




공식문서 읽기 : Functions

Functions - The Swift Programming Language (Swift 5.5)

Functions are self-contained chunks of code that perform a specific task. You give a function a name that identifies what it does, and this name is used to "call" the function to perform its task when needed.

 <https://docs.swift.org/swift-book/LanguageGuide/Functions.html>

Functions는 함수라는 뜻으로, 특정 작업을 수행하는 독립적인 코드이다. 함수는 모두 이름을 가지고 있고 호출할 때 이름으로 호출한다.

Swift에서는 매개변수 이름이 없는 C언어 스타일부터 각 매개변수에 대한 이름, 인수, 레이블이 있는 복잡한 Objective-C 스타일의 함수까지 표현 가능하다. 매개변수는 함수 호출을 간단하게 하기 위해 default 값을 줄 수도 있고 실행 후 전달된 매개변수를 수정 가능하게 하는 입력 매개변수로 전달될 수도 있다.

Swift의 모든 함수는 매개변수의 타입과 반환 타입으로 구성된다. 이를 사용하면 다른 함수로 매개변수를 전달하고 함수에서 함수로 쉽게 값들을 반환할 수 있다.

유용한 기능을 캡슐화하기 위해 다른 함수 속에 함수를 작성해 함수를 중첩할 수도 있다.

Defining and Calling Functions

함수를 정의할 때 함수에 매개변수라고 불리는 입력값의 타입을 선언할 수 있다.

물론 반환값이라고 하는 출력값의 타입도 선언할 수 있다. 매개변수와 반환값은 선택사항으로 함수 정의에 반드시 필요한 것은 아니다.

모든 함수는 이름을 가진다. 함수를 이용하려면 이름을 가지고 호출하고 매개변수를 타입에 맞게 입력하면 된다.

이때 매개변수의 인수는 항상 선언될 때의 순서와 동일하게 작성해야 한다.

```
func greet(person: String) -> String {
    let greeting = "Hello, \(person)!"
    return greeting
}

print(greet(person: "Anna"))
// prints "Hello, Anna!"
print(greet(person: "Brian"))
// prints "Hello, Brian!"
```

위에서 정의한 함수를 실제로 사용한 코드이다.

`greet` 함수에서 첫 줄에 `greeting`이라는 상수를 선언한 것을 볼 수 있다. 이렇게 선언된 `greeting`은 `return`에 다시 사용되어 반환된다. 이렇게 만들어진 함수는 매개변수를 다르게 해서 몇 번이고 호출할 수 있다.

Functions Parameters and Return Values

Swift에서의 함수의 매개변수와 반환값은 아주 유연하게 사용할 수 있다. 이름이 지정되지 않은 단일 매개변수를 사용하는 함수부터 매개변수 이름과 다른 매개변수 옵션을 사용하는 복잡한 함수도 선언할 수 있다.

Functions Without Parameters

함수를 선언할 때 매개변수를 정의하지 않고 만들 수 있다.

```
func sayHelloWorld() -> String {  
    return "Hello, World!"  
}  
print(sayHelloWorld())  
// prints "Hello, World!"
```

이렇게 선언한 함수는 사용할 때 매개변수가 없지만 함수 이름 뒤에 빈 괄호를 붙여야 사용할 수 있다.

Functions With Multiple Parameters

함수는 여러 개의 매개변수를 가질 수 있다.

```
func greet(person: String, alreadyGreeted: Bool) -> String {  
    if alreadyGreeted {  
        return greetAgain(person: person)  
    } else {  
        return greet(person: person)  
    }  
}
```

이렇게 선언한 함수는 person, alreadyGreeted라는 이름의 매개변수를 갖는다. 여기서 중요한 것은 아까 선언한 greet(person:) 함수와 지금 선언한 greet(person:alreadyGreeted) 함수는 다른 함수라는 것이다. 이름은 같지만 매개변수의 개수가 다르기 때문에 두 함수는 서로 다르다고 할 수 있다.

