TEMA 2.- INTRODUCCIÓN A KOTLIN

PROGRAMACIÓN MULTIMEDIA y DISPOSITIVOS MOVILES

Roberto Sánchez de la Rosa
rsanchezro@educa.jcyl.es

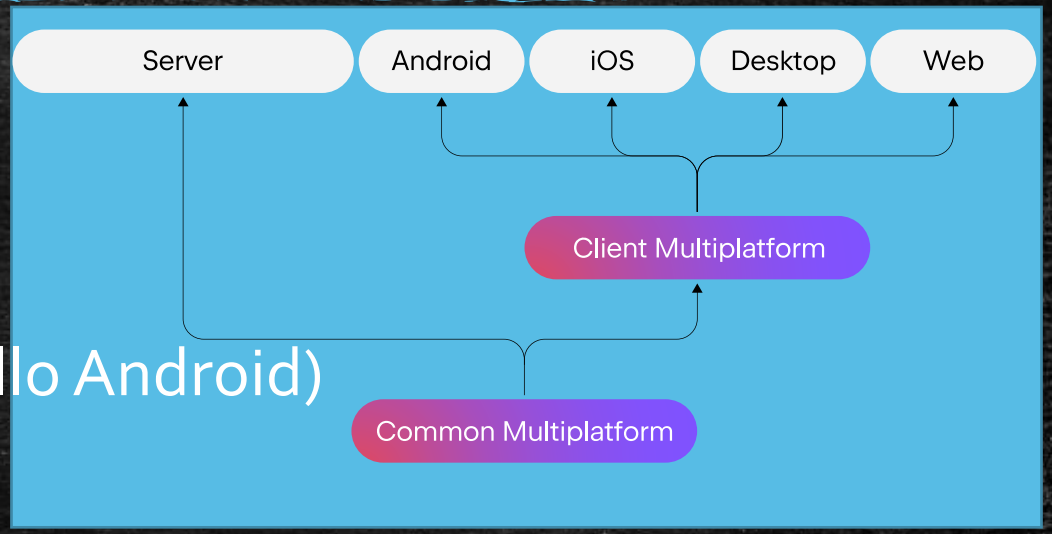
CURSO 2024-2025
2º D.A.M
IES RIBERA DE CASTILLA

TEMA 2.- INTRODUCCIÓN A ANDROID

1. INICIOS EN KOTLIN
2. TOUR EN KOTLIN
3. ARRAYS
4. ESTRUCTURAS DE CONTROL
5. FUNCIONES
6. LAMBDAS
7. POO
8. GENERICOS
9. COLECCIONES
10. FUNCIONES DE ÁMBITO

1. INICIOS EN KOTLIN

- WEB OFICIAL (Comencemos)
- Primeros Pasos en una hora (video)
- CARACTERISTICAS
  - Multipropósito (no solamente para desarrollo Android)
  - Multiplataforma
  - Interoperable con Java
  - Inferencia de tipos
  - Principalmente para trabajar con la JVM, pero existen transpiladores y compiladores para desarrollo en otras plataformas (aplicaciones nativas, aplicaciones javascript, aplicaciones IOS, etc..)
  - Aplicación Android puede convertirse en IOS.

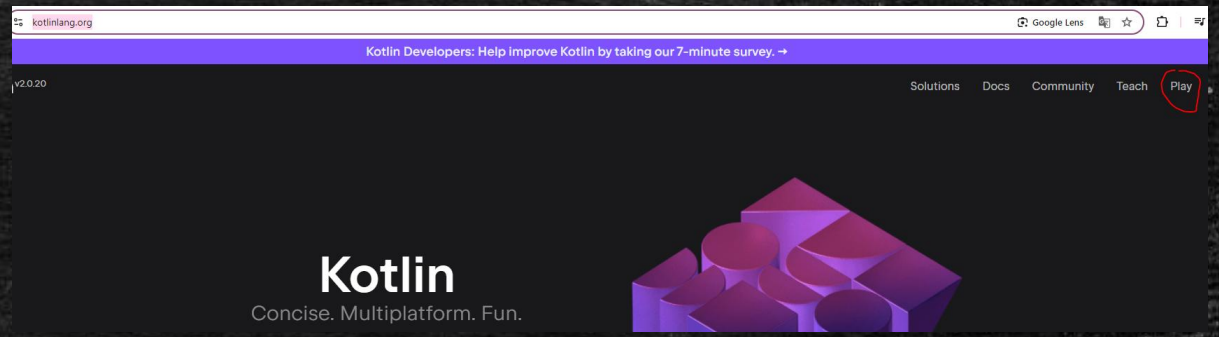


1. INICIOS EN KOTLIN

- INSTALACIÓN:
  - JVM (necesario el JDK)
  - Editor o IDE (En Android Studio ya instala todo lo necesario)
    - Se podría instalar por separado (compilador kotlinc),
      - 1º descarga (necesario jdk) (scroll hacia abajo, kotlin_compiler.zip)
      - 2º compilar: kotlinc fuente extensión kt -include-runtime -d archivo.jar
      - Ejecutar (maquina virtual de java)
    - Ejercicio: Hola Mundo, compilar y ejecutar

1. INICIOS EN KOTLIN

- IDE WEB PLAYGROUND KOTLIN (pruebas online)
- HOLA MUNDO-TOUR WEB OFICIAL
  - Estructura de un programa Kotlin
  - La extensión de los fuentes .kt
- TIPOS DE DATOS
  - No existen tipos primitivos (todo son clases)
  - Lenguaje inferenciado (al asignar un valor implícitamente se le da un tipo)
  - Ver los tipos de datos



Consejo: Usa el tipo adecuado

1. INICIOS EN KOTLIN

- APRENDE EL LENGUAJE KOTLIN DESDE 0 – MANUAL ANDROID STUDIO

2. TOUR EN KOTLIN

- VARIABLES.- Espacios de memoria, lenguaje fuertemente tipado, las variables no cambian de tipo en su ciclo de vida

```
var nombre[: Tipo] [= valor]
```

- CONSTANTES.- Solo se pueden inicializar una vez (en la declaración o posterior)

```
val nombre[: Tipo] [= valor]
```

- Inferencia de tipos.- El lenguaje deduce el tipo

```
var numero=12 //Infiere el tipo a Int
```

Consejo: Indica siempre el tipo

2. TOUR EN KOTLIN

- Comprobación de Tipos, operador is
- STRINGS
  - Concatenación
    - Operador +
    - String Templates ($).- Implícitamente provoca una invocación al método toString. Sintaxis:

```
var edad:Int=16
println( "La edad de Juan es $edad")
println("La potencia de $base ^ $exp es: ${potencia(base,exp)}
```

$*nombre variable* **ó** ${expresión}

    - String.format().- Para dar formato al string
  - toString().- Igual que en Java

2. TOUR EN KOTLIN

- Nulos en KOTLIN
  - Kotlin NO permite almacenar por defecto null en los objetos, si un objeto va a almacenar nulos es necesario declararlo con **?**

    {var|val} *nombre*[: *Tipo*]? [= *valor*]
  - Si se intenta asignar a una variable no null un valor null, error en tiempo de compilación
  - Operador de llamada segura (call safe) **?.** , si la variable es null devuelve null, en otro caso se accede a la propiedad/método

    Objeto?.propiedad/método
  - Operador de aserción not null **!!** , estas seguro al 100% que no va a ser null, si es null, arroja Null PointerException

    objeto!!propiedad/método

2.TOUR EN KOTLIN

- Nulos en KOTLIN
  - Operador Elvis ?: Comprueba la nulidad de una variable, si es null devuelve el valor indicado después del operador (similar a un if)

```
Objeto?propiedad?:valor_si_null
```

2. TOUR EN KOTLIN

- Entrada y Salida: println y readln
- Check (operador is) y Cast (operador as)
  - Smart Cast.- Cuando se chequea un tipo se hace un cast implícito
  - Safe Cast.- Cuando se intenta castear un tipo y no funciona

En Kotlin TODAS las variables son OBJETOS

3. ARRAYS

- Arrays.- Varias formas para su definición:
  - Más habitual

```
val <nombre>[: Array<tipo>] [= arrayOf(listadovalores)]
```

  - CONSTRUCTOR

```
val <nombre>[: Array<tipo>] [=Array(num_elem){valorinicial/expresión}]
```

  - Método joinToString().- Genera un string con todos los valores
  - Acceso a valores de arrays igual que en Java [indice]
  - Cuando usar arrays?
  - Comparar Arrays

3. ARRAYS

- Función map, permite obtener una colección a partir de un array

```
fun main(args: Array<String>) {

val numbers = listOf(1, 2, 3)
println(numbers.map { it })
for(numero in numeros$)
{
		println(numero)
}
```

- Arrays de tipos primitivos
  - Se pueden convertir Arrays a tipos primitivos y viceversa.

