# ESTRUCTURAS DE DATOS

La **estructura de datos en Kotlin** es el sistema mediante el cual se organizan un conjunto de datos bajo un mismo nombre, para que se puedan usar de manera eficiente.

Existen estructuras de datos **estáticas** y **dinámicas**:

* Una estructura de datos **ESTÁTICA** es aquella que el tamaño ocupado en la memoria se define antes de que el programa se ejecute y no puede ser modificado durante la ejecución del programa.

Son estructuras estáticas: Arrays y Matrices.

* Las estructuras de datos **DINÁMICAS** son aquellas en las que el tamaño podrá modificarse durante la ejecución del programa; teóricamente no hay límites a su tamaño, salvo el que impone la memoria disponible en la computadora.

Son estructuras dinámicas: Listas, Pilas, Colas, Árboles, Grafos

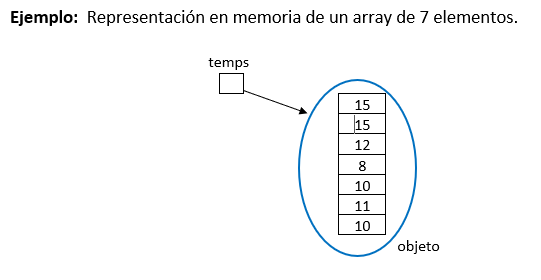
# LOS ARRAYS

## **Arrays unidimensionales**

Un array es una estructura de datos que permite almacenar un número indeterminado de variables del mismo tipo, que son referenciadas por el mismo identificador.

Características de un array:

* Se puede acceder a cada elemento mediante su **posición**
* La posición es un **número entero** que normalmente se le llama **índice**.
* Es importante tener en cuenta que el **primer elemento** de un array, tiene como índice el número cero.
* Dentro del array, cada valor ocupa una posición y se puede recuperar o modificar su valor combinando el identificador y dicha posición.
* Un array es una referencia a la estructura donde se almacenan sus valores (o sus referencias, si es un array de objetos), por lo que, si se pasa como parámetro a un método, se pasará como **referencia**.



Como se puede observar en la siguiente figura, la primera posición del array es la posición 0. Como consecuencia, la última posición del array será igual a la longitud del mismo menos uno.



### Declaración y creación del array

**Declarar** un array es declarar una **referencia** al array que posteriormente se creará. El array se declara igual que una variable, indicando que es una variable o una constante y el tipo de dato a guardar.

**val** **nombre**: **Array**<tipo>

Para crear un array en Kotlin y reservar espacio en memoria, se utiliza la función arrayOf() o la función Array().

**val** miArray: **Array**<Int> = **arrayOf**(1, 2, 3, 4, 5)

Por ejemplo, para crear un array de enteros con 5 elementos.

**val** otroArray: **Array**<Int> = **Array**(5) { 0 }

Crea un array de 5 elementos inicializados en 0.

Recuerda que en Kotlin los arrays son objetos, pero el tipo de los elementos debe ser especificado entre los signos de menor < y mayor >.

Diferentes ejemplos de ceación de Arrays:

Crear un array de cadenas (Strings) con elementos predefinidos:

|  |
| --- |
| val nombres: Array<String> = arrayOf("Juan", "María", "Carlos") |

Crear un array de números enteros con valores por defecto:

|  |
| --- |
| val numeros: Array<Int> = Array(5) { 0,2,4,6,8} |

Crear un array de booleanos con valores predeterminados:

|  |
| --- |
| var booleanos: Array<Boolean> = Array(3) { false } |

Crear un array de números flotantes utilizando la función floatArrayOf():

|  |
| --- |
| val flotantes: FloatArray = floatArrayOf(1.5f, 2.3f, 5.6f) |

Crear un array de caracteres (chars) utilizando la función charArrayOf():

|  |
| --- |
| val caracteres: CharArray = charArrayOf('a', 'b', 'c', 'd') |

Crear un array vacío y luego asignar valores:

|  |
| --- |
| val miArray: Array<Int> = arrayOf() // Crea un array vacío  miArray[0] = 10 // Asigna el valor 10 a la primera posición del array |

Crear un array de tamaño fijo con elementos inicializados en un valor específico:

|  |
| --- |
| val arrayInicializado: Array<Int> = Array(7) { 100 } // Crea un array de 7 |

Para asignar valor a cada uno de los espacios que se han reservado se puede acceder al array e indicar la posición para la que se quiere asignar

telefonos[0] = "976123456";

Para recorrer todos los elementos de un array se suele utilizar un bucle **for** junto con un índice que va desde 0 hasta el tamaño del array menos 1.

