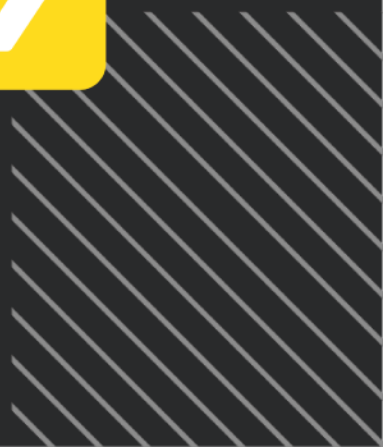




# Swift





## Tema 3. Tipos de datos II

### Objetivos

Luego de navegar esta cápsula, deberás poder hacer lo siguiente:

- comprender qué son las tuplas como tipo de dato;
- crear, acceder y modificar tuplas;
- definir rangos abiertos y cerrados;
- comprender qué es un Array y sus formas de instanciación;
- agregar, eliminar y acceder a elementos de un Array;
- comprender qué es un Set y sus formas de instanciación;
- agregar, eliminar y acceder a elementos de un Set;
- comprender qué es un Dictionary y sus formas de instanciación; y
- agregar, eliminar y acceder a elementos de un Dictionary.

### Tuplas

Las tuplas permiten agrupar combinaciones de distintos tipos de datos en una misma estructura, ya sean dos, tres o más.

```
let respuesta = (true, "Respuesta satisfactoria", 200)
```

```
let resultado = (false, "Archivo no encontrado")
```

En el ejemplo previo, en respuesta, se están agrupando tres valores: Booleano, String e Int, por lo que el tipo implícito será (Bool, String, Int). Por otro lado, en resultado, se están agrupando dos valores: un Booleano y un String (Bool, String).

Las tuplas son particularmente útiles como valores de retorno de una función:

```
func buscarArchivo(nombre: String) -> (Bool, String) {  
  var encontrado = false
```

```
  var mensaje = ""
```

```
  // realizar la búsqueda
```

```
// encontrado: true si hay éxito

// mensaje: Mensaje de error

return (encontrado, mensaje)

}
```

Veremos la definición de “funciones” más adelante en la cápsula «Funciones».

Existen diferentes formas de acceder a los valores de las tuplas:

### Forma directa

Para obtener el primer elemento de la tupla, se usa `.0`, para obtener el segundo, `.1`, y así sucesivamente.

```
let resultado = (false, "Archivo no encontrado")

resultado.0 // false

resultado.1 // "Archivo no encontrado"
```

### Por coincidencia de patrones (*pattern matching*)

Esta forma de acceder a los valores de las tuplas es útil, ya que automáticamente quedan separados los valores en identificadores distintos:

```
let (encontrado, mensaje) = (false, "Archivo no encontrado")

encontrado // false

mensaje // "Archivo no encontrado"
```

### Tuplas con nombres

Es posible definir un nombre para cada uno de los valores de la tupla. Esta forma aporta claridad sobre el significado de cada valor:

```
let resultado = (exito: false, mensaje: "Archivo no encontrado")
```

Además, permite separar los valores más fácilmente:

```
resultado.exito // false
```

```
resultado.mensaje // "Archivo no encontrado"
```

## Ejercitación 1 en el Playground de Xcode

1. Representa la fecha de hoy (día, mes, año) en una tupla de enteros.
2. Actualiza la tupla del ejercicio anterior agregando nombres a los elementos de forma que pueda identificarse mejor cuál es el día, mes y año respectivamente.
3. Utilizando la tupla anterior, muestra en consola un mensaje con la fecha completa, por ejemplo, "Hoy es 10 del mes 6 del año 2021".

## Rangos

Un rango corresponde a un tipo de dato que contiene un intervalo de valores. Por ejemplo, la expresión `0...9` crea el rango de valores: 0,1,2,3,4,5,6,7,8,9.

## Tipos de rangos

### Rango cerrado

Este tipo de rango incluye a ambos extremos; se define utilizando `...:`

```
let rangoCerrado = 1...3 // 1, 2, 3
```

### Rango abierto

Este tipo de rango incluye el valor de la izquierda, pero no incluye el extremo derecho; se define utilizando `..<`:

```
let rangoAbierto = 1..<3 // 1, 2
```

### Rango sin límite por un lado

Este tipo de rango incluye todos los posibles valores desde un valor específico o hasta un valor específico.

