravity)
println(PhysicsSystem.WorldConstants.computeForce(10.0, 10.0))
```

Herencia

- Herencia implícita: Todos los objetos heredan de *Any* (en Java esto es así con *Object*)
  - equals(), hashCode() y toString()

- Para que se pueda heredar de una clase, esta debe ser marcada con la palabra reservada *open* (esto es exactamente al revés que en Java, donde hay que marcar las clases para que **no** se pueda heredar de ellas con *final*)

Interfaces

- Como en Java, Kotlin tiene herencia simple de clases, pero se pueden implementar tantas interfaces como se desee.
- En Kotlin las interfaces pueden tener propiedades.
- Las interfaces de Kotlin pueden tener implementaciones.
- Útiles para abstraer y agrupar ciertos comportamientos.

¿Preguntas?

Fin