

Introducción a Kotlin (Parte II)

Clases en Kotlin

Competencias:

Clases en Kotlin
Data class en Kotlin

Contexto

Uno de los pilares básicos de la programación orientada a objetos es el uso de clases y objetos. En este capítulo veremos cómo funcionan las clases en Kotlin y lo fácil que es crear una clase POJO en Kotlin con las “Data Class”.

Clases en Kotlin

Tarde o temprano necesitaremos utilizar algo más que las clases básicas de Kotlin (Int, String, Boolean, etc.), ahí es donde entran nuestras clases. Las clases son plantillas que nos permiten crear nuestros propios objetos, definiendo nosotros las propiedades y funciones.

En Kotlin y Java existen las mismas clases por lo que no profundizaremos sobre este contenido en este capítulo.

Nuestra primera clase en Kotlin :

```
class Persona{ }
```

Lo primero que extrañamos en nuestra clase, es el uso de la palabra reservada “public”, en Kotlin todas las clases son “public” por defecto, por lo tanto declarar una clase como public es redundante en Kotlin.



Imagen 1. Palabra reservada Public.

Data Class en Kotlin

Kotlin también puede trabajar con clases tipo P.O.J.O. (Plain Old Java Object) de forma mucho más eficiente que en otros lenguajes, cuando necesites crear una clase con atributos, get y set. Data class es la solución a este problema.

P.O.J.O en Java

```
public class Persona {  
    private String nombre;  
    private String rut;  
    private int edad;  
  
    public Persona(String nombre, String rut, int edad) {  
        this.nombre = nombre;  
        this.rut = rut;  
        this.edad = edad;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public String getRut() {  
        return rut;  
    }  
  
    public void setRut(String rut) {  
        this.rut = rut;  
    }  
  
    public int getEdad() {  
        return edad;  
    }  
  
    public void setEdad(int edad) {  
        this.edad = edad;  
    }  
}
```

Esta misma clase en Kotlin se puede desarrollar con un Data Class

```
data class Persona(val nombre: String, val rut: String, val edad: Int)
```

Main.kt

```
fun main(){
    val persona = Persona("Juan", "1-9", 39)
    println("Nombre Persona = ${persona.nombre}")
    println("Rut Persona = ${persona.rut}")
    println("Edad Persona = ${persona.edad}")
}
```

Solución



Imagen 2. Solución Data Class

Open Class en Kotlin

En Kotlin por defecto las clases son finales, es decir, no se puede heredar de dicha clase. Para poder utilizar la Herencia en Kotlin lo hacemos utilizando los ":" seguido del nombre de la clase padre y la llamada al constructor de esta clase.

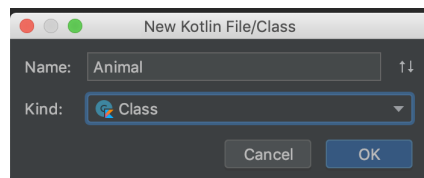


Imagen 3. Crear clase animal.

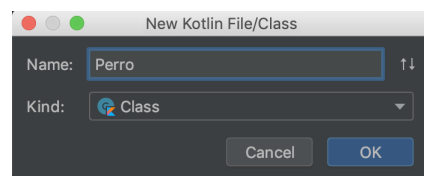


Imagen 4. Crear clase perro.

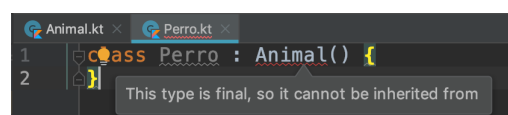


Imagen 5. Clase perro.

Acá vemos como nuestra clase Animal por defecto es una clase “final”

A screenshot of an IDE with two tabs: 'Animal.kt' and 'Perro.kt'. The 'Animal.kt' tab is active, showing the following code:

```
1 open class Animal {  
2 //Acá el cuerpo de la clase  
3 }
```

Imagen 6. Clase final.

Al declarar nuestra clase como open, esta se puede heredar en otra clase:

A screenshot of an IDE with two tabs: 'Animal.kt' and 'Perro.kt'. The 'Perro.kt' tab is active, showing the following code:

```
1 class Perro : Animal() {  
2 //Acá el cuerpo de la clase  
3 }
```

Imagen 7. Heredar desde otra clase.

Ya no tenemos el error de que nuestra clase es “final”.

