

Apuntes

[Descargar estos apuntes](#)

Tema 0 . Programación Básica en Kotlin

Índice

1. [Introducción](#)
2. [Control de Nulos](#)
3. [Variables](#)
4. [Control de flujo](#)
5. [Funciones](#)
6. [Clases e Interfaces](#)
7. [Clases Enum en Kotlin](#)
8. [Clases y métodos parametrizados](#)
9. [Colecciones](#)
 1. [Listas](#)
 2. [Mapas](#)

Introducción

Kotlin es un lenguaje de programación fuertemente tipado desarrollado por **JetBrains** en 2010 y que está influenciado por lenguajes como **Scala**, **Groovy** y **C#**.

La mayoría de desarrollos con Kotlin, tienen como destino **Android** o la máquina virtual de **Java** (**JVM**), y aunque también ofrece la posibilidad de desarrollos para la máquina virtual de **JavaScript** o código nativo, por ahora todavía son pocos los programadores que lo usan para estos destinos. A partir de la actualización **Kotlin 1.3.30**, se incluyeron las mejoras para **Kotlin/Native** que permite compilar el código fuente de Kotlin en datos binarios independientes para diferentes sistemas operativos y arquitecturas de CPU, incluido **IOS**, **Linux**, **Windows** y **Mac**.

En el siguiente [enlace](#) puedes encontrar un **compilador de Kotlin**, y otros lenguajes, online que permite entrada de datos por consola, lo puedes usar para probar los ejemplos.

Control de Nulos

Kotlin ha aportado una serie de elementos que permiten realizar un mayor control de los tipos que pueden ser nulos **Null safety**, y que pretende evitar la tan conocida excepción

NullPointerException.

Como ocurre con otros lenguajes, Kotlin permite definir un tipo no nulo como anulable. Esto lo hace mediante el elemento **?**, por ejemplo una variable de tipo entera no podría almacenar un valor nulo, pero se puede cambiar esta condición de la siguiente manera.

```
fun main()
{
    var numero:Int
    numero=null //ERROR de compilación, los tipos valor no son anulables
    var numeroAnulable:Int?
    numeroAnulable=null //OK
}
```

Hacer que una variable pueda ser nullable

Para comprobar si las variables son nulas, siempre se puede usar el condicional **if/else**, aunque **Kotlin** nos permite otras opciones. El operador **llamada segura** **?.** que solo realizará la llamada en caso que el valor sea distinto de nulo y devolverá **null** en otro caso.

```
fun main()
{
    var cadenaAnulable:String?=null
    println(cadenaAnulable?.length)
}
```

Llamada segura

En este ejemplo, si no usáramos el operador de llamada segura, el código lanzaría NPE. Pero ahora mostrará null.

Otro operador, no tan recomendado es el operador de **aserción no nula** **!!**. convierte cualquier valor en un tipo no nulo y lanza una excepción si el valor es null. Podemos decir de este operador, que vuelve las cosas a la normalidad ya conocida.

Por lo tanto, si deseas un NPE, puedes tenerla solicitándola con este operador.



```
fun main()
{
    var cadenaAnulable:String?=null
    println(cadenaAnulable.length) //ERROR de compilación, no permite posibilidad de prod
    println(cadenaAnulable!!.length) //Solicitamos esta opción con el operador !!
}
```

Aserción no nula
!! En caso de nulo, salta excepción

Y por último tenemos el operador **elvis** **?:** que comprueba que el valor no es nulo, permitiendo la llamada en ese caso o devolviendo lo que decidamos en caso que sea nulo.

```
fun main()
{
    var cadenaAnulable:String?=null
    println(cadenaAnulable?.length?:0)
}
```

Elvis
?? = ?:

Si la expresión a la izquierda de **?:** no es null, se realiza la llamada a **length**, de lo contrario devuelve la expresión de la derecha, en este caso **0**.