Functions Without Return Values

함수를 만들 때 반환값을 정의하지 않을 수도 있다.

```
func greet(person: String) {  
    print("Hello, \"(person)!")  
}  
greet(person: "Dave")
```

위 코드에서 선언한 `greet` 함수는 " → 반환타입 " 이 없다. 엄밀히 말하면 반환값이 정의되어 있지 않더라도 어떠한 값을 반환하게 되는데 이는 `Void` 타입의 특수값을 반환하게 된다. 이는 그냥 빈 튜플이며 `()`로 작성된다.

```
func printAndCount(string: String) -> Int {
    print(string)
    return string.count
}

func printWithoutCounting(string: String) {
    let _ = printAndCount(string: string)
}
printAndCount(string: "Hello, World!")
// prints "Hello, World!" and returns a value of 13
printWithoutCounting(string: "Hello, world!")
// prints "Hello, World!" but doesn't return a value
```

```
printAndCount(string: "Hello, World!")
printWithoutCounting(string: "hello, world!")
```

13

Xcode에서 직접 실행했을 때의 화면이다. 콘솔에서는 모두 "Hello, World!"를 출력하지만 `printAndCount(string:)` 함수는 리턴하는 값을 가지고 있다.

`printWithoutCounting(string:)` 함수는 상수에 `printAndCount`의 반환값을 저장하는데, `_`를 사용해 이를 사용하지 않겠다는 표현을 했다. 이렇게 반환값을 사용하지 않겠다고 무시는 할 수 있겠지만 반환값이 있는 함수에게 반환하지 말라고는 할 수 없다.

Functions with Multiple Return Values

매개변수가 여러개 있을 수 있는 것처럼 반환값도 여러개 있을 수 있다. 이 때 여러개의 값을 튜플로 만들어 반환하게 된다.

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..
```

위 코드는 Array의 최소값, 최대값을 구해서 반환하는 함수이다. 반환값에 (min: Int, max: Int) 타입을 반환한다고 정의한 것을 볼 수 있다. 즉 두 개의 Int값을 가진 튜플을 반환한다는 의미이다.

Optional Tuple Return Types

만약 함수에서 튜플이 반환되면 모든 튜플에 값이 없을 수도 있다. 반환타입에 (Int, Int)?와 같이 물음표를 붙여 값이 존재하지 않을 수도 있다고 알려줘야 한다. 여기에서 (Int, Int)?는 (Int?, Int?)와 다르다.

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..
```

아까 만든 함수에 발생할 수 있는 오류는 빈 array가 입력되었을 경우이다. 이 경우에는 반환할 수 있는 Int 값이 존재하지 않기 때문에 nil을 반환해야 한다. 이럴 때 미리 반환값의 타입을 옵셔널로 선언하면 오류를 방지할 수 있다. 물론 이렇게 사용하면 옵셔널 값이 반환되기 때문에 옵셔널 바인딩을 사용해야 한다.

Functions With an Implicit Return

만약 함수의 내용 전체의 길이가 한 줄이라면 그 줄이 return값이 될 수 있다.

```
func greeting(for person: String) -> String {
    "Hello, \(person)!"
}
print(greeting(for: "Dave"))
// prints "Hello, Dave!"

func anotherGreeting(for person: String) -> String {
    return "Hello, \(person)!"
}
```

```
print(anotherGreeting(for: "Dave"))  
// prints "Hello, Dave!"
```

위처럼 함수의 코드가 한 줄이라면 `return`을 생략하고 `return`값을 정할 수 있다.

Function Argument Labels and Parameter Names

각 함수에 있는 매개변수는 `argument label`과 매개변수 이름이 존재한다. `argument label`은 함수가 호출될 때 각각의 매개변수들의 입력값을 정해줄 때 쓰인다. 매개변수 이름은 함수 내에서 함수 구현에 사용된다. 만약 `argument label`이 정의되지 않으면 매개변수 이름을 `argument label`로 사용한다. 모든 매개변수는 고유한 이름을 가져야 한다. 물론 모든 매개변수가 동일한 `argument label`을 가질 수는 있지만 좋은 방법은 아니다.

