79. Clases

Se declara con la palabra clave *Class*

```
class Person { /*...*/ }
```

Una clase es un agregado de datos y funciones: **propiedades** (también se pueden llamar **atributos**) y **metodos**.

79.1 Componentes de una clase

- Variables: son propiedades que almacenan valores, pueden ser mutable o inmutables.
- Métodos: son acciones que realizan una tarea determinada.

79.2 Acceso y Modificación

Las propiedades pueden modificarse con la notación punto:

```
val obj = MyClass()
val value = obj.myProperty  // Acceso
obj.myProperty = newValue   // Modificación (sólo si es mutable)
```

Esta notación punto equivale al **get** y **set** en Java.

79.3 Visibilidad de propiedades y métodos.

1. `public` **(predeterminado)**:
2. La propiedad es visible en todas partes.

```
public var name: String = "John Doe"
```

3. **private**:

La propiedad es visible solo dentro de la clase que la contiene (o en el archivo fuente si es una propiedad de nivel superior).

1. **protected**:

La propiedad es visible dentro de la clase que la contiene y en sus subclases. No tiene sentido a nivel de fichero a diferencia del caso de private.

1. **internal**:

La propiedad es visible dentro del mismo módulo. (Sólo en kotlin no existe en Java)

En Kotlin, a diferencia de Java, no existe el modificador `protected` para propiedades de nivel superior, y el modificador internal es específico de Kotlin y permite una visibilidad restringida a todo el módulo donde se declara la propiedad.

79.4 getter y setter personalizados

En Kotlin, el compilador genera automáticamente los métodos getter y setter para las propiedades definidas en las clases. Estos métodos permiten acceder y modificar las propiedades respectivamente. La notación de punto se utiliza para acceder a estas propiedades, lo cual invoca implícitamente los métodos getter y setter. Por ejemplo, si tenemos una clase Persona con una propiedad nombre, podemos acceder a esta propiedad con persona.nombre y asignarle un valor con persona.nombre = "Nuevo Nombre".

Kotlin permite definir getter y setter personalizados para las propiedades:

```
class MyClass {
    var myProperty: Int = 3
        get() {
            // Código personalizado para el getter
        }
        set(value) {
            // Código personalizado para el setter
        }
}
```

Ejemplo:

```
class Person {
    var name: String = "John Doe"
        get() = field.capitalize()
        set(value) {
            field = value.trim()
        }
}
```

* las funciones get() y set() se definen a continuación de la propiedad. * La palabra clave **field** solo se puede usar dentro de la definición del getter y setter y se refiere a la propiedad.

79.5 Constructores

Las clases kotlin tienen un **constructor primario** y opcionalmente uno o varios constructores secundarios.
El constructor primario se declara en la cabecera de la declaración de la clase:

```
class Person constructor(firstName: String) { /*...*/ }
```

Y si el constructor primario no tiene modificadores ni anotaciones puede omitirse la palabra **constructor**

```
class Person(firstName: String) { /*...*/ }
```

El constructor primario crea la instancia e inicializa las propiedades.
No puede contener ningún código ejecutable , si es necesario algún tipo de inicialización se utilizan bloques **init{}**

```
class Estudiante(val nombre: String, val apellido: String) {
    val id: String
    init {
        id = "$nombre-$apellido"
    }
}
```

Los parámetros del constructor primario se pueden utilizar en los bloques de inicialización y en la inicialización de las propiedades:

```
class Customer(name: String) {
    val customerKey = name.uppercase()
}
```

En Kotlin, los parámetros del constructor primario pueden convertirse automáticamente en propiedades de la clase si se les precede con las palabras clave **val** o **var**. Esto es una característica útil que permite una definición más concisa de las clases. Aquí hay una explicación más detallada:

- Con val o var: Si precedes los parámetros del constructor primario con val o var, estos parámetros se convierten automáticamente en propiedades inmutables o mutables de la clase, respectivamente.

```
class Person(val name: String, var age: Int)
```

En este ejemplo, name es una propiedad inmutable y age es una propiedad mutable de la clase Person.

En el constructor se pueden añadir la visibilidad de los parámetros:

```
class Person(private val name: String, private var age: Int)
```

- Sin val o var: Si los parámetros del constructor primario no están precedidos por val o var, entonces no se convierten en propiedades de la clase y solo están disponibles dentro del cuerpo del constructor.

```
class Person(name: String, age: Int) {
    init {
        println("Name is $name, Age is $age")
    }
}
```

En este ejemplo, **name** y **age** no se convierten en propiedades de la clase Person, solo son accesibles dentro del bloque init y otros bloques de inicialización en el cuerpo de la clase.

