Clases y objetos en Kotlin: Modela tus datos con elegancia

1. Introducción

Si has llegado hasta aquí, ya sabes cómo manejar variables, colecciones, controlar el flujo de tu programa y crear funciones o lambdas para organizar tus acciones. ¡Enhorabuena! Tienes unas bases de programación fabulosas. Pero ahora toca dar el siguiente paso lógico y entrar de lleno en la **Programación Orientada a Objetos (POO)**.

Imagina que en tu aplicación tienes que gestionar la información de decenas de usuarios: su nombre, su identificador, su correo y su edad. ¿Vas a usar variables sueltas por todo el código para cada cosita de cada usuario? Sería un absoluto caos y muy propenso a errores. Para eso están las **clases** y los **objetos**. En esta unidad, vamos a ver cómo Kotlin hace que crear y agrupar estructuras de datos sea un proceso muy limpio, y descubriremos la magia de las **data classes** (clases de datos), una de las características estrella del lenguaje que te ahorrará horas de escribir código.

2. ¿Qué son las clases y cómo se declaran?

En Kotlin, la programación orientada a objetos es muy directa. Una clase te permite declarar un conjunto de características compartidas para un concepto. Piensa en la clase como si fuera el «molde» (por ejemplo, el concepto de `Cliente`) y en los objetos como las «galletas» reales que salen de ese molde (el cliente Juan, la clienta María).

Para declarar una clase, simplemente utilizamos la palabra reservada `class`:

```
1.  class Cliente
```

3. Propiedades de una clase

Las características concretas de ese molde se llaman **propiedades**. Puedes declarar las propiedades de una clase directamente entre paréntesis `()` justo después del nombre de la misma. A esta parte se le conoce como el **encabezado de la clase** (*class header*).

```
1.  class Contacto(val id: Int, var email: String)
```

También puedes definir propiedades adicionales dentro del **cuerpo** de la clase, abriendo unas llaves `{}`:

```
1.  class Contacto(val id: Int, var email: String) {
2.      val categoria: String = "trabajo"
3.  }
```

-

> *La regla de oro: Te recomendamos encarecidamente que declares tus propiedades como de solo lectura (`val`) siempre que sea posible. Usa variables mutables (`var`) únicamente si tienes la absoluta certeza de que ese dato necesitará cambiar después de haber creado el objeto.*

Al igual que ocurría con las funciones, las propiedades de una clase pueden tener **valores por defecto**:

```
1.  class Contacto(val id: Int, var email: String = "ejemplo@gmail.com")
    {
2.      val categoria: String = "trabajo"
3.  }
```

> *El dato: Si pones parámetros en el encabezado sin escribir `val` o `var` delante, Kotlin los tratará como simples parámetros de configuración y no como propiedades reales de la clase; por lo tanto, no podrás acceder a ellos después. ¡Acuérdate de poner siempre `val` o `var`!*
>
> Nota extra: *Kotlin te permite poner una coma al final de la última propiedad (conocido como* trailing comma*) para que te sea más fácil reordenar el código copiando y pegando líneas.*

4. Crear instancias (objetos)

Para poder usar esa clase en la vida real, necesitas instanciarla (crear un objeto) a través de un **constructor**. En lenguajes más antiguos como Java tendrías que usar la palabra `new` a la fuerza, pero a Kotlin le gusta el código conciso, así que no hace falta.

Por defecto, el lenguaje te crea un constructor automáticamente exigiendo los parámetros que hayas puesto en el encabezado.

```
1.  class Contacto(val id: Int, var email: String)
2.
3.  fun main() {
4.      // Creamos la instancia y la guardamos en una variable
```

```
    val contacto1 = Contacto(1, "maria@gmail.com")
}
```

En este ejemplo:

- `Contacto` es la **clase** (el molde).
- `contacto1` es la **instancia** (el objeto real en la memoria).

5. Acceder a las propiedades

Para leer o modificar una propiedad de un objeto que ya has creado, la sintaxis es facilísima: solo tienes que escribir el nombre de la instancia, seguido de un punto `.`, y el nombre de la propiedad.

```
fun main() {
    val contacto = Contacto(1, "maria@gmail.com")

    // Imprimimos el valor de la propiedad
    println(contacto.email)
    // Salida: maria@gmail.com

    // Actualizamos el valor (es posible porque lo definimos como 'var')
    contacto.email = "juana@gmail.com"

    // Imprimimos el nuevo valor
    println(contacto.email)
    // Salida: juana@gmail.com
}
```

Si necesitas concatenar el valor de esa propiedad dentro de un texto, recuerda usar las **plantillas de cadenas** con el símbolo del dólar. Como estás accediendo a una propiedad y no a una variable simple, es **obligatorio** envolverlo todo entre llaves `${}`:

```
println("Su correo electrónico de empresa es: ${contacto.email}")
```

6. Funciones miembro

Las clases no son simples «bolsas» para guardar datos estáticos. También pueden tener comportamiento propio. A las funciones que declaramos dentro del cuerpo de una clase se las denomina **funciones miembro** (o métodos).