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {  
  
}  
  
someFunction(firstParameterName: 1, secondParameterName: 2)
```

위 코드는 `argument label`이 정의되지 않은 함수의 예를 나타낸 것이다. `argument label`은 정의되지 않았지만 실제 함수를 호출할 때 매개변수 이름을 `argument label`로 사용하는 것을 볼 수 있다.

Specifying Argument Labels

`argument label`을 선언하는 방법은 매개변수 이름 앞에 `argument label`을 적는 것이다.

```
func someFunction(argumentLabel parameterName: Int) {  
  
}  
  
func greet(person: String, from hometown: String) -> String {  
    return "Hello \(person)! Glad you could visit from \(hometown)."  
}
```

```
print(greet(person: "Bill", from: "Cupertino"))  
// prints "Hello Bill! Glad you could visit from Cupertino."
```

두 함수를 비교해보자. person이라는 매개변수는 argument label이 없으므로 함수가 호출될 때 매개변수 이름인 person이 argument label처럼 쓰인다. 하지만 hometown이라는 매개변수는 from이라는 argument label이 존재한다. 그러므로 함수가 호출될 때 from으로 매개변수에 입력값을 주는 것을 확인할 수 있다.

Omitting Argument Labels

만약 argument label이 너무 귀찮아서 사용하기 싫다면 argument label 자리에 _를 쓴다.

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {  
  
}  
someFunction(1, secondParameterName: 2)
```

Default Parameter Values

매개변수 입력값으로 아무것도 주지 않아도 자동으로 특정 값을 입력하게 만드는 방법이 있다.

바로 default value를 정의하는 방법인데, 방법은 아래와 같다.

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) {  
  
}  
  
someFunction(parameterWithoutDefault: 3) // parameterWithDefault: 12  
someFunction(parameterWithoutDefault: 4, parameterWithDefault: 7)
```

위의 코드에서 선언 뒤에 =12와 같이 값을 미리 정해주는 것이다. 이렇게 하면 실제로 매개변수에 아무런 값이 입력되지 않을 경우 미리 정해둔 default value로 함수를 실행할 수 있다.

이렇게 default value를 사용할 때 매개변수의 순서를 지켜주는 것이 좋다. 앞쪽에 default value를 정의하지 않은 매개변수들을 두고, 뒤쪽에 default value를 사용한 매개변수들을 두는 것이 좋다.

Variadic Parameters

Variadic 매개변수는 매개변수의 입력값에 0개부터 아주 많게도 넣을 수 있는 매개변수를 말한다.

매개변수의 타입 뒤에 ...를 붙여 Variadic 매개변수를 사용할 수 있다.

이렇게 입력된 Variadic 매개변수는 적절한 타입의 Array로 함수 내에서 사용된다.

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}

arithmeticMean(1, 2, 3, 4, 5) // 3.0
arithmeticMean(3, 8.25, 18.75) // 10.0
```

In-Out Parameters

함수의 매개변수는 기본적으로 상수이다. 함수 내에서 매개변수의 값을 변경하려고 하면 컴파일 오류가 발생한다. 하지만 매개변수를 수정하고싶고 이를 함수가 끝난 후에도 유지하고 싶다면 in-out 매개변수를 사용한다.

매개변수의 타입 앞에 inout 키워드를 작성하면 in-out 매개변수를 만들 수 있다. in-out 매개변수의 동작 원리는 일단 상수 타입으로 매개변수를 받고, 이 매개변수를 수정 가능하게 하기 위해 함수에서 다시 전달되어 기존의 매개변수 값을 대체하게 된다.

이런 방식으로 사용되다보니 in-out 매개변수에는 입력값으로 변수만 올 수 있다. 즉 상수가 올 수 없다는 것이다. in-out 매개변수를 호출해서 사용할 때는 변수의 이름 바로 앞에 &를 작성해 해당 변수가 함수 내에서 수정될 수 있다는 것을 알려야 한다. 이렇게 선언된 in-out 매개변수는 default값과 variadic 매개변수를 사용할 수 없다.

