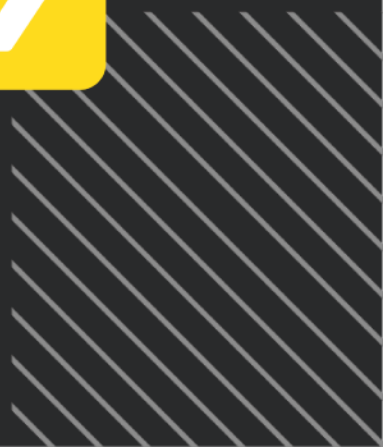




Swift





Tema 5. Funciones

Objetivos

Luego de navegar esta cápsula, deberás ser capaz de hacer lo siguiente:

- definir y llamar funciones;
- crear funciones con valores pasados por parámetro o valores de retorno;
- retornar valores de forma implícita en funciones;
- utilizar parámetros con etiquetas y valores por defecto en funciones;
- utilizar parámetros variados en funciones; y
- crear funciones que reciban o devuelvan otras funciones.

Funciones

Las funciones son código contenido en un bloque que realiza una tarea específica. A este bloque de código se le da un nombre para identificar qué hace, y con ese nombre se “llama” a la función para que realice su tarea cuando se necesite.

Para definir una función, se debe utilizar la palabra reservada `func` al inicio; luego, el nombre, seguido de los parámetros entre paréntesis (si se requieren); y, finalmente, especificar el valor de retorno (si se requiere) agregando `->` y el tipo del valor de retorno al final. El bloque de código debe incluirse dentro de llaves `{ }`.

Ejemplo:

```
func saludarMundo() {  
  
    print("Hola Mundo!")  
  
}  
  
func saludar(persona: String) -> String {  
  
    return "Hola, " + persona + "!"
```

}

Para devolver un valor en una función, debe agregarse la palabra reservada `return`, seguida del valor que se desea retornar.

La función `saludarMundo()` no recibe parámetros ni tiene valor de retorno. Por su parte, `saludar(persona:)` recibe un parámetro de tipo `String` y retorna un valor de tipo `String`.

Estas funciones se invocan de la siguiente manera:

```
saludarMundo() // "Hola Mundo"
```

```
let saludoMatias = saludar(personas: "Matias")
```

```
print(saludoMatias) // "Hola, Matias!"
```

Como vemos, existen diferentes formas de definir parámetros y valores de retornos para funciones en Swift.

Actividades de repaso

1. ¿Qué son las funciones?

Son códigos contenidos en un bloque que realiza una tarea específica.

Son códigos contenidos en un bloque que realiza diversas tareas sin especificidad.

Son códigos que nos permiten mejorar las *apps*.

Son códigos que nos permiten cubrir nuestras *apps* de ataques.

Justificación: hemos visto en esta primera parte que las funciones son código contenido en un bloque que realiza una tarea específica.

2. De las siguientes opciones, selecciona los pasos correspondientes para definir una función:

Usar la palabra `func` al inicio.

El nombre debe estar seguido de los parámetros en paréntesis `()`.

Especificar el valor de retorno, el bloque debe incluirse dentro de llaves `{}`.

Especificar el valor de retorno, el bloque debe incluirse dentro de corchetes `[]`.

El nombre debe estar seguido de los parámetros en corchetes `[]`.

Justificación: como vimos a lo largo del desarrollo, para definir una función, se debe utilizar la palabra reservada `func` al inicio; luego, el

nombre, seguido de los parámetros entre paréntesis (si se requieren); y, finalmente, especificar el valor de retorno (si se requiere) agregando `->` y el tipo del valor de retorno al final. El bloque de código debe incluirse dentro de llaves `{ }`.

A continuación, se profundizará en algunos ejemplos:

Funciones sin parámetros

No es necesario que una función tenga parámetros de entrada. La función necesita ser declarada con paréntesis (en este caso, vacíos) y, además, debe ser invocada. Aquí hay un ejemplo de una función que devuelve un `String` sin ningún parámetro ingresado:

```
func holaMundo() -> String {  
  
    return "Hola Mundo"  
  
}
```

Y se la invoca de la siguiente manera:

```
let saludoMundo = holaMundo()  
  
print(saludoMundo) // "Hola Mundo"
```