4. ESTRUCTURAS DE CONTROL

- ESTRUCTURAS SELECTIVAS
  - IF.- Sintaxis similar a Java, aunque incluye diferentes posibilidades
    - El resultado de una sentencia **if** se puede asignar a una variable, obligatoria la rama else

```
val max = if (a > b) {
            print("Choose a")
            a
            }
else {
            print("Choose b")
            b
}
```

  - WHEN.- Similar a switch de Java, con sintaxis diferente
    - El resultado se puede asignar a variable
    - En las **ramas** se puede usar **operador in**
      - O el **operador is** para comprobar el tipo

```
in valor_inicial..valor_final
ó
in array_valores
```

```
is tipo -> -> [{] sentencias [}]
```

EJEMPLOS WHEN

```
when [( variable o expresión)]{
            valor1 o expr1 -> [{] sentencias [}]
            valor2 o expr2-> [{] sentencias [}]
            .......
            else -> [{] sentencias [}]
}
```

4. ESTRUCTURAS DE CONTROL

- ESTRUCTURAS REPETITIVAS
  - For.- similar a foreach de Java, útil para los siguientes casos (no sólo):
    - Procesar colecciones
    - Validar entradas
    - Generar interfaces de usuario dinámicas
    - Operaciones con arrays y gráficos (videojuegos)

VER EJEMPLOS

..: Keyword operator

Sintaxis:

```
for ( variable [:Tipo] in colección )
for ( variable in rango)
for ( (index,valor) in array.withIndex())
```

Rango es: valorinicial {..|downTo} [<]Valorfinal [step valor]

4. ESTRUCTURAS DE CONTROL

- ESTRUCTURAS REPETITIVAS
  - While y do-while.- Sintaxis similar a Java.

5. FUNCIONES

- Sintaxis definición:

```
fun nombre([parámetros,...])[:tipodevuelto]{
// Instrucciones   }
```

  - **Parámetros**.- Notación Pascal

```
nombre:tipo [ = valordefecto]
```

  - Tipos devueltos:
    - Unit.- Similar al void de Java, por defecto si no se indica devuelve Unit
    - Any.- Similar al Object de Java, clase raíz de todas las clases
  - Invocación.-
    - Puedes saltar el orden de los parámetros

5. FUNCIONES

- Funciones con una única sentencia, no necesario tipo devuelto

```
fun <nombre> ([parámetros,...])[:tipo ]= sentencia
```

- Funciones con número de parámetros variable, el último que se defina, marcado con la palabra reservada ***vararg***

```
fun <nombre> ([...,vararg parámetro_multiple])[:tipo ]{ }
```

  - Solo un parámetro puede marcarse como múltiple
  - Puedes pasar múltiples valores o incluso un Array (no primitivo), anteponiendo el operador spread al nombre del array (*)

VER EJEMPLOS

5. FUNCIONES

- AMBITO DE LAS FUNCIONES
  - Funciones Locales.- Funciones definidas dentro de otras funciones, podrán acceder a las variables de la función contenedora.
  - Funciones miembro.- Funciones (métodos en OO) de una clase
- FUNCIONES GENÉRICAS.- Funciones que tienen parámetros genéricos (T). Tanto los parámetros como el tipo devuelto pueden hacer referencia al tipo genérico. Ver ejemplo

```
fun <T> nombre([parámetros,...])[:tipodevuelto]{  // Instrucciones  }
```

```
fun <T> singletonList(item: T): List<T> { /*...*/ }
```

5. FUNCIONES

- AMBITO DE LAS FUNCIONES
  - Funciones Locales.- Funciones definidas dentro de otras funciones, podrán acceder a las variables de la función contenedora.
  - Funciones miembro.- Funciones (métodos en OO) de una clase
- FUNCIONES GENÉRICAS.- Funciones que tienen parámetros genéricos (T). Tanto los parámetros como el tipo devuelto pueden hacer referencia al tipo genérico. Ver ejemplo

```
fun <T> nombre([parámetros,...])[:tipodevuelto]{  // Instrucciones  }
```

```
fun <T> singletonList(item: T): List<T> { /*...*/ }
```

5. FUNCIONES

- Funciones anónimas.- Funciones sin nombre, se suelen asociar a variables para poderlas usar

fun ([parámetros,…])[:tipodevuelto]{ // Instrucciones y sentencia return si hay que devolver algo, se puede inferir el tipo devuelto }
o
fun ([parámetros,…])[:tipodevuelto]= // Instrucción sin return , para devolver algo, se puede inferir el tipo

```
var mifuncion=fun (x:Int,y:Int)=x+y
var mifuncion2=fun(x:Int,y:Int):String
 {
    cad:String=""
    println("valor de x $x")
    return "valor de x * y : ${x*y}"
}
println(mifuncion(4,2))
println(mifuncion2(5,3))
```

EJERCICIOS BÁSICOS KOTLIN

- Ejemplo: función Anagrama(cadenas_anagrama.kt)
- Ejercicios básicos (Ejercicios_Basicos_Kotlin.pdf)

6. LAMBDAS

- QUE ES UN LAMBDA
  - Son funciones que se usan como parámetros en otras funciones.
  - Un nombre propio: Alonzo Church
  - Definición sencilla: Función que se puede usar como argumento en otras funciones. Artículo muy interesante sobre las Lambdas en Java
  - Video explicativo
  - Funciones de orden superior: funciones que tienen como argumentos otras funciones o retornan una función.
  - **CodeLab en AndroidStudio**

findViewById(R.id.button_id);

6. LAMBDAS

- POR QUE LAMBDAS?
  - Forma más sencilla y rápida de definir funcionalidad en callbacks(funciones que se ejecutan en un momento indeterminado)
  - Ejemplo: Cuando se hace click a un botón se ejecuta un método (indirectamente onClick()) que inicialmente esta vacio porque es de una Interface (View) y lo que hacemos en tiempo de edición es redefinir el código de ese método (implementando la interface) para que cuando en tiempo de ejecución se ejecute onClick(), tenga código

```
//En Java
//Instancio el objeto button
Button button = findViewById(R.id.button_id);
MiclaseListener milistener=new MiclaseListener();
//Internamente se define el objeto a otro objeto interno, cuando se pulsa boton se invoca al método onClick() de ese objeto
 button.setOnClickListener(milistener);
```

```
//Defino MiclaseListener
Class MiclaseListener implements View.OnClickListener{
public void onClick(View v) {
      // Aquí va el código cuando se haga click en el botón
      Log.d(TAG, "Se hizo clic en la vista con id: " +
v.getId());
   }
}
```

6. LAMBDAS

- POR QUE LAMBDAS?

```
//En Java
//Lo mismo de antes de otra forma más reducida
Button button = findViewById(R.id.button_id);
 button.setOnClickListener(new View.OnClickListener() {
          public void onClick(View v) {
             // Codigo que se ejecuta cuando pulso boton
          }
       });
```

Podemos hacer esto más sencillo?