Sealed Class en Kotlin

Las “Sealed Class” o clases selladas, se utilizan para definir una jerarquía de clases restringidas, esto con el fin de utilizar valores predeterminados. Funciona de forma similar a las enumeraciones, con la ventaja de que las clases selladas son mucho más flexibles que las enumeraciones y lo veremos en el siguiente ejemplo:

Creamos una enumeración llamada TipoDeMensaje.kt

```
enum class TipoDeMensaje(val tipo : String) {  
    ALERTA("Esto es una alerta"),  
    TOAST("Esto es una tostada"),  
    DIALOGO("Esto es un cuadro de diálogo"),  
    VENTANA("Esto es una ventana")  
}  
  
fun main(){  
    val tipoDeMensaje = TipoDeMensaje.ALERTA  
    println(tipoDeMensaje.tipo)  
}
```

El mismo ejemplo, lo podemos hacer con clases selladas Creamos una clase sealed llamada TipoDeMensaje.kt

```
sealed class TipoDeMensaje
class MensajeAlerta(val mensaje: String) : TipoDeMensaje()
class MensajeToast(val mensaje: String) : TipoDeMensaje()
class MensajeDialog(val mensaje: String) : TipoDeMensaje()
class MensajeVentana(val mensaje: String) : TipoDeMensaje()

fun main() {
    var alerta = MensajeAlerta("Esto es una alerta")
    val toast = MensajeToast("Esto es una tostada")
    val dialog = MensajeDialog("Esto es un cuadro de diálogo")
    val ventana = MensajeVentana("Esto es una ventana")
    println(alerta.mensaje)
    println(toast.mensaje)
    println(dialog.mensaje)
    println(ventana.mensaje)
}
```

Acá queda en evidencia la ventaja de las clases selladas ya que si quisiéramos agregar un nuevo tipo de mensaje, no necesitamos modificar la clase principal, lo único que debemos hacer es crear una nueva clase que herede de nuestra clase sellada.

Ejercicio: “Probando las clases en Kotlin”

1. Crear una clase POJO en Kotlin llamada JuegoDeArcade
2. Definir las siguientes propiedades: nombre, cantidad de jugadores, tipo de juego.
3. Crear funcion main
4. Declarar una variable de tipo “JuegoDeArcade” y asignarle valor
5. Imprime el valor por pantalla de la variable creada.

JuegoDeArcade.kt

```
data class JuegoDeArcade(val nombre : String, val cantidadJugadores : Int, val tipoJuego: String)
```

Método main

```
fun main() {  
    val juegoDeArcade = JuegoDeArcade("Kof98", 2, "Peleeas")  
    println("Nombre Juego: ${juegoDeArcade.nombre}")  
    println("Nro de Jugadores: ${juegoDeArcade.cantidadJugadores}")  
    println("Tipo de Juego: ${juegoDeArcade.tipoJuego}")  
}
```

```
Nombre Juego: Kof98  
Cantidad de Jugadores: 2  
Tipo de Juego: Peleeas  
Process finished with exit code 0
```

Imagen 8. Resultado de la ejecución del método main.

Constructores en Kotlin

Competencias:

- Constructores en Kotlin
- Constructor sin parámetros
- Constructor con parámetros
- Sobrecarga de constructores

Contexto

El uso de los constructores es esencial para la programación orientada a objetos, sin constructores nuestras clases no pueden ser instanciadas. En este capítulo veremos qué es un constructor, como se declaran un constructor con y sin parámetros y aprenderemos cómo trabajar con sobrecarga de constructores en una clase Kotlin.

Constructores en Kotlin

Kotlin, al igual que Java necesita crear constructores en sus clases para que estas puedan crear objetos, es tan importante el constructor en una clase, que el mismo lenguaje se encarga de crear un constructor sin parámetros en caso de que nosotros no lo creamos en nuestra clase.

Constructor sin parámetros

El constructor sin parámetros, es el constructor por defecto que tiene una clase, como ya mencionamos, si nosotros creamos una clase Kotlin, podremos llamar al constructor de esa clase por medio de la siguiente instrucción:

```
class Perro {  
}  
  
fun main(){  
    val perroSalchicha = Perro()  
}
```

A diferencia de Java, en Kotlin no utilizamos la palabra reservada “new” para llamar al constructor. En Kotlin para llamar al constructor se hace por medio del signo de igualdad “=” seguido del nombre del constructor que nosotros queramos llamar. En este caso se llama al constructor sin parámetros que crea por defecto Kotlin “= Perro()”.

Constructor con parámetros

Cuando nosotros declaramos un constructor con parámetros, esto se debe hacer de la siguiente manera:

```
class Perro(val nombreParam: String, val edadParam: Int) {  
    val nombre = nombreParam  
    val edad = edadParam  
}  
  
fun main(){  
    val perroSalchicha = Perro("Toky", 2)  
    print("El nombre del perro es : ${perroSalchicha.nombre} y la edad es: ${perroSalchicha.edad}.")  
}
```

En este caso para llamar al constructor con parámetros, llamamos al nombre de la clase, seguido de los parámetros de su constructor. “= Perro(“Toky”, 2)”.