También es posible inicializar los valores de un array aprovechando esta estructura de bucle **for**, de forma que se puede asignar un valor a cada elemento del mismo, posición por posición.

El único requisito para recorrer un array es conocer el número de elementos, para ello contamos con un método que poseen los arrays que se llama **length**.

|  |
| --- |
| val telefonos = arrayOfNulls<String>(5)  // Inicialización de cada elemento del array con "..."  for (i in telefonos.indices) {  telefonos[i] = " . . . "  } |

|  |
| --- |
| val telefonos = arrayOfNulls<String>(5)  // Inicialización de cada elemento del array con "..."  for (i in telefonos.indices) {  telefonos[i] = " . . . "  } |

|  |
| --- |
| // Declaración del array de Strings con tamaño 5  val telefonos = arrayOfNulls<String>(5)  // Inicialización de cada elemento del array con "..."  var i = 0  while (i < telefonos.size) {  telefonos[i] = " . . . "  i++  } |

También es posible crear un array asignando directamente los valores que queremos que contenga el mismo. En ese caso se declara un array tal y como se muestra a continuación y después se recorre:

|  |
| --- |
| val telefonos = arrayOf("976123456", "91123456", "986789876")    // Para imprimir los números de teléfono  for (telefono in telefonos) {  println(telefono)  } |

Hay que tener en cuenta que se pueden crear arrays de tipos primitivos, de clases de kotlin, y también de cualquier clase de las que hayas programado en el proyecto.

val misMotos = arrayOfNulls<Moto>(10)

**Ejemplo** de creación de array de la clase Kotlin: String:

// Declaro y creo array de 4 objetos String (cadenas)

var ciudades = **arrayOfNulls<String>**;

ciudades[0] = "París";

ciudades[1] = "";

ciudades[2] = "Madrid";

ciudades[3] = ciudades[0];

// Escribe el contenido del array

**for** (i in 0 until ciudades.size) {

printf(“${ciudades[i]}”);

}

**for** (ciudad in ciudades) {

println("$ciudad ")

}

**for** (ciudad in ciudades) {

println("$ciudad ")

}

// Escribirá: **París Madrid París de tres formas distintas**

### Utilización del array

La utilización del array es muy sencilla. Cada elemento del mismo, una vez indicada la posición, se puede tratar como una variable del mismo tipo que el array, por lo que las operaciones que se pueden hacer con cada elemento son las mismas que se pueden hacer sobre variables individuales.

**Ejemplo**: Si tenemos un array de **int :**

var temps: Array<Int> = Array(5) { 0,2,4,6,8}

Cada elemento del array se puede utilizar exactamente igual que cualquier otra variable, es decir, se le puede asignar un valor o se puede usar dentro de una expresión.

En nuestro ejemplo, cada elemento del array es un entero, temps[0] es un entero y se podrá utilizar su valor de la misma manera que si fuera una variable entera:

**var** suma = **temps[0]** + **temps[1]**

**Ejemplo**: Tenemos un array de una clase Moto que hemos programado en el proyecto. Podemos invocar al método de cada elemento del array, de esa clase Moto:

var misMotos:Array<Motos> = Array(5){m1,m2,m3,m4,m5}

*// Poblar el array . . .*

for (i in 0 until misMotos.Size) {

misMotos[i].pasarRevision()

}

También se puede declarar una variable para tomar la referencia del elemento del array. Esto a veces es más cómodo para acceder a los métodos del objeto.

var misMotos:Array<Motos> = Array(5){m1,m2,m3,m4,m5}

*// Poblar el array . . .*

. . .

for (i in 0 until misMotos.Size) {

var moto:Motos = misMotos[i];

if (!moto.estaRevisada()) {

moto.pasarRevision();

}

}

Los arrays se pueden pasar como parámetro a un método, y como se ha dicho, un array es una referencia a la estructura donde se almacenan sus valores (o sus referencias, si es un array de objetos), por lo que, si se pasa como parámetro a un método, se pasará como referencia, por lo que es posible modificar los datos del array desde el método que se pasa como parámetro. La excepción está en los objetos de la clase String, que como sabemos, estos objetos no pueden modificarse.

fun main() {

val numeros = IntArray(5)

for (i in numeros.indices) {

numeros[i] = 10

}

incrementar(numeros)

println(numeros[3]) // Imprimirá 11

}

fun incrementar(valores: IntArray) {

for (i in valores.indices) {

valores[i]++

}

}

También debemos de tener en cuenta, que **los arrays** **NO pueden crecer,** por lo que, si necesitamos añadir un nuevo elemento a un array, tendremos que crear un nuevo array.