- Rango sin límite inferior:



Para definir un rango sin límite inferior, basta con especificar el valor final e indicar si es abierto ( $..<$ ) o cerrado ( $...$ ).

```
let sinLimiteInferiorAbierto = ..<2 // -∞, ..., -1, 0, 1
```

```
sinLimiteInferiorAbierto.contains(-1) // true
```

```
sinLimiteInferiorAbierto.contains(2) // false
```

```
sinLimiteInferiorAbierto.contains(10) // false
```

```
let sinLimiteInferiorCerrado = ...2 // -∞, ..., -1, 0, 1, 2
```

```
sinLimiteInferiorCerrado.contains(-1) // true
```

```
sinLimiteInferiorCerrado.contains(2) // true
```

```
sinLimiteInferiorCerrado.contains(10) // false
```

- Rango sin límite superior:

En este caso, debe especificarse el valor inicial del rango, pero no el final. Cabe resaltar que los rangos sin límite superior solo pueden definirse como rangos cerrados.



```
let sinLimiteSuperior = 2... // 2, 3, 4, ..., ∞
```

```
sinLimiteSuperior.contains(-1) // false
```

```
sinLimiteSuperior.contains(2) // true
```

```
sinLimiteSuperior.contains(10) // true
```

**En los rangos abiertos, se usó el símbolo  $-\infty$  y  $\infty$ . Sin embargo, en realidad, sí existe un valor límite, y lo define cada tipo de dato. Para números enteros dependiendo de la arquitectura es un número enorme.**



Los arreglos, así como los `strings` y los demás tipos de datos, contienen algunas funciones y propiedades que sirven para obtener información de ellos. Por ejemplo, el caso de `contains` en los rangos indica si un valor específico está incluido dentro del rango. Para conocer más de estas funciones y propiedades de rangos, puedes visitar la [documentación de Apple](#).

## Ejercitación en el Playground de Xcode

1. Define una constante que sea un rango con los números del 0 al 100 inclusive.
2. Define una constante que sea un rango con todos los números hasta el 300 sin incluirlo.

# Arreglos

Los arreglos son un tipo de dato que permiten guardar elementos de un mismo tipo. Al igual que en cualquier otro lenguaje, es probablemente el tipo de datos compuesto más usado.

## Declaración

Para declarar un arreglo (por ejemplo de tipo `Int`), se puede hacer lo siguiente:

`Array<Int>` o como `[Int]`;

sin embargo, la segunda forma es la más usada.

Por lo tanto, la forma más común para declarar un arreglo es especificando el tipo de dato entre corchetes: `[tipo_de_dato]`, donde `tipo_de_dato` puede ser un tipo de dato básico (`Int`, `String`, etc.) o un tipo de dato compuesto.

## Inicialización

La inicialización de un arreglo puede hacerse sin valores iniciales o con estas:

- Sin valores iniciales:

```
var edades: [Int] = []
```

```
var nombres = [String]()
```

`edades` se inicializa como un arreglo vacío de `Int` especificando el tipo de forma explícita, mientras que `nombres` se inicializan como un arreglo de `String` vacío tomando el tipo de forma implícita gracias a la inferencia de tipos.

- Con valores iniciales:

```
var edades: [Int] = [9, 16]
```

```
var nombres = ["Juan", "Nicolás"]
```

Aquí se inicializan igualmente `edades` como un arreglo de `Int` y `nombres` como un arreglo de `String`, pero, en este caso, con valores iniciales.

## Acceder a los elementos de un arreglo

Al igual que en muchos lenguajes, los elementos de un arreglo se acceden usando `[]` e indicando el número de la posición deseada:

```
var nombres = ["Juan", "Nicolás"]
```

```
nombres[1] // "Nicolás"
```

Los arreglos en Swift empiezan en la posición cero.

## Los rangos como subíndices en arreglos

Los rangos pueden usarse también para obtener partes de un arreglo:

```
let frutas = ["Melon", "Pera", "Sandía", "Naranja", "Manzana"]
```

```
let desdeLaTercera = frutas[2...] // ["Sandía", "Naranja", "Manzana"]
```

```
let primerasDosFrutas = frutas[..<2] // ["Melon", "Pera"]
```

## Agregar elementos a un arreglo

Los elementos pueden agregarse al final del arreglo o en una posición específica:

- Agregar elementos al final del arreglo:

Forma 1: `nombres.append("Facundo")`

```
var nombres = ["Juan", "Nicolás"]
```

```
nombres.append("Facundo")
```

```
nombres // ["Juan", "Nicolás", "Facundo"]
```

Forma 2: `nombres += ["Facundo"]`

```
var nombres = ["Juan", "Nicolás"]
```

```
nombres += ["Facundo"]
```

```
nombres // ["Juan", "Nicolás", "Facundo"]
```