Sobrecarga de Constructores

Cuando necesitemos una clase con más de un constructor a esto se le llama sobrecarga de constructores y esto se hace de la siguiente manera:

```
class Perro {  
    val nombre: String  
    val edad: Int  
    constructor() {  
        nombre = "Sin Nombre"  
        edad = 0  
    }  
    constructor(nombreParam: String, edadParam: Int){  
        nombre = nombreParam  
        edad = edadParam  
    }  
}  
  
fun main() {  
    val perroSalchicha = Perro()  
    val perroBulldog = Perro("Firulais", 15)  
    println("El nombre del perro es : ${perroSalchicha.nombre} y la edad es: ${perroSalchicha.edad}.")  
    println("El nombre del perro es : ${perroBulldog.nombre} y la edad es: ${perroBulldog.edad}.")  
}
```

```
El nombre del perro es : Sin Nombre y la edad es: 0.  
El nombre del perro es : Firulais y la edad es: 15.  
  
Process finished with exit code 0
```

Imagen 9. Salida del programa.

Ejercicio: “Probando los constructores en Kotlin”

1. Crear una clase llamada EquipoDeportivo
2. Definir las siguientes propiedades: nombre y deporte.
3. Crear constructor con parámetros con las dos propiedades
4. Crear constructor sin parámetros y asigna por defecto los valores “Sin nombre” para el nombre del equipo y “Fútbol” para el deporte.
5. Crear la función main.
6. Declarar dos variables de tipo “EquipoDeportivo”
7. Asignar valor a la primera variable llamando al constructor sin parámetros.
8. Asignar valor a la segunda variable llamando al constructor con parámetros.
9. Imprime el valor por pantalla de las 2 variables creadas.

EquipoDeportivo.kt

```
class EquipoDeportivo {  
    val nombre: String  
    val deporte: String  
    constructor() {  
        nombre = "Sin Nombre"  
        deporte = "Fútbol"  
    }  
    constructor(nombreParam: String, deporteParam: String){  
        nombre = nombreParam  
        deporte = deporteParam  
    }  
}
```

Método main

```
fun main() {  
    val equipo = EquipoDeportivo()  
    val equipo2 = EquipoDeportivo("Los Ferros", "Fútbol Americano")  
    println("Nombre Equipo: ${equipo.nombre}")  
    println("Deporte: ${equipo.deporte}")  
    println("Nombre Equipo: ${equipo2.nombre}")  
    println("Deporte: ${equipo2.deporte}")  
}
```

```
Nombre Equipo: Sin Nombre  
Deporte: Fútbol  
Nombre Equipo: Los Ferros  
Deporte: Fútbol Americano  
  
Process finished with exit code 0
```

Imagen 10. Resultado de la ejecución del método main.

Funciones del lenguaje

Competencias:

- Funciones en archivos Kotlin.
- Funciones de extensión en Kotlin

Introducción

En Kotlin además de crear funciones dentro de una clase, podemos crear funciones fuera de una clase e incluso podemos crear funciones de extensión para clases donde no tengamos acceso al código fuente.

Funciones en archivos Kotlin

En Kotlin al momento de crear una clase también podemos crear archivos, estos archivos Kotlin tienen la misma extensión que una clase “.kt”. En estos archivos podemos crear nuestras funciones kotlin sin la necesidad de crear una clase, hacer esto en Java es imposible, ya que todas las funciones deben estar dentro de una clase.

Botón derecho en carpeta “src” -> “New” -> “Kotlin File/Class”.



Imagen 11. Crear nueva clase/archivo.

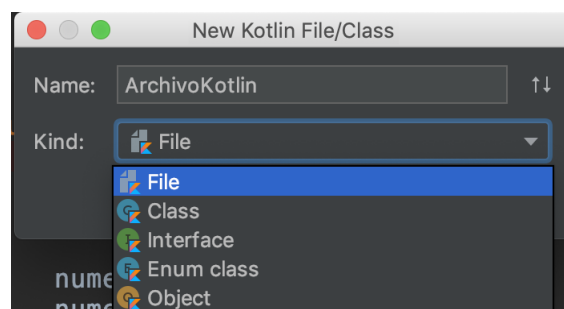


Imagen 12. Crear archivo Kotlin.