Para llamar a estas funciones, volvemos a usar la sintaxis del punto `.`:

```
class Contacto(val id: Int, var email: String) {
    // Esta es una función miembro
    fun imprimirId() {
        println("El identificador secreto es: $id")
    }
}

fun main() {
    val contacto = Contacto(1, "maria@gmail.com")
    // Llamamos a la función
    contacto.imprimirId()
    // Salida: El identificador secreto es: 1
}
```

7. El superpoder de Kotlin: data classes

Aquí es donde Kotlin brilla y demuestra por qué es un lenguaje tan popular. En el día a día, muchas veces vas a crear clases cuyo *único* propósito sea almacenar información de forma pasiva. Para estos casos concretos, Kotlin inventó las **data classes** (clases de datos).

Tienen exactamente la misma funcionalidad que las clases normales que acabamos de ver, pero el compilador de Kotlin les inyecta automáticamente por detrás un montón de funciones extra para que no tengas que escribir código repetitivo o *boilerplate*.

Para declararlas, solo tienes que poner la palabra `data` antes del `class`:

```
data class Usuario(val nombre: String, val id: Int)
```

Al hacer esto, ganas acceso automático a tres funciones vitales que analizamos en los siguientes puntos.

7.1. Imprimir como texto (toString)

Si imprimes una clase normal de Kotlin por la consola, te escupirá un nombre incomprensible con su referencia en memoria (algo tipo `Usuario@2f92e0f4`). Pero las data classes sobrescriben la función `toString()` de forma nativa para que la salida sea humana y legible.

```
fun main() {
    val usuario = Usuario("Alex", 1)

    println(usuario)
    // Salida automática y bonita: Usuario(nombre=Alex, id=1)
}
```

Esto es indispensable cuando estás haciendo pruebas o guardando *logs* en tu servidor.

7.2. Comparar instancias (==)

Si tú comparas dos instancias distintas de una clase normal usando `==`, Kotlin te dirá que son diferentes `false`, porque mirará si ocupan el mismo espacio físico en la memoria del ordenador.

Sin embargo, en las data classes, el operador `==` (que llama a la función `equals()`) es inteligente: **compara los datos reales**.

```
fun main() {
    val user1 = Usuario("Alex", 1)
    val user2 = Usuario("Alex", 1)
    val user3 = Usuario("Max", 2)

    println("user1 y user2: ${user1 == user2}") // true (¡Tienen los mismos datos!)
    println("user1 y user3: ${user1 == user3}") // false
}
```

7.3. Copiar instancias (copy)

Imagina que quieres crear un objeto nuevo partiendo de uno existente, pero alterando únicamente uno de sus datos. Modificar el objeto original usando un `var` puede causar fallos encadenados si otra parte de tu código estaba usando a ese mismo usuario.

La función `copy()` te clona el objeto a la perfección, y como parámetros opcionales le puedes pasar el nuevo valor para las propiedades que quieras alterar:

```
fun main() {
    val usuario = Usuario("Alex", 1)

    // Clonación exacta
    println(usuario.copy())
    // Usuario(nombre=Alex, id=1)

    // Clonación con el nombre alterado
    println(usuario.copy(nombre = "Max"))
    // Usuario(nombre=Max, id=1)
}
```

8. Ejercicios

Aquí tienes varios ejercicios de menos a más dificultad (los tres primeros son traducciones de la documentación oficial de Kotlin y los dos últimos son extras para asentar los conceptos). Abre tu Kotlin Playground y vamos a mancharnos las manos.

8.1. El empleado

Define una *data class* llamada `Employee` con dos propiedades: `name` (para el nombre en formato texto) y `salary` (para el salario en enteros). Asegúrate de que la propiedad del salario sea **mutable**, ¡o de lo contrario el empleado jamás podrá recibir un aumento a final de año!

El `main` a continuación te muestra cómo lo vamos a usar.

```
// Escribe tu código aquí

fun main() {
    val emp = Employee("Mary", 20)
    println(emp)
    emp.salary += 10
    println(emp)
}
```

8.2. Estructuras anidadas

A veces los objetos contienen a otros objetos. Declara las *data classes* adicionales que se necesitan para que el siguiente código compile sin errores. Fíjate bien en la llamada dentro de `main()` para adivinar qué propiedades y de qué tipo necesita cada clase.

```
data class Person(val name: Name, val address: Address, val ownsAPet: Boolean = true)
// Escribe tu código aquí:
// data class Name(...)

fun main() {
    val person = Person(
        Name("John", "Smith"),
        Address("123 Fake Street", City("Springfield", "US")),
        ownsAPet = false
    )
    println(person)
}
```

8.3. Generador aleatorio de empleados

Para probar tu código en el futuro, vas a necesitar un generador que cree empleados al azar. Define una clase normal `RandomEmployeeGenerator`.