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

실제로 사용된 예를 보면 이해하기 쉽다. 위의 코드처럼 in-out 매개변수는 값의 수정이 가능한 것을 볼 수 있다.


```
var someInt = 3
var anotherInt = 107
swap(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// prints "someInt is now 107, and anotherInt is now 3"
```

Function Types

모든 함수는 함수 타입을 가지고 있다. 이는 매개변수의 타입과 반환값의 타입에 의해 결정된다.

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

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

위의 코드에 선언된 두 함수의 타입을 확인해보면 다음과 같다. 두 함수 모두 $(Int, Int) \rightarrow Int$ 타입이다. 이는 두 개의 `Int` 매개변수를 입력받고 `Int` 값을 반환한다는 의미이다.

```
func printHelloWorld() {
    print("Hello, World!")
}
```

위 코드와 같이 매개변수와 반환값이 없는 경우에는 $() \rightarrow Void$ 타입이라고 나타낼 수 있다.

Using Function Types

함수 타입을 Swift의 다른 타입들처럼 사용할 수 있다. 예를 들어, 상수나 변수의 타입을 함수 타입으로 만들고 여기에 함수를 저장할 수 있다.

```
var mathFunction: (Int, Int) -> Int = addTwoInts
print("Result: \(\mathFunction(2,3))")
// prints "Result: 5"
```

위와 같이 함수를 변수나 상수에 저장할 수 있다.

물론 다른 타입들과 마찬가지로 변수로 선언되었다면 다른 함수로 바꾸는 것도 가능하다.

```
mathFunction = multiplyTwoInts
print("Result: \(\mathFunction(2,3))")
// prints "Result: 6"
```

Function Types as Parameter Types

함수 타입을 다른 함수의 매개변수로도 사용할 수 있다.

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(\mathFunction(a, b))")
}

printMathResult(addTwoInts, 3, 5)
// prints "Result: 8"
```

위에서 선언한 `addTwoInts`를 재사용해서 코드를 만들 수 있다.

위의 코드에 선언된 `printMathResult` 함수는 `a`, `b` 값을 입력받아 `mathFunction`이라는 함수 타입의 매개변수의 입력값으로 사용한 값을 출력하는 기능을 갖는다.

Function Types as Return Type

물론 함수 타입을 반환값으로도 사용할 수 있다. 함수 반환값의 타입을 적는 곳에 함수 타입을 반환값의 타입으로 사용한다.

```
func stepForward(_ input: Int) -> Int {
    return input + 1
}

func stepBackward(_ input: Int) -> Int {
    return input - 1
}
```

```

func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}

var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)

print("Counting to zero:")

while currentValue != 0 {
    print("\(currentValue)...")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")

```

위의 코드에서는 (Int) → Int 함수 타입의 stepForward 함수와 stepBackward 함수를 선언하고 이를 chooseStepFunction의 반환값으로 사용했다. 이를 활용해 moveNearerToZero라는 상수에 함수를 넣어주고 이것으로 함수들을 사용할 수 있게 된다.

Nested Functions

지금까지의 함수들은 전역 범위에서 정의된 함수들을 살펴보았다. 중첩 함수라고 하는 함수는 다른 함수 속에 함수를 정의하는 것이다. 중첩 함수는 정의된 함수 외부에서는 접근할 수 없다. Enclosing 함수는 중첩 함수를 가지고 있는 함수를 말한다.

Enclosing 함수를 사용해 중첩 함수 중 하나를 다른 범위에서도 사용할 수 있다.

```

func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    func stepForward(_ input: Int) -> Int { return input + 1 }
    func stepBackward(_ input: Int) -> Int { return input - 1 }

    return backward ? stepBackward : stepForward
}

var currentValue = -4
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)

print("Counting to zero:")

while currentValue != 0 {
    print("\(currentValue)...")
    currentValue = moveNearerToZero(currentValue)
}

```

```
}  
print("zero!")
```

위와 같은 예제를 중첩함수를 통해 표현한 것이다.