Funciones con múltiples parámetros

Las funciones pueden requerir uno o múltiples parámetros, los cuales son escritos entre los paréntesis, separados por comas. Supongamos que esta función recibe dos parámetros: el nombre de la persona como `String` y un `Bool` indicando si ya se la saludó previamente. Para poder utilizar dichos parámetros dentro de la función, es necesario asignarles un nombre. Por ejemplo, para el nombre de la persona, usaremos `persona`, y para el booleano, usaremos `yaFueSaludada`: Los nombres que se les dan a los parámetros son los que se usan dentro de la función para poder acceder a dichos valores y existen solo dentro del alcance de la función donde son definidos.

```
func saludar(persona: String, yaFueSaludada: Bool) -> String {  
  
    if yaFueSaludada {
```

```
return "Hola de nuevo " + persona

} else {

return "Hola " + persona

}

}
```

Se la invoca de la siguiente manera:

```
let saludarNuevamente = saludar(persona: "Josefina", yaFueSaludada:
true)

print(saludarNuevamente) // "Hola de nuevo Josefina"
```

Funciones sin valores de retorno

No es necesario definir siempre un tipo de retorno. Acá podemos ver una versión de la función `saludar(persona:)` que imprime un `String` en vez de retornarlo:

```
func saludar(persona: String) {

print("Hola, " + persona + "!")



}
```

Podemos ver que, al no haber valor de retorno, no es necesario escribir `return`. Se la invoca de la siguiente manera:

```
saludar(persona: "Javier") // "Hola, Javier!"
```

Funciones con múltiples valores de retorno

Es posible usar una tupla como valor de retorno cuando la intención es que una misma función retorne más de un valor. En el siguiente ejemplo, la función `obtenerHorarioDeCierre(esFinDeSemana:)` recibe un `Bool` indicando si es o no fin de semana, y retorna dos valores enteros indicando el cierre del comercio:



```
func obtenerHorarioDeCierre(esFinDeSemana: Bool) -> (hora: Int,
minutos: Int) {

if esFinDeSemana {

return (13, 00)

} else {

return (18, 30)

}

}
```

Esta función retorna dos `Int`, etiquetados hora y minutos para poder accederlos después con esos nombres. A continuación, se guarda el resultado en la variable `horario` y se accede a ambos valores de la tupla como se aprendió en la sección “Tipos de datos II”.

```
let horario = obtenerHorarioDeCierre(esFinDeSemana: false)

print("El horario de cierre es \$(horario.hora):\$(horario.minutos)") // "El
horario de cierre es 18:30"
```

Recuerda que, al ser una tupla, puedes retornar 2 o más valores.

Valores de retorno implícitos

Si todo el cuerpo de la función es una única expresión, el valor puede retornarse de forma implícita, es decir, que se puede evitar el `return`. Usando de ejemplo anterior `saludar(persona:)`, se muestra a continuación cómo en estas dos versiones su comportamiento es el mismo:

```
func saludar(persona: String) -> String {

return "Hola, " + persona + "!"

}

print(saludar(persona: "Matias")) // "Hola, Matias!"
```

Y de forma implícita (omitiendo el return):

```
func saludar(persona: String) -> String {  
  
    "Hola, " + persona + "!"  
  
}  
  
print(saludar(persona: "Matias")) // "Hola, Matias!"
```

En este ejemplo, se llamó la función `saludar(persona:)` directamente como parámetro del `print()`. Al ejecutarse el `print`, se llama a la función `saludar(persona:)`, y su valor de retorno es tomado directamente como parámetro del `print()`. Se explicará esto con más profundidad en la función como tipo, más adelante.

Ejercitación 1 (en el Playground de Xcode)

1. Crea una función llamada `suma` que devuelva el resultado de sumar dos valores enteros pasados como parámetros.
2. Supón que un grupo de estudiantes está formado por 11 mujeres y 16 varones. Crea una función que se llame `calcularPorcentaje` y retorne el porcentaje de hombres y el de mujeres en la clase.