Variables

Igual que en todos los lenguajes de programación, en Kotlin también tendremos el recurso de las **variables** para almacenar valores. En Kotlin nos podemos encontrar con **variables inmutables** y **variables mutables**. En el caso de las primeras, una vez se le asigna un valor a la variable no podrá ser modificado, es decir, se comporta como una **constante** (lo mismo que utilizar **final** en Java o **const** en C#). Mientras que con la mutables podremos modificar en cualquier momento el valor de la variable.

Para declarar una variable como mutable, la tendremos que preceder de la palabra clave **var** mientras que para las inmutables usaremos **val**.

El concepto de **inmutabilidad** es muy interesante. Al no poder cambiar los objetos estos son más robustos y eficientes. Siempre que podamos hemos de usar objetos inmutables.

```
//Variables mutables
var mutable:Int=5;
mutable+=7;
var numeroDecimales=3.14F;
numeroDecimales=3.12F;

//Variables inmutables
val inmutable:Char='C';
inmutable='A' //Error compilación!! no se puede modificar una variable inmutable
```

V.Mutables
No necesario ";"
V.Inmutables

Como se puede ver en el ejemplo, cuando declaramos una variable, podemos indicar el tipo de esta o esperar a que el compilador lo infiera.

Para definir el tipo, tendremos que indicarlo con **:tipo** después del nombre:

(var|val) nombreVariable [:tipo][=valor] Declarar variables

Los tipos básicos de variables en Kotlin son:

Tipo	Valor	Tamaño
Byte	5.toByte()	8 bits
Short	5.toShort()	16 bits
Int	5	32 bits
Long	5L	64 bits
Float	1.45F	32 bits
Double	1.45	64 bits
Boolean	true	
Char	'H'	16 bits
Unit	Unit	0 bits

El tipo **Unit** corresponde a **void** en **Java** y **C#**

Cuando una **variable mutable** declarada en el **cuerpo** de una **clase**, no se quiere inicializar en el momento de la **declaración**, existe el concepto de **Inicialización tardía**, añadiendo el **modificador lateinit** delante de la **declaración** de la variable.


```
public class Ejemplo{  
    lateinit var cadena: String  
    fun miFuncion() {  
        cadena = "Hola Mundo";  
        println(cadena);  
    }  
}
```

Inicialización tardía

Si ponemos lateinit no hace falta poner ?=null

Evitaremos tener que poner ?.

Solamente para las propiedades

 Como en otros lenguajes de programación, se debe **controlar** las **operaciones** con **variables** de **distintos** **tipos**, para **evitar** **resultados** **inesperados** o incluso **excepciones**. **Kotlin** tiene varios **métodos toX()** para **cambiar** los **valores** al **tipo** que necesites:

```
fun main (args: Array <String>) {  
    var a:Int  
    var b=3.5f  
    a=b+2  
    println(a);  
}
```

Operar con diferentes tipos de dato

ERROR en este caso

Produciría el **error** *error: type mismatch: inferred type is Float but Int was expected.*

Mientras que: `a=b.toInt()+2`, evitaría el error, aunque la función redondearía a 3.

Similar a Parse

String

El tipo **String** representa el literal de cadena ya conocido, podemos ver un ejemplo del uso de templates `$` para la salida por pantalla.

```
fun main (args: Array <String>) {  
    val cadena = "El resultado de:"  
    val a = 2  
    val b = 5  
    println("$cadena $a + $b es: ${a + b}")  
}
```

Uso de templates en cadena

El resultado de: 2 + 5 es: 7

Control de flujo

El **control de flujo en kotlin** tiene algunas **diferencias interesantes** a **otros lenguajes** más conocidos. Las **instrucciones if/else, for y while** se pueden **considerar similares**, así que vamos a explicar solamente los **elementos** que las **diferencian**:

for y while

Se **diferencia** sobre todo en que se tienen que **usar rangos** en la **sentencia for** y se pueden **usar** en la **while**:

```
fun repeticionConFor(v:Int)
{
    for(i in (0..v).reversed()) println("$i")
}
fun main() { repeticionConFor(10)}
```