ArchivoKotlin.kt

```
fun saludo(){  
    print("Hola")  
}  
fun despedida(){  
    print("Chao")  
}
```

En este ejemplo podemos ver un archivo Kotlin que contiene 2 funciones, estas funciones pueden ser llamadas desde cualquier parte de nuestro proyecto

Main.kt

```
fun main(){  
    saludo() //  
    despedida()  
}
```

En este ejemplo podemos ver cómo se utilizan 2 funciones sin la necesidad de crear ninguna clase, estos métodos son propios del lenguaje Kotlin y podemos ver una representación de cómo funcionan por medio del IDE IntelliJ.

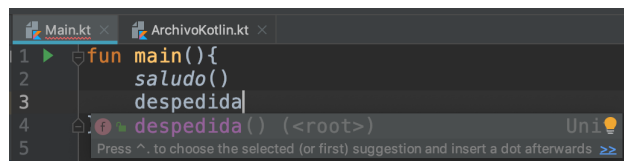


Imagen 13. Utilizar dos funciones sin una clase.

Los métodos declarados en archivos de tipo Kotlin se consideran como métodos () y gracias a esto podemos utilizarlos en cualquier clase.

Funciones de extensión Kotlin

Las funciones de extensión permiten al desarrollador agregar una función a una clase específica sin la necesidad de modificar el código fuente de la clase, estas funciones se crean de la misma manera que las funciones de los archivos Kotlin, pero se le debe indicar la clase donde queramos extender de estas funciones.

Ejemplo: Botón derecho en carpeta “src” -> “New” -> “Kotlin File/Class”.



Imagen 14. Crear clase/archivo.

Acá creamos un archivo Kotlin:

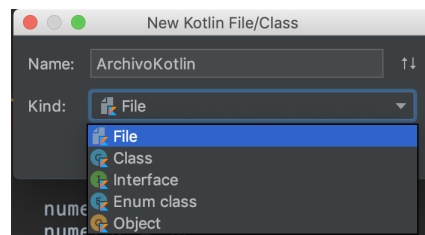


Imagen 15. Crear archivo Kotlin.

ArchivoKotlin.kt

```
fun String.saludo(){
    print("Hola")
}
fun String.despedida(){
    print("Chao")
}
```

En este ejemplo podemos ver cómo declaramos una función llamada “String.saludo” cuando al nombre de nuestra función le antepone el nombre de una clase (en este caso la clase String) significa que cualquier objeto de la clase String puede utilizar nuestra función “saludo” y “despedida”

Main.kt

```
fun main(){
    val variable = "Hola" //declaramos una variable String
    variable.saludo() //Utilizamos nuestra función de extensión "saludo"
    variable.despedida() //Utilizamos nuestra función de extensión "despedida"
}
```

Imagen 16. main.kt

En este ejemplo podemos ver cómo se utilizan 2 funciones en una variable de tipo String sin la necesidad de modificar la clase String, este tipo de función es propios del lenguaje Kotlin y podemos ver una representación de cómo funcionan por medio del IDE IntelliJ.

```
fun main(){  
    val variable = "Hola"  
    variable.saludo()  
    variable.despedida()  
}
```

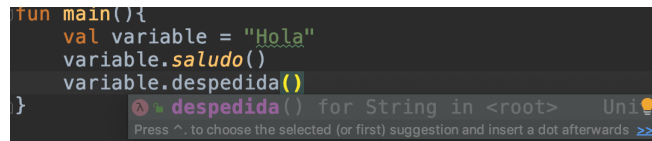


Imagen 17. Funciones en una variable string.

Las funciones de extensión declarados en Archivos Kotlin se consideran como métodos () para variables String y gracias a esto podemos utilizarlos en cualquier objeto que sea de este tipo.

Ejercicio: “Probando las funciones del lenguaje”

1. Crear un archivo Kotlin y agregar una función de extensión de la clase String llamada “encriptarValor”
2. Agregar la lógica que permita a la función “encriptarValor” cambiar los siguientes valores:

a	4
e	3
i	1
o	0
u	T_T

3. Crear la función “main” y agregar una variable de tipo String
4. Agregar un valor a la variable
5. Ejecutar la función “encriptarValor” de nuestra variable y agregar esta llamada en un método print.

Solución:

ArchivoKotlin.kt

```
fun String.encriptarValor(): String {  
    return this.replace("a","4")  
        .replace("e","3")  
        .replace("i","1")  
        .replace("o","0")  
        .replace("u","T_T")  
}
```

Main.kt

```
fun main(){  
    val variable = "Paralelepipedou"  
    println("Valor normal: $variable")  
    println("Valor encriptado: ${variable.encriptarValor()}")  
}
```

```
Valor normal: Paralelepipedou  
Valor encriptado: P4r4l3l3p1p3d0T_T  
  
Process finished with exit code 0
```

Imagen 18. Solución Ejercicio.