Etiquetas

Etiquetas para parámetros (*argument labels*)

Se pueden agregar etiquetas previas a los nombres de los parámetros, separados por un espacio. Esto permite que, al invocarlas, las funciones sean más expresivas, sin perder la claridad en los nombres de los parámetros al utilizar la función. Cuando se defina la función, la estructura sería, por ejemplo, la siguiente:

```
func nombreDeLaFuncion(etiqueta nombreDelParametro: Int) {  
  
    // En el cuerpo de la función, nombreDelParametro referirá al  
    parámetro  
  
}
```



```
nombreDeLaFuncion(etiqueta: 1)
```

```
// En la invocación, la etiqueta se usará para referir al  
nombreDelParametro
```

Volvamos al ejemplo de `saludar(persona:)`; podemos agregar un parámetro `lugarDeOrigen` con el la etiqueta `de`:

```
func saludar(persona: String, de lugarDeOrigen: String) -> String {  
  
    return "Hola \ \(persona)! ¿Qué tal el viaje desde \ \(lugarDeOrigen)?"  
  
}
```

```
print(saludar(persona: "Lucrecia", de: "La Plata")) // "Hola Lucrecia!  
¿Qué tal el viaje desde La Plata?"
```

Omitir etiquetas para parámetros

Es posible omitir la etiqueta del parámetro, utilizando un guión bajo (`_`). Esto permite que en la invocación no sea necesario poner ni la etiqueta ni el nombre del parámetro. En este caso, al parámetro `persona` le agregamos una etiqueta omitida con un `_`:

```
func saludar(_ persona: String, de lugarDeOrigen: String) -> String {  
  
    return "Hola \ \(persona)! ¿Qué tal el viaje desde \ \(lugarDeOrigen)?"  
  
}
```

```
print(saludar("Lucrecia", de: "La Plata")) // "Hola Lucrecia! ¿Qué tal el  
viaje desde La Plata?"
```

Más sobre funciones

Valores por defecto

Es posible, también, definir un valor por defecto para un parámetro, asignándolo luego de la definición del tipo.

Ejemplo:

```
func saludar(_ persona: String, yaFueSaludada: Bool = false) -> String {  
  
  if yaFueSaludada {  
  
    return "Hola de nuevo " + persona  
  
  } else {  
  
    return "Hola " + persona  
  
  }  
  
}
```

```
saludar("Matias") // "Hola, Matias"
```

```
saludar("Jose", yaFueSaludada: true) // "Hola de nuevo Matias"
```

En la primera invocación, se omite el parámetro `yaFueSaludada`, por lo que la función utilizará el valor por defecto `false`. En la segunda, utilizará el valor que le pasamos por parámetro, en este caso, `true`.

Parámetros variados

En ocasiones, puede requerirse que se ingrese cierta cantidad de parámetros pero sin saber exactamente cuándo van a ser, y que dicha cantidad pueda variar entre las llamadas de una misma función. Para cumplir con esto, se utilizan los parámetros variados (un parámetro variado acepta cero o más valores del tipo específico). Estos valores serán tomados dentro de la función como un arreglo de elementos de ese tipo. Para definir este tipo de comportamiento, se añaden tres puntos (...) luego de especificar el tipo de parámetro:

```
func suma(_ numeros: Int...) -> Int {  
  
  var total = 0
```

```
for numero in numeros {  
  
    total += numero  
  
}  
  
return total  
  
}
```

```
suma(10,20) // 30
```

```
suma(1,2,3,4,5,6,7,8) // 36
```

Dentro del cuerpo de `suma(_:)`, el parámetro `números` es evaluado como un arreglo del tipo `[Int]`.

Ejercitación 2 (en el Playground de Xcode)

1. Crea una función que reciba dos parámetros y retorne si el primero es divisible por el segundo. Agrégale etiqueta al segundo parámetro y permite que la etiqueta del primero sea omitida.
2. Usando parámetros variados de una función, calcula el promedio de un listado de números.
3. Recibiendo la información de distancia (en kilómetros) y velocidad (en horas), crea una función que calcule y retorne (implícitamente) el tiempo que le tomará a una persona recorrer dicha distancia. Se conoce que la velocidad promedio de una persona al caminar es de 5km/h; utiliza este dato como valor por defecto para la función.

La función como tipo

Toda función tiene un tipo específico, formado a partir de los tipos de parámetros y el tipo del valor de retorno.

```
func suma(_ a: Int, _ b: Int) -> Int {  
  
    return a + b  
  
}
```

```
func multiplica(_ a: Int, _ b: Int) -> Int {  
  
    return a * b  
  
}
```

Estas dos funciones, si bien son diferentes, poseen el mismo tipo de función: `(Int, Int) -> Int`. Se lee como “Una función que tiene dos parámetros, ambos del tipo `Int` y retorna un valor del tipo `Int`”.