For

Usando el reversed iremos desde el valor "v = 10" dado hasta 0

```
fun repeticionConWhile(tope:Int)
{
    var contador:Int=0
    do{
        var numero=readLine()!!.toInt();
        contador++;
    }while(contador<tope && numero!in 1..100)
}
fun main() { repeticionConWhile(10)}
```

Do While

!! En caso de nulo, salta excepción

when

Sustituto de **switch**, y que también lo podemos **encontrar similar** en **C#**. Con un **ejemplo** quedaría explicado:

```
fun getEstacion(entrada:Int):String
{
    return when(entrada)
    {
        1-> "primavera"
        2-> "verano"
        3-> "otoño"
        4-> "invierno"
        else -> "Estación incorrecta"
    }
}
fun main() { println(getEstacion(2))}
```

switch = when

En caso de introducir más de una sentencia se introduce entre llaves

Otro **ejemplo** de la **gran funcionalidad** que nos permite la sentencia **when** podría ser el siguiente. Observa que en este caso el **when** es **sin argumentos**:

```
fun noHagoNada(x:Int, s:String, v:Float):String
{
    val res = when {
        x in 1..10 -> "entero positivo menor de 10"
        s.contains("cadena") -> "incluyo cadena"
        v is Comparable<*>-> "Soy Comparable"
        else -> ""
    }
    return res
}
fun main() { println(noHagoNada(2, "Hola", 3.5f));}
```

When sin argumentos

Funciones

Si queremos crear una **función** en kotlin tendremos que **precederla** de la **palabra reservada** **fun**. Por tanto, una función **constará** de la **palabra fun** seguida por el nombre de la **función** y entre **paréntesis** los **parámetros**, siempre y cuando los tenga. Si la **función** **retorna** un **valor** se definirá al **final** de la **signatura** (esto último se puede **omitir** siempre que la **función** **no devuelva nada**).

fun **nombreFuncion**([**nombreVariable:Tipo**, **nombreVariable:Tipo**, ...]):**TipoDevolucion**

```
fun sumaDatos(datoUno:Int, datoDos:Char )
{
    println("${datoUno + datoDos.toInt()}");
}
fun main() {
    var mutable:Int=5;
    mutable+=7;
    var numeroDecimales=3.14F;
    numeroDecimales=3.12F;
    val inmutable:Char='C';
    sumaDatos(mutable,inmutable);
}
```

Con **valor de retorno**:

```
fun mayor(numeroUno:Int, numeroDos:Int):Int
{
    return if(numeroUno>numeroDos) numeroUno else numeroDos
}
fun main() { println(mayor(5, 7));}
```

En lugar de la ternaria

Si la **función** solo tiene **una instrucción**, como en el caso anterior, **Kotlin** permite **omitir** las **llaves**.



```
fun mayor(numeroUno:Int, numeroDos:Int):Int= if(numeroUno>numeroDos) numeroUno else num
```

Función

Acción

Funciones de Extensión

Kotlin al igual que otros lenguajes, nos permite **extender** la **funcionalidad** de **clases** existentes, **agregando** **funciones** a estas. En el siguiente **ejemplo**, se ha **añadido** la **función** **Escribe()** a la **clase** **String**, por lo que se **podrá** **usar** con **objetos** de **este** **tipo**.

```
String.Escribe()=println(this);
main() {"Soy una cadena".Escribe()}
```

Función de extensión

Clases e Interfaces

Todos los elementos en Kotlin son públicos por defecto, por lo que también lo serán las clases. Las clases nos sirven para referenciar objetos del mundo real, y mediante las propiedades podemos definir las distintas características que nos interese manipular de estos. En Kotlin ya no existen los atributos, sino que todo pasa a ser propiedades, similares a las que utiliza C#, se les puede dar funcionalidad si lo necesitamos.

