함수(Function)

함수는 특정 작업을 수행하기 위한 독립적인 코드 집합이다.

보통의 함수는 값을 전달받아 이 값을 이용해 특정 일을 수행하고, 그 결과를 리턴하는 형태를 취한다.

여기서 전달 받는 값을 '파라미터'라 하고, 리턴하는 값을 '반환 값'이라고 한다.

함수에 식별할 수 있도록 명명하며, 작업이 필요할 때 함수의 이름을 호출하여 사용한다.

Swift 의 함수 문법은 인자가 없는 C 언어 스타일의 함수부터 지역, 전역 인자 이름을 가지는 복잡한 Objective-C 스타일 메소드까지 모두 표현할 수 있다.

인자는 기본 값을 가지며 간단한 함수를 호출하는데 인자를 넘기거나 받을 수 있다. 이 인자들은 함수에 실행이 끝난 다음 변경되어 넘겨진다.

Swift 의 모든 함수는 타입을 가지며 함수의 인자 타입과 반환 타입을 고려해야 한다. Swift 에 다른 타입과 마찬가지로 함수에서 다른 함수로 인자를 넘기고 함수에서 함수로 인자를 반환 받는 것이 쉽다.

함수는 캡슐화를 위해 중첩된 함수 범위 내에서 작성할 수 있다.

간단히 함수와 관련된 용어 몇 가지만 짚고 시작한다:

- 인자(Argument): 함수를 호출 할 때 넘겨주는 변수(상수) 혹은 그 이름
- 매개변수(Parameter): 함수가 정의될 때 함수가 전달받게 되는 변수(상수) 혹은 그 이름

같은 것 같지만 개념 상 의미가 다름에 주의하자. 함수에 전달하느냐 함수가 받느냐 그 차이다.

함수(Function) 정의

함수를 정의할 때, 함수에 입력되는 인자 값에 이름을 정할 수 있으며, 출력 시 값의 타입 정해야 한다.

모든 함수는 함수 이름을 가지며, 이는 어떤 작업을 하는지에 대한 설명이다. 함수를 사용하기 위해 이름과 함수 인자의 타입과 일치하는 값을 넘기도록 호출한다. 함수의 입력 값은 함수의 인자 목록에 순서와 항상 일치해야 한다.

함수 인자와 반환 값(Function Parameters and Return Values)

Swift 에서 함수 인자와 반환 값은 극도로 유연하다. 이름없는 한 개 인자를 사용하는 단순한 기능성 함수에서 명시적 인자와 다른 인자 옵션을 가지는 복잡한 함수까지 정의할 수 있다.

함수는 func 라는 키워드를 통해 선언이 가능하다. 대충 아래와 같은 형식으로 정의한다.

```
func FUNCTION_NAME(PARAMETER_NAME: PARAMETER_TYPE, ...) -> RETURN_TYPE {
    FUNCTION_IMPLEMENTATIONS
    return RETURN_VALUE
}
```

이렇게 적어놓으면 못 알아보니 그냥 예제 몇 가지를 적어보자.
아래 함수는 이름을 넣으면 "Your name is 이름" 이라는 문자열을 리턴하는 함수다.

```
func getDecoratedName(name: String) -> String {
    return "Your name is W(name)"
}

var str = getDecoratedName("seorenn")
// str = "Your name is seorenn"
print(str)
```

리턴 타입이 함수 선언 오른쪽으로 온다는 것을 잘 보자. 리턴값은 항상 -> 라는 표기 이후에 따라오기 때문에 구분하는데는 어렵지 않을 것이다.
매개변수 선언이 Objective-C 혹은 C 와는 다르게 변수 이름이 먼저 오고 그 다음에 타입이 온다. 이는 스위프트의 변수나 상수 선언과 비슷한 문법이니 역시 헷갈릴 것은 없을 것이다.

```
func sayHello(personName : String) -> String{
        let greeting = “Hello, ” + personName + “!”
        return greeting
}

println(sayHello( “Anna” ))
println(sayHello( “Brian” ))
```

sayHello 함수를 String 인자 값을 넘기면서 호출한다.

sayHello(“Anna”)와 같이 호출한다. 이는 함수가 문자열 값을 반환하며 sayHello 는 println 함수 호출 안에 감싸져 있으며 반환된 값이 출력된다.

sayHello 함수는 시작하면 새로운 문자열 상수를 greeting 로 정의하며 personName 에 간단한 인사 메시지를 설정한다. 이 인사말은 `return` 키워드를 사용하여 함수 밖으로 넘겨지게 된다. `return greeting` 이 곧 호출되면 함수는 실행 종료하고 현재 greeting 값이 반환된다.