Otro caso es aquel en que una función no tiene parámetros ni valor de retorno:

```
func holaMundo(){  
  
    return "hola mundo"  
  
}
```

En este caso, el tipo de la función es `() -> Void` o “una función que no tiene parámetros de entrada y devuelve `Void`”.

`Void` es una palabra clave usada para determinar que la función no devuelve nada.

El tipo de una función se puede usar como cualquier otro tipo en Swift para definir constantes o variables, pasándolo como parámetro de una función, retornarlo como valor de retorno, entre otras funciones. Veremos algunos a continuación:

El tipo Función como variable

Podemos definir una variable o constante como un tipo de una función y asignarle una función. Si tomamos el ejemplo anterior, obtenemos lo siguiente:

```
func suma(_ a: Int, _ b: Int) -> Int {  
  
    return a + b  
  
}
```

```
var funcionMatematica: (Int, Int) -> Int = suma
```

Como suma, tiene el tipo `(Int, Int) -> Int`, visto anteriormente. Puede ser asignado a la variable `funcionMatematica`. Una vez hecho esto, esta variable puede ser invocada:

```
var resultado = funcionMatematica(2,3)
```

```
print(resultado) // 5
```

Al ser una variable, podemos asignarle otro valor, como es el caso de otra función, por ejemplo, `multiplica`:

```
func multiplica(_ a: Int, _ b: Int) -> Int {  
  
    return a * b  
  
}
```

```
funcionMatematica = multiplica
```

Si llevamos a cabo la misma invocación, obtenemos lo siguiente:

```
var resultado = funcionMatematica(2,3)
```

```
print(resultado) // 6
```

El tipo Función como parámetro

Como se indicó anteriormente, se puede usar el tipo `Función` como parámetro de otra función:

```
func imprimirFuncionMatematica(_ funcionMatematica: (Int, Int) -> Int,  
    _ a: Int, _ b: Int) {  
  
    print("Resultado: \{funcionMatematica(a, b)\}")  
  
}
```

```
imprimirFuncionMatematica(suma, 3, 5) // "Resultado: 8"
```

En este ejemplo, se define una función `imprimirFuncionMatematica` que tiene tres parámetros. El primero es una función llamada `funcionMatematica` y es del tipo `(Int, Int) -> Int`. Se le puede pasar cualquier función que tenga ese tipo, como es el caso de `suma`. El segundo y tercero corresponden a dos valores de tipo `Int`. De esta forma, podríamos pasarle diferentes funciones (mientras que cumplan con ese tipo) y podría imprimir el resultado.

El tipo Función como valor de retorno

Podemos usar un tipo de función como el tipo de la variable de retorno. Se hace simplemente escribiendo el tipo de la función inmediatamente después de la flecha de retorno de la función.

En este ejemplo, se definen dos funciones: `irHaciaAdelante` e `irHaciaAtras`, las cuales suman uno o restan uno respectivamente al valor pasado por parámetro. Ambas funciones tienen el mismo tipo `(Int) -> Int`.

```
func irHaciaAdelante(_ posicionActual: Int) -> Int {  
  
    return posicionActual + 1  
  
}  
  
func irHaciaAtras(_ posicionActual: Int) -> Int {  
  
    return posicionActual - 1  
  
}
```

Por ejemplo, supongamos una función `avanzarEnTablero` que recibe como parámetro un `Bool` indicando si se debe retroceder o no, se retorna una función u otra respectivamente:

```
func avanzarEnTablero(retroceder: Bool) -> (Int) -> Int {  
  
    if retroceder {  
  
        return irHaciaAtras  
  
    } else {
```

```
return irHaciaAdelante
```

```
}
```

```
}
```

Como el valor de retorno es `(Int) -> Int`, se pueden retornar funciones que tengan ese tipo.

Ejercitación 3 (en el Playground de Xcode)

1. Escribe la notación correcta del tipo función de las siguientes funciones:

- a. `func localizar(telefono: Int) -> String { ... }`
- b. `func imprimir() { ... }`
- c. `func generarTicket(id: Int, _ info: String) -> Int { ... }`

2. Crea una función `darNoticia` que reciba por parámetro una nota como valor entero y una función llamada `imprimir`. Esta función deberá imprimir en la consola (usando `print()`) un mensaje `String` pasado por parámetro. La función `darNoticia` llamará a esta función pasada por parámetro con el mensaje “Aprobado” si la nota es mayor que 7 o igual a 7, y “Desaprobado” si no lo es.