Una clase Persona con dos propiedades nombre y edad a las que queremos dar funcionalidad, podría quedar así:

```
class ExcepcionEdad(cadena:String):Exception(cadena){}
class Persona {
    val nombre:String
    get() = field.toUpperCase()
    var edad: Int=0
    set(value)
    {
        if(value <0 || value>125) throw ExcepcionEdad("Edad Invalida")
        else field = value
    }
}
```

Field pertenece a nombre, devolviéndolo en mayúsculas

Si te fijas, Kotlin utiliza la palabra clave `field` para referirse a la propiedad en sí, por lo que no tiene que crear variables secundarias como ocurre con C#.

Constructores

Las clases pueden tener un constructor principal y uno o más constructores secundarios. El constructor principal es parte del encabezado de la clase, esto nos permite construir un objeto sin tener que definir el constructor. Por ejemplo, para definir la clase Persona anterior pero sin funcionalidad en las propiedades, solo bastaría con hacer lo siguiente:

```
class Persona(val nombre: String, val edad: Int=0)
{
}
```

Clase

Y la creación del objeto sería:

```
val persona=Persona("Pepe", 23)
```

Objeto

Si necesitamos que ocurra algo en el momento de crear el objeto, se puede usar la clausula **init** que será ejecutada al llamar al constructor principal. Si te fijas en el siguiente código, la propiedad **edad** está declarada en el constructor, mientras que el **nombre** se ha declarado en el código de la clase.

Uso de init

```
class Persona(nombre:String, var edad:Int=0)
{
    val nombre:String
    get() {
        return field.toUpperCase()
    }
    init{ hace referencia al constructor principal
        this.nombre=nombre
        if(edad<0 || edad>125) throw ExcepcionEdad("Edad Invalida")
        else this.edad=edad
    }
}
```

Si lo que queremos es crear más de un constructor, tendremos que recurrir a la clausula **constructor** indicando los parámetros necesarios y llamando al constructor por defecto con **this** para enviarle sus propiedades. Ahora hemos añadido una propiedad más a la clase **Persona**, el **dni** que será asignado con el segundo constructor

Añadir propiedad

```
class Persona(nombre:String, var edad:Int=0)
{
    var dni:String="NINGUNO"
    val nombre:String
    get()=field.toUpperCase()
    init{
        this.nombre=nombre
        if(edad<0 || edad>125) throw ExcepcionEdad("Edad Invalida")
        else this.edad=edad
    }
    constructor(nombre:String, edad:Int=0, dni:String):this(nombre, edad)
    {
        this.dni=dni
    }
}
```

Herencia

En **Kotlin**, la clase **Any** es la raíz de la jerarquía de clases. Cada clase en el lenguaje derivará de ella si no especificas una superclase. Sería similar a la clase **Object** de **C#** y **Java**.

Por otro lado en **Kotlin**, tanto las clases como los miembros de estas son cerrados, esto significa que no se puede heredar de una clase y tampoco se pueden sobrescribir sus miembros si no lo

indicamos explícitamente. Para que de una clase se pueda heredar habrá que añadirle el modificador **open**. Por ejemplo, si la clase **Persona** queremos hacerla abierta sería:

```
open class Persona(nombre:String, var edad:Int=0)
{
    ...
}
```

Hacer clase **open** para que pueda ser heredada

Y ahora podríamos crear una clase hija de **Persona**, como por ejemplo:

Clase hija

```
class Estudiante(nombre:String, edad:Int=0, var estudios:String):Persona(nombre, edad)
{
}
```

Suponiendo que la propiedad **dni** queremos hacerla reescribible y que tiene un nuevo método también reescribible llamado **imprimir** y uno normal **esMayor**, ahora la clase quedaría:

```
open class Persona(nombre:String, var edad:Int=0)
{
    open var dni:String="NINGUNO" open = virtual
    ...
    open fun imprimir()= println("Nombre: $nombre Edad: $edad")
    fun esMayor() = if (edad >= 18) true else false
}
```

Hacer clase reescribible

Y con los elementos que queremos reescribir en la clase **Estudiante**:

Reescribir clase

```
class Estudiante(nombre:String, edad:Int=0, var estudios:String):Persona(nombre, edad)
{
    override var dni:String="ESTUDIANTE SIN DNI"
    override fun imprimir()
    {
        super.imprimir();
        println("Soy estudiante de $estudios")
    }
}
```

Interfaces

En **Kotlin** podemos implementar clases abstractas, que son iguales a las que ya conocemos de otros lenguajes, por lo que no vamos a comentar nada sobre ellas. Además también se pueden crear interfaces, que permiten definir tipos cuyos comportamientos pueden ser compartidos por varias clases que no están relacionadas. Usa la palabra reservada **interface** y su implementación

es similar a los lenguajes que conocemos con algunas pequeñas diferencias. Como ya sabemos permiten la herencia múltiple, y además:

- Pueden contener métodos abstractos (sin implementación) y métodos regulares (con implementación)
- Puede contener propiedades abstractas y regulares
- No permite declaración de constructores
- Las propiedades y métodos regulares de una interfaz pueden ser sobrescritos con el modificador override sin tener que marcarlos con open, a diferencia de en las clases abstractas.

```
interface IEstudios
{
    var curso: Int // Propiedad abstracta
    val ultimoCurso: Boolean // Propiedad regular
        get() = curso == 2
    fun estudios():String // Método abstracto
    fun soyEstudiante() { // Método regular
        println("Soy Estudiante de " + estudios())
    }
}
```

Interfaces

No es necesario poner open

Todo por defecto es open

Y ahora hacemos que la clase Estudiante además de heredar de Persona, implemente la interface IEstudios, quedando:

```
class Estudiante(nombre:String, edad:Int=0, >var estudios:String): Persona(nombre, edad)
{
    override var curso:Int=0
        set(value) {field=curso}
    override var dni:String="ESTUDIANTE SIN >DNI"
    override fun estudios()= estudios
}
```

Implementar interfaz en clase heredada

Clases Enum en Kotlin

No lo usaremos demasiado

Igual que en otros lenguajes, Kotlin nos permite crear tipos enumerados, aunque en este caso se puede ver como un modificador de clase. Una enumeración es un conjunto de valores que usan como identificador un nombre. Dicho nombre se comporta como una constante en nuestro lenguaje. Al marcar una clase con el modificador enum, la declara como una de enumeración.

```
enum class CiclosInformatica {SMR, ASIR, DAM, DAW}
fun nivelCiclo(ciclo:CiclosInformatica ):String
{
    return when(ciclo)
    {
        CiclosInformatica.SMR -> "Medio"
        else -> "Superior"
    }
}
fun main()
{
    var ciclo=CiclosInformatica.ASIR
    println(nivelCiclo(ciclo)) //Superior
}
```

Declarar una enumeración

Valor en las enumeraciones

Además a las enumeraciones en Kotlin también podemos asignarles uno o más valores. Esto se hará a través del constructor de la clase, lo podemos ver en el siguiente ejemplo, en el que al constructor se le ha añadido tanto un valor entero como un grado de tipo cadena.

Valores en las enumeraciones

```
enum class CiclosInformatica( val valor: Int, val grado: String)
{
    SMR(1,"Grado Medio"),
    ASIR(2,"Grado Superior"),
    DAM(3,"Grado Superior"),
    DAW(4,"Grado Superior")
}
fun main()
{
    10 for(v in CiclosInformatica.values()) println(v.name+" "+ v.ordinal)
    11 CiclosInformatica.values().forEach{println("${it.valor} - ${it.name} - ${it.grado}")}
}
```

SMR 0
ASIR 1
DAM 2
DAW 3

Valores asignados por defecto

1 - SMR - Grado Medio
2 - ASIR - Grado Superior
3 - DAM - Grado Superior
4 - DAW - Grado Superior

Propiedades creadas en el constructor

✎ La salida por pantalla mostraría primero los valores asignados a las propiedades por defecto **name** y **ordinal** **Línea 10** y después una lista con las propiedades que se han creado en el constructor **valor** y **estado** **Línea 11**.