SI con LAMBDAS

```
//En Kotlin
//Lo mismo de antes de otra forma más reducida
button:Button= findViewById<Button>(R.id.button_id);
 button.setOnClickListener(View.OnClickListener() {
          public void onClick(View v) {
             // Codigo que se ejecuta cuando pulso boton
          }
       });
```

6. LAMBDAS

- Son funciones que no son declaradas (anónimas), son directamente definidas como una expresión.
- DOS USOS FUNDAMENTALMENTE:
  - PRIMERO.- Asignando a una variable la función. SINTAXIS

```
{var|val} variable [:tipofuncion]={ [parámetros->]sentencias }
//ultima sentencia el valor de retorno sin return
```

    - *Tipofuncion*: A continuación
    - Si el tipo de retorno no es Unit, la última instrucción será tratada como el valor devuelto (sin return)
    - Posteriormente se podrá usar la variable con notación de función, para invocar a la función

6. LAMBDAS

Si uno de los parámetros lambda no se va a usar se pone el _ en su lugar

- DOS USOS FUNDAMENTALMENTE:
  - SEGUNDO.- Pasando la función como un argumento de una función(función de orden superior), se puede pasar una variable que almacena una función.
    - SINTAXIS INVOCACIÓN

```
Funcion_orden_superior({[parametros->]sentencias})
```

  - Si es el último argumento la función que se pasa, la expresión lambda se suele poner fuera de los paréntesis

```
Funcion_orden_superior() {[parametros]->sentencias}
```

  - Si solo hay un parámetro no es necesario especificarlo, se puede hacer referencia a él mediante el operador **it**, por lo que en ese caso se puede eliminar el símbolo ->, quedando la invocación

```
Funcion_orden_superior() {sentencias}
```

  - Si la función de orden superior no tiene más parámetros se pueden eliminar los paréntesis en la invocación

```
Funcion_orden_superior {sentencias}
```

6. LAMBDAS

- Retornando valores
  - Sin sentencia return (por defecto), ultima instrucción es el valor de retorno
  - Con sentencia return.- Necesario etiquetar con el mismo nombre que la función (return@nombrefuncion variable_a_retornar)
- Dos ptos. de vista:
  - **Invocación de la función de orden superior**.- Es donde hay que **definir el código de la función lambda**
  - **Definición de la función de orden superior**.- Es el **lugar donde se invoca a la función lambda**

6. LAMBDAS

- Función como un tipo de dato.- Cuando un parámetro de una función es otra función, en su definición se indica el tipo. Sintaxis:

```
([[nombre_parámetro:]tipo_dato,...])->tipo_dato
```

  - **Explicación**: Lo que hay dentro de los paréntesis representan parámetros, el operador -> separa los parámetros del tipo de dato que devuelve la función.

```
Ejemplo: (Int,String)->Boolean
```

  - Opcionalmente pueden tener nombre los parámetros

```
Ejemplo: (a:Int,b:String)->Boolean
```

  - La lista de parámetros puede estar vacía

```
Ejemplo: ()->String
```

En los tipos de funciones el tipo de retorno no se puede omitir, si no devuelve nada poner Unit

6. LAMBDAS

- Función como un tipo de dato (cont.)
  - Los tipos de función, opcionalmente, pueden tener un tipo de dato receptor (clase), utilizando la notación **punto** (similar a las extensiones)

Ejemplo: *(String.(Int,String)->Boolean)*
// función que te dice si el objeto String contiene otro String(parametro) a partir de una posicion (Ejemplo_lambda_extension.kt
Similar a: *((String,Int,String)->Boolean)*

  - Explicación: Permite definir una función lambda en la clase String, se puede invocar con un objeto String

6. LAMBDAS: FUNCIONES DE ORDEN SUPERIOR

- Son funciones que tienen como parámetro otra función o que retornan una función
- Definición de la función de orden superior

```
fun nombre_función([parámetros],nombre_param_lambda:tipo_dato_función)[:tipodevuelto]
{//Sentencias }
```

- Dentro de la función de orden superior se invoca a la función definida como lambda

EJEMPLOS LAMBDA

VER EJEMPLOS:

- Funciones de orden superior
  - Función fold
- Lambda_variable
- Lambda_function
- Funcion_orden_superior
- Ejemplo con filter de ArrayList

EJERCICIOS LAMBDA

- Ejercicios Lambdas(EjerciciosLambda.pdf)

7.POO CON KOTLIN

- Seguir este CodeLab (ejercicio casi resuelto) y la documentación oficial
- Definición de una clase (versión simple)

```
class nombreclase{ \\cuerpo de la clase }
```

PascalCase en los nombres de las clases, cada palabra comienza en mayúscula

- En el cuerpo de la clase pueden existir:
  - Propiedades
  - Métodos (o funciones)
  - Constructores

7.POO CON KOTLIN

- Crear una instancia de una clase

{val|var} *nombre_objeto*=*Nombre_Clase*(*[argumentos]*}

- Definición de los miembros de una clase
  - FUNCIONES (se vio en funciones).- Tienen que estar dentro de las {}
  - PROPIEDADES (similar a la declaración de variables)
    - Métodos setter y getter.- Por defecto si no se definen, el compilador los crea, SINTAXIS

var *nombre_propiedad*[: *tipo [= valor_inicial*] [*<getter>*] [*<setter>]*

  - getter y setter, sintaxis

get() { \\sentencias e instrucción return sin palabra return}
set(value) {\\asignación del valor a palabra reservada **field**}

7.POO CON KOTLIN

- Definición de los miembros de una clase(cont.)
  - CONSTRUCTORES
    - Constructor por defecto

```
class nombre_clase constructor(){ //cuerpo clase}
```

      - Si no existen anotaciones ni modificadores de visibilidad (se ven más adelante), se puede eliminar **constructor()**
    - Constructor con parámetros

```
class nombre_clase(parámetros){ //cuerpo clase}
```

      - Los parámetros se definen con var o val, sino no son propiedades y se les puede asignar valor por defecto

7.POO CON KOTLIN

- Definición de los miembros de una clase(cont.)
  - CONSTRUCTORES
    - Constructor principal

```
class nombre_clase constructor(parametros){ //cuerpo clase}
```

      - Solo puede haber uno, no puede tener código
    - Constructores secundarios.- Se definen dentro de la clase

```
constructor(parámetros):this(param const principal){ //cuerpo const}
```

      - **this** representa la invocación al constructor principal, si tiene parámetros habrá que pasar valores y/o parámetros del constructor
      - Pueden contener código

7.POO CON KOTLIN

- Bloques Init.- Son bloques que se ejecutan al instanciar un objeto de una clase, no tienen parametros y deben contener pocas líneas de código, se pueden definir múltiples bloques de inicialización

```
init { //código del bloque}
```

- Modificador **lateinit**.-, sirve para indicar que la propiedad se inicializará más tarde.
  - No puede tener getter o setter personalizado
  - No puede ser nullable
  - Solo se pueden usar en propiedades var
  - Se tienen que declarar en el cuerpo de la clase
  - No pueden ser de "tipos primitivos":Double,Float,Int,Long,Char,Boolean
  - Se podrá inicializar su valor en cualquier método de la clase (incluidos constructores secundarios)
  - Propiedad **isInitialized** para saber si la propiedad ha sido inicializada, referencia de la propiedad(operador ::)

```
lateinit var nombrevar:Tipo
```

```
If(!::color.isInitialized) {...
```

7.POO CON KOTLIN:HERENCIA

- Por defecto las clases en Kotlin no admiten herencia, para que una clase pueda ser heredada(clase padre), utilizar el modificador open

```
open class nombre_clase { //cuerpo clase}
```

- SUBCLASES.-

```
class nombre_clase [(parámetros)]:clasepadre ([parámetros]) { //cuerpo clase}
```

  - Los parámetros en el constructor de la clase heredada pueden ser parámetros(sin val o var) o propiedades(var o val)
  - Los parámetros se usan para asignar valores a las propiedades de la clase padre

7.POO CON KOTLIN:HERENCIA

- SOBRECARGA DE MÉTODOS
  - Necesario en la clase padre marcar el método con el modificador **open**

```
open fun metodo_a_sobrescribir { //cuerpo método}
```

  - En la subclase es necesario marcar el método con el modificador **override**

```
class nombre_clase [(parametros)]:clasepadre[(parámetros)]{ //cuerpo clase}
```

- POLIMORFISMO.- Instanciar un objeto de una clase padre con una clase heredada, similar a Java

```
{var|val} nombreObjeto:clasePadre=ClaseHeredada()
```