En la forma 2, lo que se hace es unir un arreglo de un solo elemento con otro, pero puede incluirse la cantidad de elementos que se deseen (siempre respetando el tipo):

```
var nombres = ["Juan", "Nicolás"]
```

```
let masNombres = ["Facundo", "Tomás"]
```

```
nombres += masNombres // ["Juan", "Nicolás", "Facundo", "Tomás"]
```

- Agregar elementos en una posición específica:

Para esto, se usa la función `insert(_at:)` de los arreglos que recibe el valor a ingresar y la posición en la que se desea poner.

```
var nombres = ["Juan", "Nicolás"]
```

```
nombres.insert("Pedro", at: 1) // ["Juan", "Pedro", "Nicolás"]
```



⚠ Se debe tener cuidado al definir la posición en la que se va a ingresar el valor, pues esta no debe ser superior al tamaño actual del arreglo. De lo contrario, se recibirá un error en tiempo de ejecución.

## Eliminar un elemento de un arreglo

Para eliminar un elemento, se utiliza la función `remove(at:)`, que recibe la posición que se desea eliminar.

```
var nombres = ["Juan", "Pedro", "Nicolás"]
```

```
nombres.remove(at: 1) // ["Juan", "Nicolás"]
```

⚠ Al igual que cuando se inserta un elemento, al eliminar un elemento, se debe tener cuidado de la posición que se define, pues, en caso que esta sea mayor que el tamaño del arreglo o igual a este, se recibirá un error en tiempo de ejecución.

Si se desea eliminar el último elemento del arreglo, puede usarse la función `popLast()`.

```
var nombres = ["Juan", "Pedro", "Nicolás"]
```

```
nombres.popLast() // ["Juan", "Pedro"]
```

Existen también otras funciones que permiten eliminar el primer elemento, un conjunto de elementos o incluso eliminarlos todos. Puedes consultar dichas funciones en la [documentación de Apple](#).

## Dos propiedades importantes de los arreglos

- `isEmpty` retorna `true` cuando el arreglo está vacío.
- `count` retorna la cantidad de elementos.

```
let nombres = ["Juan", "Nicolás"]
```

```
nombres.count // 2
```

nombres.isEmpty // false

### Ejercitación 3

1. ¿Cuál es el problema en el siguiente código?: `var ciudades: [Int] = ["Lima", "Buenos Aires", "Rio de Janeiro"]`

- a. La inicialización del arreglo: debe agregarse () al inicializar un arreglo.
- b. La inicialización del arreglo: no pueden agregarse valores iniciales al momento de crear la variable.
- c. Tipo de dato del arreglo: el tipo de dato no coincide con el tipo de los valores.
- d. La inicialización del arreglo: No debe especificarse el tipo de dato si se incluyen valores iniciales.

2. Dado el siguiente arreglo: `var frutas = ["Melon", "Pera", "Sandía", "Naranja", "Manzana"]`, escriba un rango para obtener: `["Pera", "Sandía", "Naranja"]`.

3. Consulta la documentación de Apple y, con el arreglo de frutas del ejercicio anterior, elimina los primeros 2 elementos.

## Sets (conjuntos)

### Declaración e inicialización

```
var frutasOpcion1 = Set<String>()
```

```
var frutasOpcion2: Set<String> = []
```

En ambos casos, se inicializó un `Set` vacío de tipo `String`, indicando el tipo de dato entre "<" y ">".

Puede, también, inicializarse un `Set` con valores:

```
let frutas: Set = ["azúcar", "crema", "cacao"]
```

### Agregar elementos a un Set

Para agregar elementos al `Set`, se usa la función `insert(_)`, que recibe el dato que se va a incluir:

```
frutas.insert("Manzana") // ["Manzana"]
```

```
frutas.insert("Pera") // ["Manzana", "Pera"]
```

Un `Set` descarta los duplicados:

Como se mencionó anteriormente, un `Set` almacena datos sin repetirlos. Si seguimos con el ejemplo, obtenemos lo siguiente:

```
frutas.insert("Manzana") // No hace nada! Pues "Manzana" ya se encuentra en el Set
```

```
frutas // ["Manzana", "Pera"]
```

La función `insert(_)` retorna una tupla con dos valores: `inserted` y `memberAfterInsert`, con los que se puede validar si el valor fue incluido o no en el `Set`.

Para determinar si un elemento está incluido en un `Set`, se usa la función `contains(_)`:

```
let ingredientes: Set = ["azúcar", "crema", "cacao"]
```

```
ingredientes.contains("crema") // true
```

```
ingredientes.contains("CREMA") // false
```

## Eliminar elementos de un Set

Para eliminar un elemento de un `Set`, usamos la función `remove`:

```
var vocales: Set<Character> = ["a","e","i","o","u"]
```

```
vocales.remove("a") // ["e","i","o","u"]
```

## Unión e intersección de conjuntos

`Set` soporta las operaciones matemáticas clásicas sobre conjuntos:

```
let impares: Set = [1, 3, 5, 7, 9]
```

```
let pares: Set = [0, 2, 4, 6, 8]
```

let primos: Set = [2, 3, 5, 7]

impares.union(pares) // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

impares.intersection(pares) // []

## Ejercitación 4 en el Playground de Xcode

Considera la siguiente clasificación y luego lleva a cabo las tareas propuestas a continuación.