Enumeraciones con comportamiento

También se les puede añadir un comportamiento a través de funciones abstractas o no, o incluso de interfaces. En el siguiente ejemplo podemos ver que al constructor se le ha añadido un elemento más con el nombre completo del ciclo, y además tenemos el comportamiento añadido mediante el método `informacionCompleta`. De forma que la ejecución del programa nos sacará las siglas del ciclo, el nombre completo y el grado que le corresponde a cada uno de los elementos de la enumeración.

Añadir comportamiento

```
enum class CiclosInformatica( val valor: Int, val grado: String, val nombre : String)
{
    SMR(1,"Grado Medio","Sistemas Microinformáticos y Redes"),
    ASIR(2,"Grado Superior","Administración de Sistemas Informáticos en Red"),
    DAM(3,"Grado Superior","Desarrollo de Aplicaciones Multiplataforma"),
    DAW(4,"Grado Superior","Desarrollo de Aplicaciones Web");

    fun informacionCompleta()= "${name} - ${nombre} - ${grado}"
}
fun main()
{
    CiclosInformatica.values().forEach{println("${it.informacionCompleta()}")}
}
```

Clases y métodos parametrizados

Kotlin también permite crear clases y métodos con alguno de sus miembros de tipo genérico. La lista de parámetros para tipos se incluyen en paréntesis angulares y se separan por coma si son varios `<T, U, V, ...>`

Clases Genéricas

Para crear una clase con un tipo parametrizado de forma que una de sus propiedades sea de ese tipo, se hará de la siguiente manera:

```
class ClaseGenerica<T>(t: T, c:String)
{
    var t:T
    val c:String
    init
    {
        this.t=t
        this.c=c
    }
    override fun toString()= "${t} ${c}"
    fun metodo(param: T) {t=param}
}
```

Crear clase genérica

Y si quisiéramos crear un objeto de esa clase con la propiedad parametrizada a tipo entero, se podría hacer `val objeto=ClaseGenerica(3, "Hola")`

Si quisiéramos realizar una restricción del tipo parametrizado a la interfaz Comparable, se tendría que hacer de la siguiente manera:

```
class ClaseGenerica<T: Comparable<T>>(t: T, c:String)
{
    ...
}
```

Añadir restricción de tipo

Funciones Genéricas

Para las funciones genéricas el parámetro de tipo se añadirá justo después de la palabra `fun` y las restricciones se harán de la misma manera que en las clases.

```
fun <T> funcionGenerica(param: T): T {
    return param
}
```

Colecciones

Listas

En Kotlin hay diferentes formas de generar **listas de objetos**: arrays, listas inmutables, listas mutables, etc. Cuidado, hay que distinguir la inmutabilidad de la colección de la inmutabilidad de la

variable que la contiene. En esta documentación solo comentaremos los array y las listas mutables, el resto de colecciones serán consultadas si es necesario su uso.

👉 Se recomienda aprovechar las características de este lenguaje y crear el tipo de colección apropiada para las necesidades de cada momento, así conseguiremos mayor eficiencia en las aplicaciones.

	arrayOf	listOf	arrayListOf	linkedListOf
Descripción	Array de objetos tradicional	Lista inmutable	Lista mutable	Lista enlazada mutable
Literales	arrayOf(1, 2, 3)	listOf(1,2, 3)	arrayListOf(1, 2, 3)	linkedListOf(1, 2, 3)
Inmutabilidad	no	sí	no	no
Modificar longitud	no	no	sí	sí

Usaremos menos

Usaremos esta

Arrays

Inmutable

Los arrays son mutable, es decir pueden cambiar el valor de sus elementos durante la ejecución, pero como ya conocemos por otros lenguajes, su tamaño no se puede cambiar una vez esté definido, existiendo otro tipo de colecciones para este uso.

👉 En Kotlin las declaraciones y acceso a las posiciones del array no suelen realizarse mediante corchetes, como en otros lenguajes, sino que se utilizan los métodos de la clase Array.