79.5.1 Constructor secundario

La clase puede tener un constructor secundario añadiendo el prefijo **constructor**

```
class Person(val name: String) {
    val children: MutableList<Person> = mutableListOf()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

`Cuando la clase tiene constructor primario, el constructor secundario debe llamar directamente o indirectamente al primario`

Ejemplo:

```
  class Estudiante(val nombre: String) { // Constructor primario
```

```
    var edad: Int = 0

    // Constructor secundario que llama directamente al constructor primario
    constructor(nombre: String, edad: Int) : this(nombre) {
        this.edad = edad
    }
}
fun main() {
    val estudiante1 = Estudiante("Ana") // Usa el constructor primario
    val estudiante2 = Estudiante("Carlos", 20) // Usa el constructor secundario
}
```

El constructor primario se referencia con *this*
En este ejemplo se muestra una forma indirecta de incializar/delegar en el constructor primario:

```
class Person(val name: String) {
    val children: MutableList<Person> = mutableListOf()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

NOTA: Los parámetros del constructor secundario no se convierten en propiedades de la clase aunque estén precedidas por **var** o**val**

Tanto el constructor primario como secundario pueden tener valores por defecto.

79.6 Instancias de clase

Se crean como las funciones normales:

```
val invoice = Invoice()
```

```
val customer = Customer("Joe Smith")
```

(No se utiliza 'new' como en Java.)

79.7 Miembros de la clase

Las clases pueden contener: * Constructores y bloques de inicialización * Funciones (mètodos) * Propiedades * Clases interiores e inicializadas * Declaraciones de objetos

79.8 Tipos de clases

79.8.1 Data Class

En Kotlin, las `data classes` están diseñadas para almacenar datos cuando no necesitan código.
Se generan automáticamente varios métodos útiles cuando declaras una clase como una data class. Estos métodos incluyen `toString()`, `hashCode()`, `equals(other: Any?)`, `copy()` y los métodos de componentes `componentN()`.

Aquí está el detalle de los métodos que puedes y no puedes sobreescribir en una data class:

1. **`toString()`**:
2. Puedes sobreescribir este método si deseas proporcionar una representación en cadena personalizada de la instancia de tu data class.
3. **`hashCode()`** y **`equals(other: Any?)`**:
4. También puedes sobreescribir estos métodos si deseas proporcionar una implementación personalizada para la comparación de igualdad y la generación de hashcodes.
5. **`copy()`**:

6. Este método no se puede sobreescribir. Es generado automáticamente y proporciona una forma de crear una nueva instancia de la data class, los valores usados en la llamada se sobreescriben, el resto quedan igual. En el siguiente ejemplo original y update tienen todos los datos igual salvo "age"

```
data class User(val name: String, val age: Int)

fun main() {
    val original = User(name = "Juan", age = 30)
    val updated = original.copy(age = 31)
    println(original) // Output: User(name=Juan, age=30)
    println(updated) // Output: User(name=Juan, age=31)
}
```

1. **Métodos de componentes (`componentN()`)**:
2. Estos métodos tampoco se pueden sobreescribir. Son generados automáticamente y proporcionan una forma de desestructurar la instancia de la data class en variables individuales.

Es importante mencionar que sobreescribir los métodos `equals`, `hashCode`, y `toString` en una data class es una práctica que debe ser manejada con cuidado, ya que estos métodos son generados automáticamente para trabajar bien con las propiedades de la data class y proporcionar comportamientos útiles y coherentes. Si decides sobreescribir estos métodos, asegúrate de entender bien cómo funcionan y cómo deberían interactuar con tu data class.

79.8.2 Object

Declaración para el patrón *singleton*. Este patrón se utiliza cuando se necesita una **única instanci**a de un objeto. Un ejemplo típico es cuando abrimos una conexión a un servidor de base de datos

```
import java.sql.Connection
import java.sql.DriverManager
import java.sql.SQLException
```

```
object DatabaseManager {
    private const val URL = "jdbc:mysql://localhost:3306/mydatabase"
    private const val USERNAME = "username"
    private const val PASSWORD = "password"

    var connection: Connection? = null
        private set

    init {
        try {
            Class.forName("com.mysql.jdbc.Driver")
            connection = DriverManager.getConnection(URL, USERNAME, PASSWORD)
        } catch (e: ClassNotFoundException) {
            println("Driver no encontrado: ${e.message}")
        } catch (e: SQLException) {
            println("No se pudo conectar a la base de datos: ${e.message}")
        }
    }

    fun closeConnection() {
        try {
            connection?.close()
        } catch (e: SQLException) {
            println("Error al cerrar la conexión: ${e.message}")
        }
    }
}
```