Lenguajes Modernos: Go, Kotlin, Swift, TypeScript

Lenguajes Maduros: C#, Java, C++

Lenguajes Viejos: C, Fortran, Cobol

1. Declara cada uno de los tres conjuntos: `lengModernos`, `lengMaduros`, `lengViejos`.
2. Escribe un conjunto `lengTodos` que incluya las tres categorías.
3. Muestra un código de ejemplo para determinar si un lenguaje pertenece a una categoría dada.

## Diccionarios

El diccionario es un tipo de dato muy usado en Swift. Permite asociar valores de forma `Key - Value`: `Key` que es la llave para acceder al dato, y `Value` es el valor en el que estamos interesados.

Veamos con ejemplos:

```
var respuestasHttp = [200: "OK",
```

```
403: "Acceso denegado",
```

```
404: "No encontrado",
```

```
500: "Error interno en el servidor"]
```

El tipo de dato de `respuestasHttp` es un diccionario `[Int: String]`, donde `Key` es un `Int` y `Value` es un `String`.

## Declaración

Al igual que para declarar un arreglo, existen dos formas para declarar un diccionario: `Dictionary<Int, String>` o `[Int: String]`. Ambos casos declaran un diccionario donde la `Key` es de tipo `Int` y el `Value` es de tipo `String`.

## Inicialización

Los diccionarios pueden también inicializarse con o sin valores iniciales.

- Sin valor inicial:

```
var libros: [Int: String] = [:]
```

- Con valor inicial:

```
var libros = [123: "La vuelta al mundo en 80 días",  
145: "Relatos de un náufrago",  
274: "Cien años de soledad"]
```

## Acceder a valores de un diccionario

Al igual que los Arrays, para acceder a un valor, se usa `[]`. Si bien los Arrays permiten acceder mediante un subíndice (p. ej., `edades[0]`), los diccionarios son accedidos mediante una `Key` (clave) definida por el usuario.

En términos generales, podemos decir que para un diccionario `[tipo_key: tipo_value]`, la estructura es la siguiente:

```
dict[key] = value
```

Aquí, `key` es de tipo `tipo_key` y `value` es de tipo `tipo_value`.

```
var libros = [123: "La vuelta al mundo en 80 días",
```

145: "Relatos de un naufrago",

274: "El Aleph"]

libros[145] // "Relatos de un naufrago"

Los diccionarios retornan `Optional<tipo_value>`.

Al acceder a un diccionario mediante `[]`, el valor de retorno es `Optional<TipoValue>`, puesto que, si el elemento no es encontrado, se retorna `nil`.

## Agregar valores a un diccionario

Usando el operador subscript `[]`, podemos agregar una entrada nueva o actualizar una existente:

var libros = [123: "La vuelta al mundo en 80 días",

145: "Relatos de un naufrago",

274: "El Aleph"]

libros[280] = "El principito" // Agrega un nuevo valor asociado a la key 280

libros[145] = "Pulgarcito" // Actualiza el valor asociado a la key 145

## Eliminar un valor de un diccionario

Para eliminar un valor, basta con asignarle el valor `nil` a la entrada:

var libros = [123: "La vuelta al mundo en 80 días",

145: "Relatos de un naufrago",

274: "El Aleph"]

libros[274] = nil // Eliminamos "El Aleph"

## Ejercitación 5 en el Playground de Xcode

1. Crea una variable de tipo `diccionario` que contenga las notas de los alumnos:

```
| Nombre | Nota |  
|-----|-----|  
| Alberto | 10 |  
| Ignacio | 4 |  
| Pedro | 7 |
```

2. Tendrás una lista de ciudades y necesitarás una estructura para almacenar cuáles fueron visitadas y cuáles no. Escribe la declaración para almacenar dicha información.

```
| Ciudad | Visitada |  
|-----|-----|  
| CABA | Si |  
| Mar del Plata | No |  
| Santa fe | No |  
| Rosario | Si |
```

3. Agrégale a la estructura anterior una ciudad que hayas visitado y una que te gustaría visitar.

4. Luego de tus vacaciones, pudiste conocer esa ciudad que querías. Actualiza la estructura para indicar que ya la has visitado.