En el siguiente ejemplo podemos ver diferentes formas de recorrer el array mediante un bucle for.

```
fun main()
{
    val weekDays = arrayOf("primavera", "verano", "otoño", "invierno")
    for (dato in weekDays) println(dato) // Similar al foreach de C#
    for (i in weekDays.indices) println(weekDays.get(i)) // Acceso mediante índice
    for ((posicion, valor) in weekDays.withIndex()) // Volcando ambos datos en una tupla
        println("La posición $posicion contiene el valor $valor")
}
```

Además Kotlin distingue los arrays de primitivas usando clases propias:

BooleanArray, ByteArray, CharArray, ShortArray, IntArray, FloatArray, DoubleArray, y la forma

de construir literales es mediante: `booleanArrayOf`, `byteArrayOf`, `charArrayOf`, `shortArrayOf`, `intArrayOf`, `floatArrayOf`, `doubleArrayOf`.

En el siguiente ejemplo se ha creado un array de float de tamaño 4, el acceso está realizado mediante `[]` además la última línea produce excepción al intentar realizar un acceso no permitido.

```
fun main()
{
    val arrayFloat=FloatArray(4)
    arrayFloat[3]=2.5f;
    arrayFloat.set(0, 3f);
    print(arrayFloat[0])
    arrayFloat[4]=8.9f//Excepción por fuera de índice
}
```

Acceso a posiciones del Array

Si queremos crear un array de un tipo de objeto determinado, vamos a suponer un tipo `Persona` con el siguiente código:

```
class Persona(val nombre: String, val edad: Int)
{
    fun imprimir()= println("Nombre: $nombre Edad: $edad")
    fun esMayor() = if (edad >= 18) true else false
}
```

Clase Persona

El array de `Personas` lo podremos inicializar en el momento de creación del array, de la siguiente manera (en este caso solo tendrá un elemento):

```
fun main()
{
    val personas=arrayOf(Persona("Ana",12))
    personas.get(0).imprimir()
}
```

Array tipo Persona

Otra opción es permitir que el array pueda tener valores nulos, para añadir posteriormente los elementos. En este caso tendremos que utilizar el operador de llamadas seguras `?.`, que solo llamará al método en el caso que el valor no sea nulo, evitando `NullPointerException`.

```
fun main()
{
    val personas=arrayOfNulls<Persona>(2)
    personas.set(0,Persona("Ana",12))
    personas.get(0)?.imprimir()
}
```

Llamadas seguras

Array Bidimensional

En **kotlin** los **arrays** de **más** de una **dimensión** se tratan como **arrays de arrays** (similar a las **tablas dentadas** de **C#**). Por lo que se pueden **crear** **filas** de **distintos** **tamaños**. El siguiente código crea una **matriz** de enteros de **4 x 4** **inicializada** a **valor 0** y en el **elemento** **segunda fila** y **segunda columna** a **valor 3**.

```
fun main()
{
    val matriz = Array(4) { IntArray(4) }
    matriz[1][1]=3
    for (e in matriz) println(e.contentToString())
}
```

Array de arrays

Si **no** queremos **inicializar** cada una de las **filas** cuando **creamos** el **array**, se **puede** hacer **posteriormente** si se **crear** la **matriz anulable**. En este **ejemplo** creamos una **matriz** de **tres** **filas**, cada una de ellas con **tamaño** de **columnas** **distinto** (**2**, **3** y **4** respectivamente) e **inicializamos** toda la **matriz** a **valor 3**.

```
fun main()
{
    val dentada = arrayOfNulls<IntArray>(3) Tres filas
    dentada[0] = IntArray(2)
    dentada[1] = IntArray(3) Tamaño columnas
    dentada[2] = IntArray(4)

    for(i in 0..dentada.size-1)
        for(j in 0..dentada[i]!!.size-1) dentada[i]!!.set(j, 3) Inicializar valor a 3
}
```