7.POO CON KOTLIN:HERENCIA

- POLIMORFISMO(cont.).- Si se quiere acceder a propiedades o métodos de la clase hija con un objeto de la clase padre con instancia de la clase hija será necesario realizar un cast

```
var figura:Figura=Rectangulo(2,5)
//Si quiero acceder a una propiedad o método de clase Rectangulo propio será necesario hacer cast
//Imaginemos que solo los rectangulos tienen ancho y alto, para acceder a alto o ancho
(figura as Rectangulo).ancho
```

7.POO CON KOTLIN:HERENCIA

- INVOCACIÓN a métodos CLASE PADRE
  - Palabra reservada **super** .Se invoca dentro del método a sobrescribir (1ª instrucción)

```
super.método([argumentos])
```

  - En la subclase es necesario marcar el método con el modificador **override**
- SOBRESCRIBIR PROPIEDADES
  - Necesario definirlas como **var** en clase padre y marcar la propiedad como **open**
  - En la clase heredada la propiedad hay que marcarla con el modificador **override**

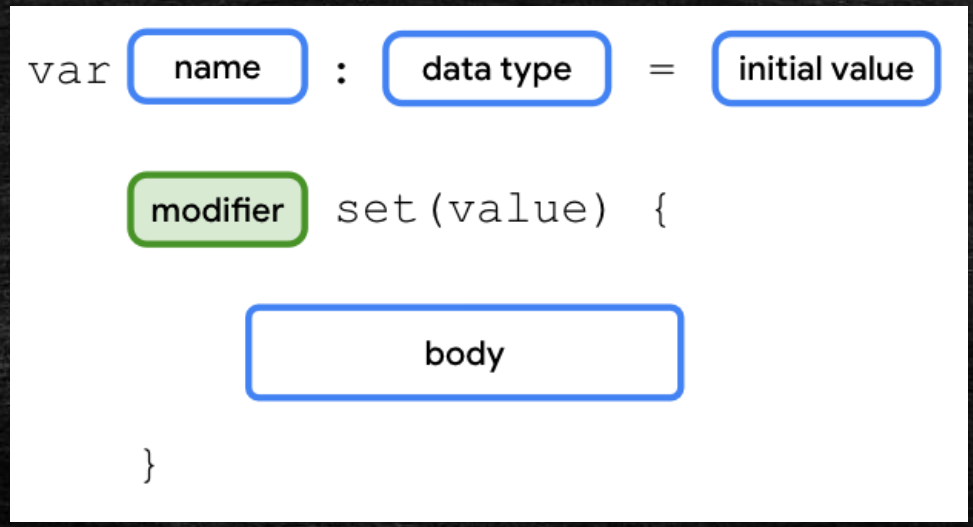
7.POO CON KOTLIN:ENCAPSULAMIENTO

- Existen 4 modificadores de visibilidad que afectan a clases, métodos y propiedades
  - **public.-** Modificador por defecto, accesible desde todos los sitios.
  - **private**.- Se puede acceder solo desde la misma clase
  - **protected**.- Accesible en las subclases
  - **internal.-** Similar al privado pero puedes acceder a ellos si están en el mismo módulo: colección de archivos y configuraciones de compilación. Por encima del concepto de paquete (proyecto en AndroidStudio)

7.POO CON KOTLIN:ENCAPSULAMIENTO

- Modificadores en propiedades

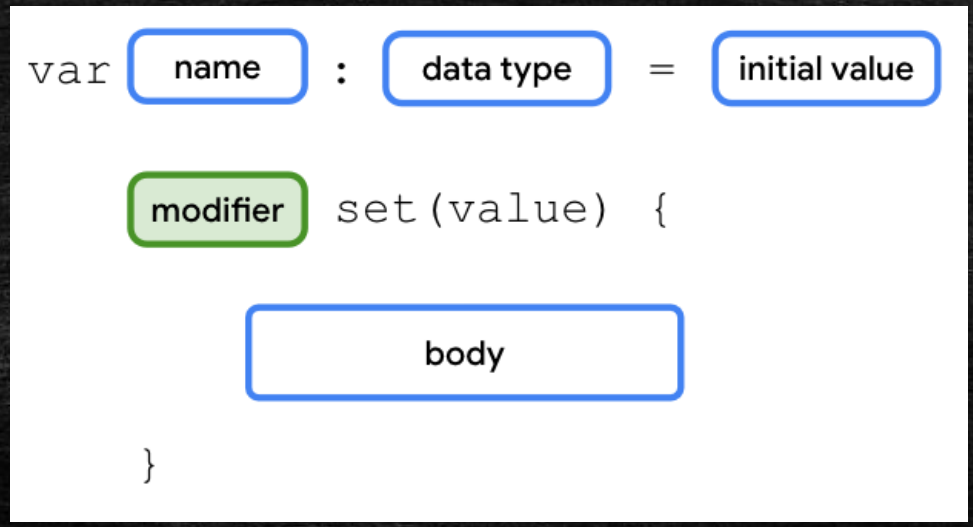
  - En el método **set,** debe tener el mismo modificador que la propiedad