Que se utilizaría:

```
fun main() {
    val query = "SELECT * FROM users"
    val statement = DatabaseManager.connection?.createStatement()

    try {
        val resultSet = statement?.executeQuery(query)
```

```
        while (resultSet?.next() == true) {
            val userId = resultSet.getInt("user_id")
            val userName = resultSet.getString("user_name")
            println("User ID: $userId, User Name: $userName")
        }

    } catch (e: SQLException) {
        println("Error al ejecutar la consulta: ${e.message}")
    } finally {
        statement?.close()
        DatabaseManager.closeConnection()
    }
}
```

Se podrían llamar varias veces a *DatabaseManager* pero solo se crearía un objeto siguiendo el patrón singleton y de esto se ocupa el compilador kotlin.

79.8.3 Enum Class

Tipo enumerado:

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

Cada constante enumerado es una clase y pueden ser inicializados:

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

Para usar los enumerados.

En kotlin las clases Enum implementan métodos para listar las constantes enumeradas.:

```
EnumClass.valueOf(value: String): EnumClass
EnumClass.values(): Array<EnumClass>
```

Por ejemplo:

```
enum class RGB { RED, GREEN, BLUE }

fun main() {
    for (color in RGB.values()) println(color.toString()) // prints RED, GREEN, BLUE
    println("The first color is: ${RGB.valueOf("RED")}") // prints "The first color is: RED"
}
```

79.9 Herencia de clases

Todo tipo en Kotlin, ya sea un tipo definido por el usuario o un tipo predefinido, hereda de la clase **Any**. La clase Any proporciona tres métodos: **toString()**, **hashCode()**, y **equals(other: Any?): Boolean**, que están disponibles para (sobreescribir) todas las instancias de cualquier clase en Kotlin.

```
public open class Any {
    public open operator fun equals(other: Any?): Boolean
    public open fun hashCode(): Int
    public open fun toString(): String
}
```

Estos métodos pueden ser sobrescritos por cualquier clase derivada. Por ejemplo, puedes sobrescribir el método toString() para proporcionar una representación de cadena personalizada de una instancia de tu clase, o sobrescribir equals() para proporcionar una comparación de igualdad personalizada entre instancias de tu clase.

79.9.1 Control en la herencia de clases

Las clase puede ser **final**, que significa que no pueden derivarse clases (es el valor por defecto) o bien **open** y entonces puden tener clases derivadas [TODO]

La clase derivada referencia a la superclase poniendo el tipo despues de ":" y los parámetros del constructor.

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

Sobreescribir métodos -override-
Sólo se pueden sobreescribir los métodos marcados como **open**

```
open class Base {
    open fun v() {}
    open val x: Int get() = 1
}

class Derived() : Base() {
    override fun v() {}
    override val x: Int get() = super.x + 1
}
```

Sobreescribir propiedades
Igual que los métodos:

```
open class Shape {
    open val vertexCount: Int = 0
}

class Rectangle : Shape() {
```

```
    override val vertexCount = 4
}
```

Al sobreescribir la propiedad declarada con `val` puede sobreescribirse con `var` pero no al revés.

Llamar a métodos y propiedades de la clase base

Se usa la palabra reservada **super**

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

class FilledRectangle : Rectangle() {
    override fun draw() {
        super.draw()
        println("Filling the rectangle")
    }

    val fillColor: String get() = super.borderColor
}
```

79.9.2 Clases genéricas o parametrizadas

Igual que en Java

```
class Box<T>(t: T) {
    var value = t
}
```

y para crear objetos:

```
val box: Box<Int> = Box<Int>(1)
```

Incluso se puede omitir el tipo `T` si el compilador puede inferir el tipo:

```
val box = Box(1) // 1 has type Int, so the compiler figures out that it is Box<Int>
```

79.10 Interfaces

Similares a Java. Una clase puede implementar uno o varios intefaces:

```
// Definición de la primera interfaz
interface InterfaceA {
    fun doSomethingA()
}

// Definición de la segunda interfaz
interface InterfaceB {
    fun doSomethingB()
}

// Implementación de las dos interfaces en una clase
class MyClass : InterfaceA, InterfaceB {
    // Implementación del método de InterfaceA
    override fun doSomethingA() {
        println("Haciendo algo en A")
    }

    // Implementación del método de InterfaceB
    override fun doSomethingB() {
        println("Haciendo algo en B")
    }
}
```

79.10.1 Propiedades en Inteface

En los interfaces se pueden delarar propiedades que pueden ser abstractas o proporcionar getter/setter (No se puede inicializar directamente)

```
interface MyInterface {
    val prop: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

79.11 Apendice

Enlaces: * https://kotlinlang.org/docs/classes.html

Versión v1.0, 1-11-23