Matriz columnas diferentes

Con la siguiente línea de código creamos una **matriz cuadrada** de **tamaño 4** con la **diagonal** a **1**.

```
val matriz = Array(4) { f -> IntArray(4) { c -> if(f==c)1 else 0 } }
```

Matriz 1 en diagonal

Listas Mutables

Las **listas mutables** están **representadas** por la **clase** **ArrayList** que **implementa** la **interface** **MutableList**. Seguramente serán las que **más** se **utilicen**, porque **permiten** un **comportamiento** más **flexible**. Pero como se ha comentado antes, para **mejorar** la **eficiencia** se debe **utilizar** la **colección** más **apropiada** para cada **necesidad**.

```
fun main()
{
    val personas=ArrayList<Persona>()
    personas.add(Persona("Ana",12))
    personas.add(Persona("Pedro", 15))
    for(p in personas) p.imprimir()
}
```

Para acceder a un elemento se usa la función **get(posicion)** y para reemplazar **set(posicion, elemento)**, como en los arrays. Se dispone de una función que añade una colección a la lista mutable **addAll(colección)**, para eliminar un elemento de la lista se usará **remove(elemento)** o **removeAt(posicion)**. Para buscar tenemos **indexOf(elemento)** o **lastIndexOf(elemento)**, la función **clear()** para limpiar toda la lista. Además dispone de una función **iterator()** o el elemento **forEach**. Vamos a ver un ejemplo de esto último

```
fun main()
{
    val personas=ArrayList<Persona>()
    personas.add(Persona("Ana",12))
    personas.add(Persona("Pedro", 15))
    personas.add(Persona("Rosa",13))
    personas.add(Persona("Andrea",15))
    val itr = personas.iterator ()
    while (itr.hasNext ()) itr.next().imprimir()
    personas.forEach{it.imprimir()}
}
```

Además para las colecciones podemos usar un filtro **filter** que nos permitirá realizar una selección de algunos elementos de la colección que cumplan la condición del filtro y después con **forEach** se recorre el resultado. Para el anterior ejemplo, podríamos filtrar por la edad de 15 años de la siguiente manera:

```
personas.filter{it.edad==15}.forEach{it.imprimir()}
```

Mapas

Diccionarios

También tendremos mapas `MutableMap` y mapas inmutables `Map` en `kotlin`. El funcionamiento es similar, por lo que vamos a ver los ejemplos de los mapas mutables.

Mapas mutables

```
fun main()
{
    val personas = mutableMapOf<String, Persona>()

    personas["21456874L"] = Persona("Ana",12)
    personas["13232345K"] = Persona("Luis", 13)
    personas.forEach{print(it.key + " ")
                    it.value.imprimir()}

}
```

En el siguiente ejemplo se introducen una serie de elementos más. La clase `Pair` que nos permite crear pares de valores y que ayuda a añadir elementos a una mapa, entre otras cosas. Además se puede ver la utilización del operador `+=` como otra manera de agregar elementos. Y otro ejemplo de filtro con conteo usando los elementos `filter` y `count`.

Introducir elementos en mapa mutable

```
fun main()
{
    val personas = mutableMapOf<String, Persona>(Pair("21456874L", Persona("Ana",12)),
                                                Pair("13232345K", Persona("Luis", 13)))

    personas += Pair("24568947L", Persona("Pedro",15))
    personas += Pair("25412321L", Persona("Laura",12))
    // Filtramos por nombres que comienzan por L y contamos
    println(personas.filter{ it.value.nombre.startsWith('L') }.size)
    // Lo mismo que antes pero usando count
    println(personas.count { it.value.nombre.startsWith('L') })
}
```

También se puede recorrer un mapa usando tuplas y `for` de la siguiente manera:

Recorrer mapa

```
for((dni,valor) in personas)
{
    print(dni)
    valor.imprimir()
}
```

Hay varios métodos de utilidad para trabajar con mapas, algunos son iguales a los que ya hemos visto en las anteriores colecciones y otros son propios, para más información consultar en la página oficial.