El constructor de la clase debe recibir el `minSalary` y el `maxSalary` (ambos enteros y mutables).

Dentro del cuerpo de la clase, guarda una lista fija de nombres potenciales. Y, por último, crea una función miembro llamada `generateEmployee()` que devuelva una

nueva instancia de tu *data class* `Employee` usando un nombre aleatorio de tu lista y un salario aleatorio dentro de los márgenes dados.

Pista: Las listas de Kotlin tienen una extensión muy útil llamada `.random()` que te devuelve un ítem al azar. Usa `Random.nextInt(from = minSalary, until = maxSalary)` para generar el número.

```
import kotlin.random.Random

data class Employee(val name: String, var salary: Int)

// Escribe tu clase RandomEmployeeGenerator aquí

fun main() {
    val empGen = RandomEmployeeGenerator(10, 30)
    println(empGen.generateEmployee())
    println(empGen.generateEmployee())

    empGen.minSalary = 50
    empGen.maxSalary = 100
    println(empGen.generateEmployee())
}
```

8.4. El inventario de la tienda

Crea una clase (¡normal, no data class!) llamada `Producto` que reciba en su constructor un `nombre` de tipo String y un `precio` de tipo Double (ambos inmutables).

Añade una propiedad **dentro del cuerpo** de la clase llamada `stock` de tipo entero, inicializada en `0`, que por supuesto sea mutable.

Crea una función miembro llamada `añadirStock(cantidad: Int)` que le sume esa cantidad al stock actual.

En tu función `main`, crea un producto, añádele 50 unidades de stock, e imprime un texto final diciendo: *«Tenemos [stock] unidades de [nombre]»*.

8.5. Clonando clientes

Tienes una `data class Cliente(val id: Int, val nombre: String, val email: String)`. Fíjate en que todo está blindado y es inmutable (`val`).

Crea una instancia para «Ana» con ID 1 y correo «». De repente, Ana te avisa de que ha cambiado su correo corporativo a «».

Como no puedes hacer `ana.email = ...` porque daría error, usa la función de las *data classes* que hemos aprendido para crear un nuevo objeto llamado `anaActualizada` basado en Ana, pero con su correo modificado. Imprime por consola a `anaActualizada`.

9. Soluciones a los ejercicios

No hagas trampas. Échale un vistazo a las soluciones únicamente cuando te hayas peleado un buen rato con tu editor de código y el compilador te haya gritado un par de veces.

9.1. El empleado

```
data class Employee(val name: String, var salary: Int)

fun main() {
    val emp = Employee("Mary", 20)
    println(emp)
    emp.salary += 10
    println(emp)
}
```

9.2. Estructuras anidadas

```
data class Person(val name: Name, val address: Address, val ownsAPet: Boolean = true)
data class Name(val first: String, val last: String)
data class Address(val street: String, val city: City)
data class City(val name: String, val countryCode: String)

```

```
fun main() {
    val person = Person(
        Name("John", "Smith"),
        Address("123 Fake Street", City("Springfield", "US")),
        ownsAPet = false
    )
    println(person)
}
```

9.3. Generador aleatorio de empleados

```
import kotlin.random.Random

data class Employee(val name: String, var salary: Int)

class RandomEmployeeGenerator(var minSalary: Int, var maxSalary: Int)
{
    // Definimos la lista en el cuerpo de la clase
    val names = listOf("John", "Mary", "Ann", "Paul", "Jack",
"Elizabeth")

    // Función miembro que devuelve un Employee
    fun generateEmployee() = Employee(
        names.random(),
        Random.nextInt(from = minSalary, until = maxSalary)
    )
}

fun main() {
    val empGen = RandomEmployeeGenerator(10, 30)
    println(empGen.generateEmployee())
    println(empGen.generateEmployee())

    empGen.minSalary = 50
    empGen.maxSalary = 100
    println(empGen.generateEmployee())
}
```

9.4. El inventario de la tienda

```
class Producto(val nombre: String, val precio: Double) {
    // Propiedad en el cuerpo inicializada a 0
    var stock: Int = 0

    fun añadirStock(cantidad: Int) {
```

```
            stock += cantidad
        }
    }

    fun main() {
        val miProducto = Producto("Teclado Mecánico", 45.99)
        miProducto.añadirStock(50)

        println("Tenemos ${miProducto.stock} unidades de
    ${miProducto.nombre}")
    }
```

9.5. Clonando clientes

```
    data class Cliente(val id: Int, val nombre: String, val email:
    String)

    fun main() {
        // Creamos nuestra instancia original
        val ana = Cliente(1, "Ana", "ana@correo.com")

        // Clonamos el objeto modificando únicamente la propiedad email
        val anaActualizada = ana.copy(email = "ana.nueva@empresa.com")

        // Comprobamos la magia
        println(anaActualizada)
        // Salida: Cliente(id=1, nombre=Ana, email=ana.nueva@empresa.com)
    }
```