8.POO CON KOTLIN:ENCAPSULAMIENTO

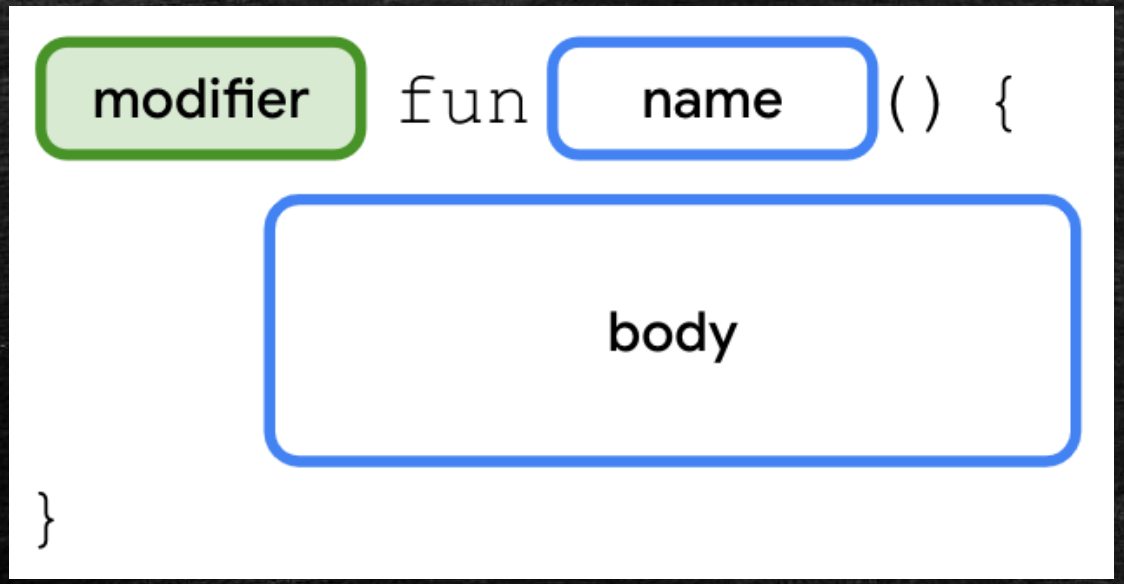
- Modificadores en propiedades

  - En el método **set,** debe tener el mismo modificador que la propiedad



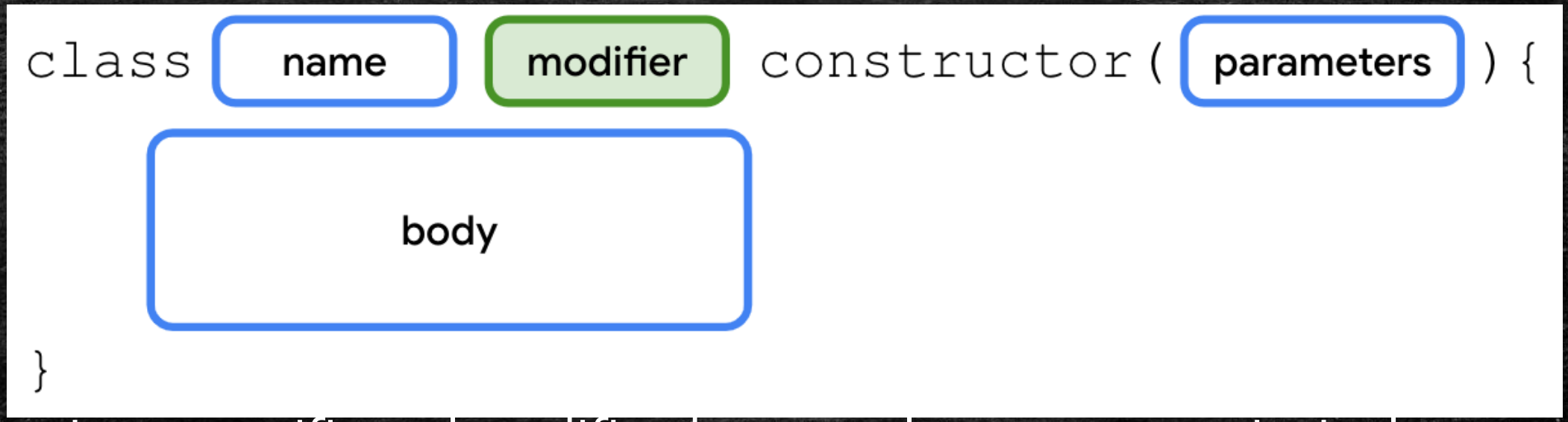
7.POO CON KOTLIN:ENCAPSULAMIENTO

- Modificadores en métodos



7.POO CON KOTLIN:ENCAPSULAMIENTO

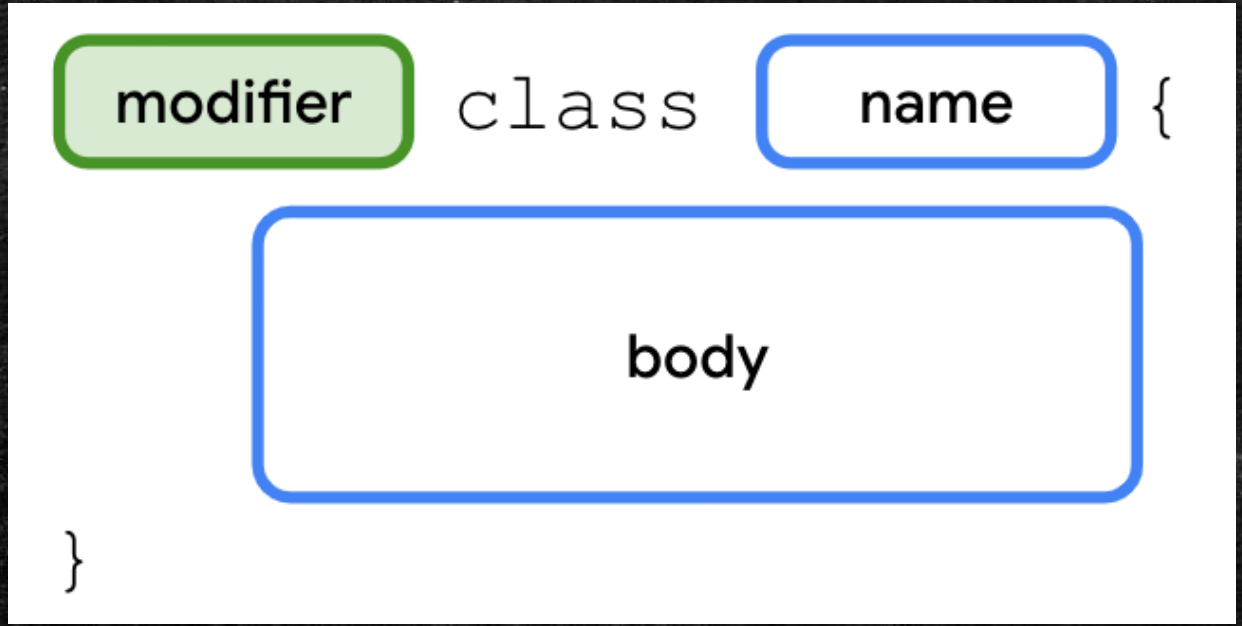
- Modificadores en constructores



  - Si necesitas especificar el modificador para el constructor principal, es necesario mantener la palabra clave constructor y los paréntesis incluso cuando no haya parámetros

7.POO CON KOTLIN:ENCAPSULAMIENTO

- Modificadores en clases



7.POO CON KOTLIN:ENCAPSULAMIENTO

- CONSEJOS
  - Lo ideal es buscar la privacidad total de los miembros de la clase(protected), si no inténtalo con protected, internal o public en ese orden.

Modificador	Accesible en la misma clase	Accesible en subclase	Accesible en el mismo módulo	Accesible fuera del módulo
`private`	✓	X	X	X
`protected`	✓	✓	X	X
`internal`	✓	✓	✓	X
`public`	✓	✓	✓	✓

7.POO CON KOTLIN:CLASES ABSTRACTAS

- DEFINICION.- Clases de las que no se instancian objetos

```
abstract class nombre_clase { //cuerpo clase}
```

  - No es necesario anotar la clase con open
  - Los atributos se pueden definir como abstractos (no tienen valor en la clase), con modificador **abstract**

```
abstract {var|val} nombre_atributo:Tipo
```

  - Los métodos abstractos no tienen implementación
  - Pueden existir métodos concretos (con implementación)
  - Los métodos sin implementación deben estar marcados con la palabra reservada **abstract** y no tendrán cuerpo (sin llaves)

7.POO CON KOTLIN:CLASES ABSTRACTAS

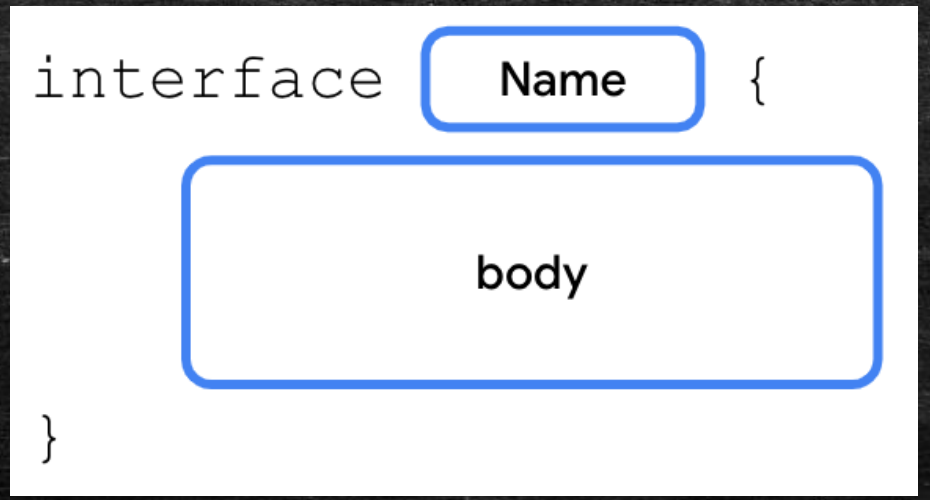
- HERENCIA DE UNA CLASE ABSTRACTA

```
class nombre_clase : clase_abstracta() { //cuerpo clase}
```

  - Deberá implementar todos los métodos y atributos abstractos o marcarlos como **abstract** y por lo tanto la clase heredada será abstract
  - Los atributos y funciones se sobrescriben con **override**

8.POO CON KOTLIN:INTERFACES

- DEFINICIÓN



  - Pueden definir funciones abstractas o no(con código)
  - Los métodos y propiedades no es necesario que se definan con **abstract**
  - No se pueden instanciar

7.POO CON KOTLIN:INTERFACES

- Una clase puede implementar de varias interfaces, sintaxis para implementar una interface

```
class nombre_clase [(parametros)]: interface_implementada,.. {//cuerpo clase}
```

- No se puede instanciar un objeto de una interface
- No pueden definirse constructores en una interface
- Los métodos pueden contener código
- Las propiedades no pueden tener valor (solo implementación de método get)
- Las propiedades y métodos no definidos en la interface deberán implementarse en la clase que define la interface

7.POO CON KOTLIN:EJERCICIOS

- Ejemplo a realizar en clase (EnunciadoEjemplo_POO.pdf)
- Ejercicio 1: Realizar el siguiente CODELAB
- Ejercicio 2: Realizar el siguiente CODELAB
- Ejercicio 3:

7.POO CON KOTLIN:ENUM

- Enum.- Clases type-safe (tipado seguro), clases que toman valores que se conocen de antemano

```
enum class nombre_clase {valores_en_mayusculas}
```

  - Los valores se escriben en MAYUSCULAS separados por comas
  - Los valores son del tipo enum class definido
  - Para declarar una "variable", de la misma forma que con el resto de tipos
    - El valor debe ser uno de los valores del enum class, ***nombre_clase.VALOR***

```
enum class Vcolor { RED,YELLOW,WHITE,BLACK}
Fun main(){ var color:Vcolor=Vcolor.RED}
```

    - Una "variable" enum tiene dos propiedades: name y ordinal, name es una representación String del valor y ordinal la posición del elemento desde CERO.