Vemos a continuación, un ejemplo de cómo podríamos hacerlo. Tenemos un método que toma un **array por parámetro** y **un nuevo elemento**. Veremos cómo, crea y **devuelve** un nuevo array

/\*\*

\* Crea un nuevo array con un elemento más y guarda en él,

\* el valor pasado por parámetro.

\*

\* **@param** lista array inicial

\* **@param** valor a añadir

\* **@return** nuevo array con el contenido del inicial

\* más uno con el valor.

\*/

private fun agregarElementoArray(lista: IntArray, valor: Int): IntArray {

// Creo un nuevo array con 1 elemento más

val nuevo = IntArray(lista.size + 1)

// Copio todos los elementos al nuevo array

for (i in lista.indices) {

nuevo[i] = lista[i]

}

// Agrego el nuevo valor en la última posición

nuevo[nuevo.size - 1] = valor

return nuevo

}

Una llamada válida al método anterior:

**val lista = intArrayOf(3, 4, 5)**

**// Añade un elemento más a la lista con un 10**

**lista = agregarElementoArray(lista, 10)**

Al guardar la referencia devuelta al nuevo array, en la **misma referencia** que se usaba para el array original, el array original deja de ser referenciado y será eliminado de memoria cuando pase el Garbage Collector (**gc**).

### Copia de un array

Si necesitamos copiar un array, se hace igual que con dos variables del mismo tipo:

**Ejemplo**: Tenemos los siguientes arrays:

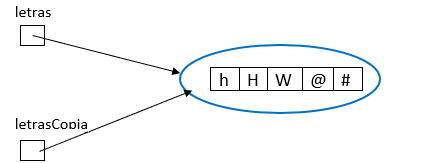
val letras = charArrayOf('h', 'H', 'W', '@', '#')

var letrasCopia: CharArray

Para copiar este array en uno nuevo llamado letrasCopia:

letrasCopia = letras // Asigna el array letras a letrasCopia

El resultado de la instrucción anterior es que tanto letras como letrasCopia, referencian al mismo array (**se copió la referencia al array, no se hizo una copia de los datos**).



Esta situación hace que cualquier cambio que se haga en letrasCopia, cambie también en letras, puesto que **solo hay uno**.

La forma correcta de copiar un array de tipos primitivos, es la siguiente:

//val letrasCopia = letras.copyOf() // Copia el array letras??Probar

// Creo un nuevo array con el mismo tamaño

val letras = charArrayOf('h', 'H', 'W', '@', '#')

var letrasCopia = CharArray(letras.size)

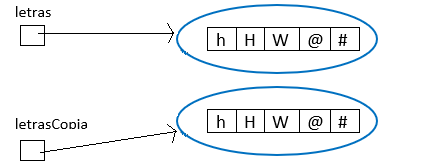
// Creo un nuevo array con el mismo tamaño

// Recorro el array copiando cada elemento

for (i in letras.indices) {

letrasCopia[i] = letras[i]

}



**NOTA:** Si se quisiera copiar un array de objetos, se deberían “clonar” esos objetos. Esto es algo que se verá más adelante.

### Comparación de arrays

Un array es igual que otro, si todos sus elementos son iguales. Cuidado si se utiliza el operador **==** para comparar arrays, lo que se **comparan son las referencias** y **NO el contenido** de los arrays.

**Ejemplo**

**val letras = charArrayOf('h', 'H', 'W', '@', '#')**

**val letras2 = charArrayOf('h', 'H', 'W', '@', '#')**

**val letras3: CharArray**

letras3 = letras2;

**if** (letras == letras2){ // NO se cumple

...

}

**if** (letras3 == letras2){ // SI se cumple

...

}

Para comparar dos arrays hay que hacer un recorrido de ambos arrays, e ir comparando dato a dato, aunque si el tamaño no coincide, no es necesario realizar el recorrido puesto que ya NO son iguales, kotlin también tiene el método letras.**contentEquals**(letras2).

letras.contentEquals(letras2) //compara el contenido

val letras = charArrayOf('h', 'H', 'W', '@', '#')

val letras2 = charArrayOf('h', 'H', 'W', '@', '#')

var iguales = false

if (letras.size == letras2.size) {

iguales = true

}

var i = 0

while (i < letras.size && iguales) {

if (letras2[i] != letras[i]) {

iguales = false

}

i++

}

if (iguales) {

println("Los arreglos letras y letras2 tienen el mismo contenido.")

} else {

println("Los arreglos letras y letras2 tienen contenido diferente.")

}

### Reasignación de arrays

Un array en Kotlin puede ser inicializado en varias ocasiones, al igual que ocurre con todas las variables en Kotlin:

**val notas = IntArray(5)**

**...**

**val notas = IntArray(8)**

Primero, notas tiene 5 elementos y después tiene 8. Pero, ¿QUÉ implicaciones tiene el código anterior?