7.POO CON KOTLIN:ENUM

- Puede contener funciones.- Necesario poner un ; en el último valor para separarlo de las funciones

enum class *nombre_clase* {valores_en_mayusculas}

```
enum class Vcolor {
ROJO,AZUL,VERDE,NEGRO;
fun retornar_Color():Color {
        when(this){
                ROJO-> Color.valueOf(Color.RED)
                AZUL-> Color.valueOf(Color.BLUE)
                VERDE->Color.valueOf(Color.GREEN)
                NEGRO->Color.valueOf(Color.BLACK)
                }
} //Ejercicio: Pruebe a quitar alguna rama,que sucede?
```

7.POO CON KOTLIN:ENUM

- Puede contener propiedades, si se definen en el constructor principal será necesario inicializar cada valor del enum. Los objetos tendrán esas nuevas propiedades

enum class *nombre_clase*(*propiedad*) {valorenum(valor_prop)}

```
enum class Vcolor(var num_color:Int,var nombre_color:String) {
ROJO(1,"Red"),AZUL(2,"Azul"),VERDE(3,"Green"),NEGRO(4,"Black")
}
//Un objeto Vcolor tendrá 2 nuevas propiedades: num_color y nombre_color
```

Ejercicio: Defina una aplicación android que permita cambiar el color de fondo de la app, en función del radioButton seleccionado, se mostrará el color seleccionado en un TextView.

7.POO CON KOTLIN:NESTED e INNER CLASS

- Nested Class.- Clases anidadas o clases dentro de otras clases
  - Para definir un objeto de la clase interna: var objeto=ClaseExterna.ClaseInterna()

```
{var|val} objeto=ClaseExterna.ClaseAnidada()
```

  - Con el objeto se podrá acceder al contenido de la clase interna (métodos y/o propiedades)
- Inner Class.- Si desde la clase interna queremos acceder a propiedades/funciones de la clase externa deberemos definir la clase anidada como interna con la palabra reservada **inner**
  - Para definir un objeto de la clase interna será necesario instanciar la clase externa

```
{var|val} objeto=ClaseExterna().ClaseInterna()
```

  - Operador this en las clases internas (this@ClaseExterna) para acceder a los miembros clase Externa

Ver Ejemplo: Clase_Anidada_Interna.kt

7.POO CON KOTLIN:CLASES ANONIMAS

- En Kotlin se denominan Objects Expressions, son un tipo de clases internas
- Son instancias de clases que no tienen nombre. Sintaxis:

```
object [:clasepadre() o interface]{ //cuerpo de la clase, propiedades y funciones
}
```

- No admiten definición de constructores (ni principal ni secundario).
- Son útiles cuando es necesario implementar una interface o heredar de una clase pero no necesitamos la clase porque no se va a volver a usar. Se usan habitualmente como argumentos de algunos métodos que requieren un objeto de una interface.

Ejemplo: Proyecto Android EscuchadorTeclado

- Una función puede retornar una clase anónima

Ver Ejemplos: clases_anonimas.kt

7.POO CON KOTLIN:DATA CLASS

- Clases preparadas para almacenar datos

```
data class nombre_clase(parametros){//cuerpo de la clase}
```

  - Los parámetros se convertirán en las propiedades del data class
- Automáticamente se crean una serie métodos:
  - toString().- Representación en forma de String de las propiedades que se definan en el constructor principal
  - equals().- Para comparar 2 objetos de la clase (valores de sus propiedades del constructor principal), se puede sustituir por el operador ==
  - copy().- Permite copiar los valores de un objeto en otro, en el método se pueden pasar argumentos para diferenciar el valor en alguna propiedad

```
var objeto_copia=objeto.copy(edad=34)
```

7.POO CON KOTLIN:DATA CLASS

- Desestructuración.- Dividir un objeto en los valores de sus atributos

```
val (atributo1,atributo2)=objeto_dataclass
```

  - Se llaman funciones de tipo ComponentN, se podrá acceder a los datos de las propiedades en las variables atributo1, atributo2, etc..
  - Un data class define de forma implícita funciones ComponentN, donde N es un nº que va de 1 hasta el número de propiedades que tenga la clase
- Pueden incluirse funciones pero deben estar dedicadas al trabajo con los propios datos.

Ejemplo2: ClasePadreDataClass.kt

Ejemplo1: trabajador_dataclass.kt

7.POO CON KOTLIN:TYPE ALIAS

- Nombrar de una forma más legible a tipos existentes

```
typealias nombre_tipo=tipodato o tipofunción o clases anidadas
```

- Se suelen definir fuera de la definición de clases
- Se podrá usar el nuevo tipo definido para declarar las variables
- Type Alias para tipo de dato

```
typealias mapMutable=MutableMap<Int,ArrayList<String>>
...
var  mimapmutable:mapMutable=mutableMapOf()
```

- Para tipo función.

```
typealias myfun=(Int,String)->Boolean
```

- Para clases anidadas

```
typealias miclaseinterna=ClaseExterna.ClaseAnidada
.....
var  objeto_interno=miclaseinterna()
```

7.POO CON KOTLIN:DESTRUCTURACIÓN

- Dividir un objeto en sus componentes

```
val (variable1,variable2,...)= objeto o expresión_retorna_objeto
```

  - En variable1, variable2, etc.. Se van a almacenar los valores de las propiedades de ese objeto
- Se puede usar una invocación a una función que retorna un objeto
- Si el objeto no es dataclass será necesario implementar en la clase métodos de tipo componentN que retornan el valor del atributo
- Si no queremos desestructurar una determinada propiedad se usa el _ en la posición de ese atributo

```
val (c1,_,c3)=objeto
```

- Los mapas se pueden destructurar

```
for((indice,valor) in mapa)
```

7.POO CON KOTLIN:EXTENSION

- Documentación oficial.- Mecanismo para ampliar miembros de una clase sin tener que crear otra clase que herede de ella
- Extendiendo FUNCIONES.-

```
fun clasebase.nombrefuncionext([parametros]):tiporetorno{//cuerpo función}
```

  - Define una función nueva para la clasebase, se podrá invocar con un objeto de la clase base.
- Dentro del cuerpo de la función la referencia ***this*** se refiere al objeto de la clase base que invoca a la función.

Ejemplos: ProyectoPOO

EJERCICIOS

- Ejercicio1:REALIZAR EL SIGUIENTE CODELAB, pero en una aplicación móvil, utiliza las funciones de extensiones vistas en la diapositiva anterior.
- Ejercicio2: Realizar un programa Android que según se van introduciendo caracteres en un campo de texto se vaya mostrando en un TextField tantos _ como caracteres tiene la cadena que se va introduciendo, hacer uso de la función de extensión definida en la anterior diapositiva.
- Ejercicio 3:Tenemos que hacer una aplicación que permita gestionar un cuestionario. Para ello es necesario definir una clase que represente las preguntas. Entre sus miembros habrá que definir lo siguientes:
  - Propiedad que permita almacenar la pregunta (String)
  - Una propiedad que defina el tipo de dificultad de la pregunta (fácil, media, difícil)
  - Una propiedad que nos permita almacenar la respuesta, en este caso las preguntas pueden ser de 3 tipos:
    - Preguntas para completar con una cadena
    - Preguntas de tipo test con 4 posibles respuestas (a,b,c,d)
    - Preguntas de verdadero o falso
    - Preguntas con una respuesta numérica (matemática)
  - Una función que nos devuelva si la respuesta dada por el usuario (necesita como parámetro la respuesta introducida) es correcta o no.

8.TIPOS GENERICOS

- Documentación oficial.- Mecanismo para poder hacer flexibles clases,funciones. CODELAB
- EJEMPLO: Tenemos la siguiente clase (Genericos.kt)
- Sintaxis

```
class NombreClaseGenerica<tipos_genericos>[(propiedades)]{//Cuerpo clase
```

```
fun <tipos_genéricos> nombre_función(t:tipo_incluido generico):[Tipo_dato_incluido generico]{//Cuerpo}
```

  - *Tipos_genéricos.-* No existe ninguna norma definida para indicar el tipo genérico, pero por convección se usan letras en mayúsculas de la A a la Z (ultimas del alfabeto habitualmente T,R)
    - Se pueden definir varios tipos genéricos
    - Los tipos genéricos se pueden usar para definir propiedades de ese tipo en la clase

8.TIPOS GENERICOS

- Coovariancias (In, Out).- Articulo para entender el problema y la solución
  - Documentación Oficial
    - Si el tipo genérico es In, se usa para parámetros (elementos de entrada) (similar a ? Super T)
    - Si el tipo genérico es Out, se usa para devoluciones (elementos de salida), (similar a ? Extends T)
    - Solamente se permite para la definición de clases e interfaces
  - Ejemplo de Coovarianza
- EJEMPLO DE TIPOS GENERICOS CODELAB

9.COLECCIONES

- Documentación oficial.- Variables que almacenan un número variable de elementos de un tipo.
- Kotlin permite trabajar con 3 tipos:
  - **List** (Listas).- Colección ordenada de elementos con acceso mediante índices (nºs enteros). Pueden contener valores duplicados. Ej: Colección de preguntas
  - **Set** (Conjuntos).- Colección de elementos únicos (no se pueden repetir). Ej:Nºs generados en un juego de lotería
  - **Map** o dictionary(Mapas o diccionarios).- Conjunto de pares clave-valor. Las claves son únicas. Los valores se pueden repetir. Para almacenar objetos que tienen relación entre el valor y la clave. Ej: Login, contraseña de un usuario

Aconsejable declarar objetos constantes (val)

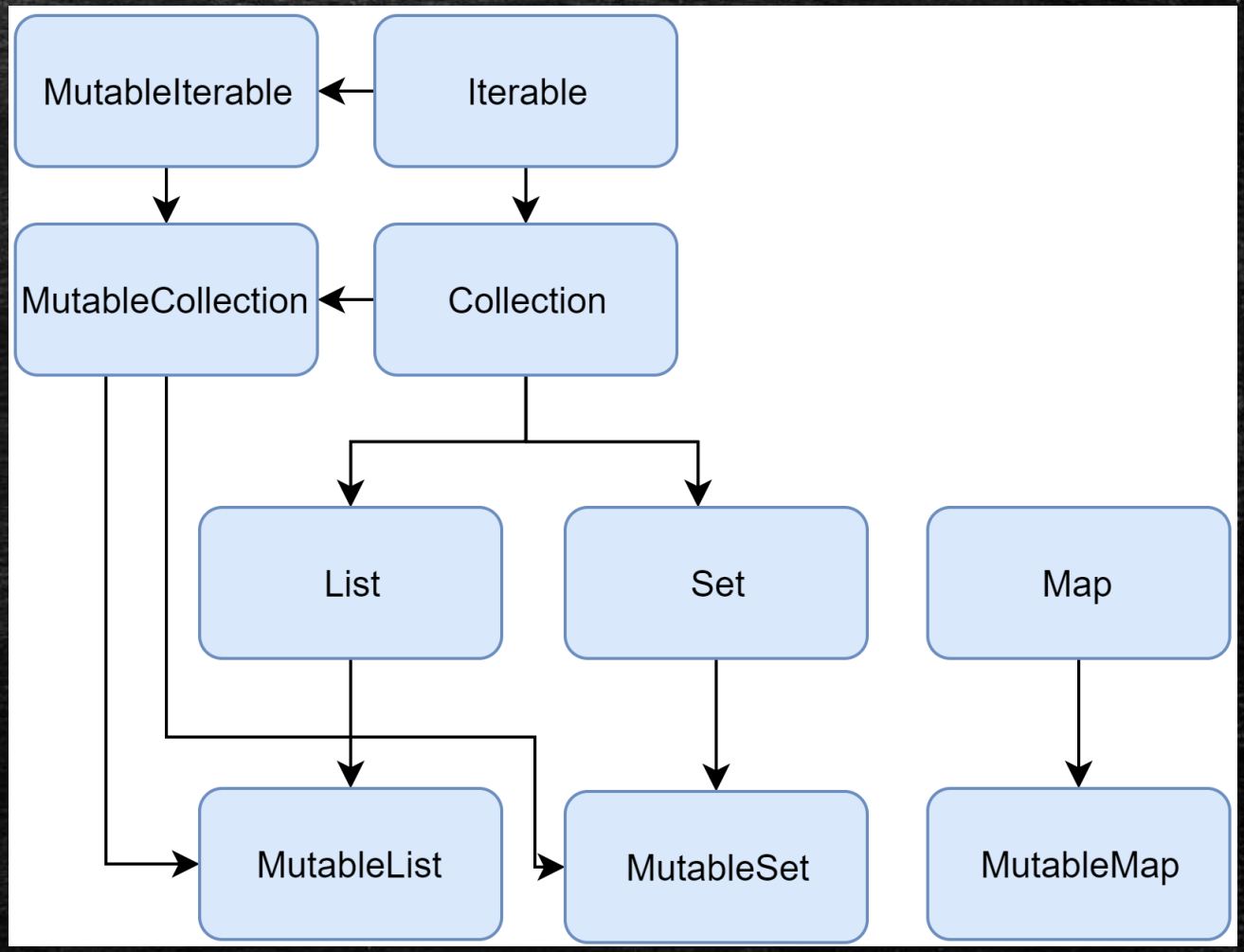
9.COLECCIONES

- A su vez cada colección puede ser de 2 tipos
  - Mutables.- Se pueden cambiar los elementos que tiene una vez definidos (add, remove)
  - No mutables.- No se pueden cambiar los elementos una vez inicializados
- Todas las colecciones comparten algunas características
  - Se pueden recorrer con función foreach((E) -> Unit)
  - Pueden hacer uso del operador común in (for (elemento **in** colección)), que realmente es una implementación de la función contains (Ver operadores)

Aconsejable declarar objetos constantes (val)

9.COLECCIONES

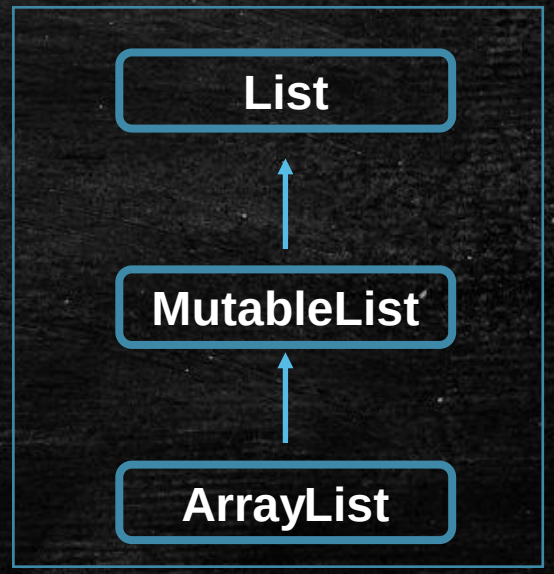
- JERARQUIA DE CLASES e INTERFACES



9.COLECCIONES:LISTAS

- Doc. Oficial
- Acceso a los elementos con sintaxis Array []
- Índices numéricos desde el 0 hasta nºelementos -1
- Métodos y propiedades comunes
  - IndexOf (Element E).- Posición de la 1ª ocurrencia encontrada del elemento o -1 si no se encuentra
  - size.- Propiedad que almacena el tamaño de la lista