Es fácil ver que el segundo **crea un nuevo array**, o lo que es lo mismo, un nuevo objeto y su referencia se guarda en notas.

Por tanto, el anterior objeto queda **sin referencia**, entonces ya no puede utilizarse, pero sigue ocupando espacio.

En Kotlin un **recolector de basura** se encarga cada cierto tiempo de liberar la memoria que ha quedado sin referencia, librando al programador de esta tarea.

Desde el código podemos forzar la recolección de basura, invocando al método **gc** *(garbage collector)* de la clase **System**:

**System.*gc*();**

Esta instrucción no hace que el recolector de basura se ejecute inmediatamente, pero podrá ser útil cuando nuestro código genere basura más rápidamente de lo que el recolector puede eliminar.

## **Clase Arrays**

En Kotlin, se puede trabajar con algunas funciones y extensiones útiles para operaciones con arrays dentro del paquete kotlin.collections. Aunque no hay una clase específica como java.util.Arrays en Java, muchas de las operaciones básicas que se realizan con arrays pueden llevarse a cabo utilizando funciones de extensión o funciones de alto nivel en Kotlin.

Algunas funciones útiles para trabajar con arrays en Kotlin se encuentran en el paquete kotlin.collections y son extensiones de las clases de arrays estándar. Algunas de estas funciones son:

**forEach**: Permite realizar operaciones en cada elemento del array.

**filter**: Filtra los elementos del array basándose en un predicado.

**map**: Transforma cada elemento del array en otro valor según una función proporcionada.

**reduce / fold**: Realiza una operación acumulativa en los elementos del array.

**count**: Cuenta los elementos que cumplen cierta condición.

**sorted**: Ordena los elementos del array.

Por ejemplo:

**val numbers = arrayOf(3, 1, 4, 1, 5, 9, 2, 6)**

**val sum = numbers.sum() // Suma todos los elementos del array**

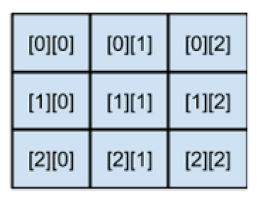
**val sortedArray = numbers.sortedArray() //copia ordenada del array**

**numbers.forEach { println(it) } // Imprime cada elemento del array**

## **Arrays bidimensionales o MATRICES**

Un array bidimensional utiliza dos índices para localizar cada elemento. Podemos ver este tipo de dato como un array que, a su vez, contiene otros arrays. También se puede ver como una cuadrícula en la que los datos quedan distribuidos en filas y columnas, de ahí que los arrays bidimensionales, también sean llamados **matrices**.

Al poderse interpretar como un array de dos dimensiones, **la primera dimensión hará referencia a las filas, y la segunda a las columnas.**



Se declaran e inicializan de la misma manera que los arrays unidimensionales, pero esta vez hay que tratar con dos dimensiones.

Podemos crear e inicializar en una única línea;

String[][] telefonos = new String[5][6]; //5 filas por 6 columnas

Para introducir u obtener cualquier elemento en la estructura, hay que indicar la posición de la fila y de la columna**:**

**// Acceder al elemento en la fila 2 y columna 3**

**val valor = telefonos[1][2]**

**// Modificar un elemento en la fila 4 y columna 5**

**telefonos[3][4] = "555-1234"**

Es posible inicializar los valores de una matriz aprovechando una estructura de bucle for, de forma que se puede asignar a cada elemento del mismo, posición por posición. En este caso será necesario anidar dos bucles for para acceder a las dos dimensiones del array.

val filas = telefonos.size //Devuelve el número de filas

val columnasFila= telefonos[i].size //Devuelve número de columnas de la fila i

val telefonos = Array(5){ Array(6) { "" } }

for (i in telefonos.indices) {

// Recorrer filas

for (j in telefonos[i].indices) {

// Recorrer columnas

telefonos[i][j] = " . . . "

}

}

También es posible crear un array bidimensional asignando directamente los valores que queremos que contenga el mismo. En ese caso se declara un array con la capacidad justa para almacenar todos los valores especificados.

val telefonos = arrayOf(

arrayOf("976123456", "91123456", "986789876"),

arrayOf("926213040", "926547814", "926503021"),

// ... otras filas

)

Y, por supuesto, también podemos crear arrays bidimensionales de objetos:

val misMotos = Array(10) { Array<Moto?>(2) { null } }

Los arrays bidimensionales se utilizan con frecuencia para situar objetos en un plano

como por ejemplo las piezas de ajedrez en un tablero, o un personaje de videojuego en un laberinto.