Ejemplos: Listas.kt



9.COLECCIONES:LISTAS

- Inmutables (Interface List)
  - Instanciación.- LisfOf(*lista de elementos, …*) función dentro del paquete kotlin.collections (ver definición de función, ejemplo de función genérica), de que tipo es el objeto que instancia?
- Mutables (Interface MutableList)
  - Instanciación.- mutableListOf(*lista de elementos, …*) función en paquete kotlin.collections
  - Se incluyen funciones como add(e:E) o removeAt(pos:Int)
  - Clase concreta ArrayList<E> es la concreción por defecto de un MutableList

Ejemplos: Listas.kt

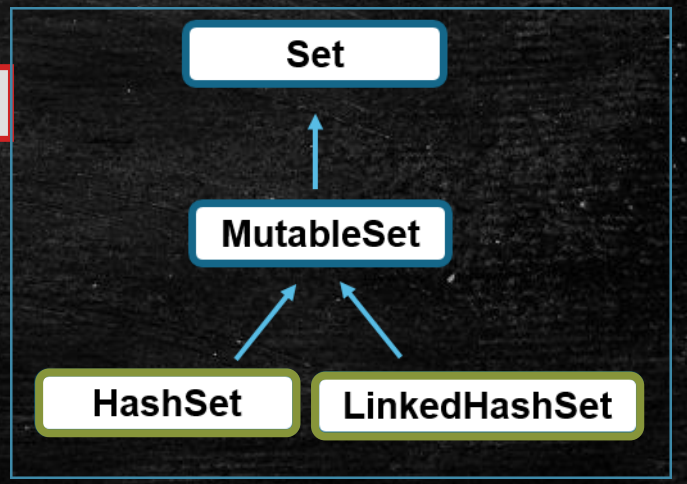
9.COLECCIONES: CONJUNTOS

- CONJUNTOS (Set).- Colección de elementos sin orden que no permite duplicados (incluido null). Ver codelab
  - Dos conjuntos son iguales si tiene el mismo tamaño y tienen los mismos elementos
  - Índices numéricos desde que se obtienen con una función hash()
  - Propiedades
    - Busqueda (indexOf)(directa) de un elemento más rápida que las Listas (en promedio similar), listas búsqueda secuencial
    - Usan más memoria que las listas (más índices de array)
  - Útiles cuando se quiere garantizar la singularidad
  - Instanciación.- SetOf(*lista de elementos,…*)

9.COLECCIONES: CONJUNTOS

- Conjuntos Mutables (Interface MutableSet)
  - Instancia con mutableSetOf([*lista_elementos,...*])
  - Se incluyen funciones como add(e:E), remove(e:E)
  - La implementación por defecto es un LinkedHashSet, que preserva el orden de inserción de los elementos, funciones con first() o last()
  - HashSet.- Otro tipo de cjto., en este caso no almacena información sobre el orden de los elementos, requiere menos memoria.

JERARQUIA DE CLASES SET



9.COLECCIONES:MAPAS

```
val mapaColores = mapOf(1 to "Rojo", 2 to "Verde", 3 to "Azul")
```

- Documentación Oficial
- Definición: Colección de pares clave-valor: Map<K,V>
  - Las claves son únicas, los valores se pueden repetir
- Técnicamente no es una Collection (no herede de..)
- Al igual que en las colecciones hay 2 tipos de mapa
  - Inmutables(Map).- Inicialización mapOf(*índice* to *valor*,...)
  - Mutables(MutableMap).- Inicialización mutableMapOf(*índice* to *valor*,...)
- La forma de acceso, sintaxis Array
- La instancia de un objeto Map es LinkedHashMap

Clave	Valor
1	Rojo
2	Verde
3	Azul

```
fun main() {
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap.put("three", 3)
    numbersMap["one"] = 11
    println(numbersMap)
}
```

9.COLECCIONES:MAPAS

- OPERACIONES
  - Acceso: mapa[clave]
  - Añadir un elemento: mapaMutable.put(clave, valor)
  - Eliminar elemento (solo en MutableMap): mapaMutable.remove(clave)
  - Recorrer un mapa

```
for ((clave, valor) in mapa) // Destructuración
{
  println("Clave: $clave, Valor: $valor")
 }
```

9.COLECCIONES:MAPAS

- FUNCIONES IMPORTANTES
  - getOrDefault: Devuelve el valor asociado o un valor por defecto.
  - containsKey y containsValue: Verifica si una clave o valor existe.
  - filter: Filtra elementos del mapa según una condición.

```
val mapaFiltrado = mapa.filter { it.key > 1 }
```

10.SCOPE FUNCTIONS

- FUNCIONES DE ÁMBITO
  - Funciones diseñadas para ejecutar un bloque de código en el contexto de un objeto
  - 5 funciones de ámbito en kotlin
    - Let, run, with, apply y also

10.SCOPE FUNCTIONS

- FUNCIONES DE ÁMBITO

Función	Referencia	Devuelve	Uso típico
let	it	resultado del lambda	Transformaciones, nullables
run	this	resultado del lambda	Inicialización y cálculo
with	this	resultado del lambda	Agrupar operaciones sobre objeto
apply	this	objeto original	Configuración inicial de objeto
also	it	objeto original	Side-effects: logging, validación

10.SCOPE FUNCTIONS

- EJEMPLO: Diferencias entre let y run. Tenemos un spinner para elegir entre usuario Normal y Administrador y unos EditText para recoger el nombre del usuario, la password y un campo extra que sirve para indicar el nivel(Usuario normal) o área(Usuario adminsitrador), se creará un objeto de la clase UsuarioNormal o UsuarioAdministrador en función de la selección del Spinner, pero solo si los campos no están vacíos.

10.SCOPE FUNCTIONS

- Ejemplo

let

```
val nombreInput = editNombre.text.toString().takeIf { it.isNotBlank() }
val edadInput = editEdad.text.toString().toIntOrNull()

nombreInput?.let { nombre ->
   edadInput?.let { edad ->
      val usuario: Usuario = when (spinnerTipo.selectedItem as String) {
         "Normal" -> {
            val nivel = editExtra.text.toString().toIntOrNull() ?: 1
            UsuarioNormal(nombre, edad, nivel)
         }
         "Administrador" -> {
            val area = editExtra.text.toString().ifBlank { "General" }
            Administrador(nombre, edad, area)
         }
         else -> Usuario(nombre, edad)
      }

      textViewDescripcion.text = usuario.obtenerDescripcion()
   }
}
```

run

```
val usuario = Usuario("Invitado", 0).run {
   nombre = editNombre.text.toString().ifBlank { "Desconocido" }
   edad = editEdad.text.toString().toIntOrNull() ?: 0
   obtenerDescripcion()   // Este será el valor devuelto por run
}

textViewDescripcion.text = usuario // usuario aquí es el String devuelto por run
```

11. FUNCIONES ÚTILES

- takeIf.- Devuelve el objeto que invoca si se cumple el predicado o null en otro caso
- isNotBlank.- Devuelve true si el String que invoca no es vacio
- ifBlank.- Devuelve la expresión de la lambda si la cadena invocante es vacia o la propia cadena si no lo es.
- map.- Devuelve una lista resultado de transformar cada uno de los elementos de cada elemento de la lista según la función de transformación
- filter.- Permite filtrar elementos de una colección, según la función predicado

EJERCICIOS

- Definir un pequeño programa en Kotlin que permita inicializar un examen (Lista de preguntas), permite resolverle y obtener la puntuación.
- Definir un programa en Android que permita generar los nº de la lotería primitiva. Que estructura utilizarías?
- Probar la función filter
- Realizar el siguiente CODELAB sobre COLECCIONES
- Ejercicio sobre Mapas