복수 입력 인자(Multiple Input Parameters)

어떤 언어든 그렇지만 매개변수를 두 개 이상 선언하는 것도 당연히 가능하다.
함수는 복수 입력 인자를 가질 수 있으며, 함수의 괄호 안에서 콤마로 분리하여 사용된다.

```
func sum(a: Int, b: Int) -> Int {
    return a + b
}
var result = sum(1, 2)
```

인자 없는 함수(Functions Without Parameters)

함수는 입력 인자를 정의할 필요는 없다.
다음은 항상 같은 5와 6을 더한 값을 반환하는 인자 없는 함수 예제이다
입력 인자가 없더라도 괄호()를 꼭 써야한다.

```
func fivePlusSix() -> Int {
    return 5 + 6
}
var result = fivePlusSix()
```

값을 반환하지 않는 함수(Functions Without Return Values)

함수에는 리턴 값도 없을 경우 리턴 타입 생략이 가능하다.
함수는 반환 타입을 정의할 필요는 없다. 다음은 반환 값 없는 함수 예제이다.
반환 값이 없기 때문에 `->`를 사용할 필요가 없고 타입도 쓸 필요도 없음.

```
func printSomeLog() {
    println("log A")
    println("log B")
}
printSomeLog()
```

리턴 타입으로 튜플의 활용 (다중 값을 반환하는 함수(Functions with Multiple Return Values))

리턴 타입으로 튜플을 사용해 함수가 멀티 리턴 값을 가지게 할 수 있다.

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

let tup = plusAndMinus(a: 6, b: 5)
let first = tup.0
let second = tup.1
print("W(first) , W(second)")
```

물론 튜플에 이름을 붙이는 것도 그대로 쓸 수 있다.

```
func plusAndMinus(a: Int, b: Int) -> (plus: Int, minus: Int) {
    return (a + b, a - b)
}
let tup = plusAndMinus(a: 6, b: 5)
let first = tup.plus
let second = tup.minus
print("W(first) , W(second)")
```

옵셔널 튜플 반환 타입(Optional Tuple Return Types)

함수에서 반환되는 튜플 타입은 값이 없는 가능성이 있음. 옵셔널 튜플 반환 타입은 튜플이 nil 일 수도 있다는 사실을 반영한다.

옵셔널 튜플 반환 타입은 닫는 괄호 다음에 물음표(?)를 사용하여 사용한다. (Int, Int)? 나 (String, Int, Bool)?.

옵셔널 튜플 타입 `(Int, Int)?` 는 옵셔널 타입을 가지는 튜플 `(Int?, Int?)` 와는 다름.

```
func plusAndMinus(a: Int, b: Int) -> (plus: Int, minus: Int)? {
    return (a + b, a - b)
}
let tup = plusAndMinus(a: 6, b: 5)
let first = tup?.c
let second = tup?.d
print("₩(first) , ₩(second)")
```

함수 인자 이름(Function Parameter Names)

함수의 인자에 이름을 지어준다.

```
func someFunction(parameterName: Int) {
// function body goes here, and can use parameterName // to refer to the argument value for that parameter
}
```

이들 인자 이름은 함수 안에서만 사용 가능하며 함수 호출할 때에는 사용할 수 없음. 이러한 종류의 인자 이름은 지역 인자 이름으로 불리는데 함수 내에서만 오직 사용 가능하기 때문임.

외부 인자 이름(External Parameter Names)

때론 함수를 호출할 때 각각의 인자에 이름을 붙이는게 유용할 때가 있는데 함수로 던져진 각 인자의 목적을 가리키기 위함이다.

함수 사용자에게 함수를 호출 할 때 인자에 이름을 지어 주길 원하면 각 인자에게 외부 인자 이름을 지역 인자 이름에 붙이도록 정의한다.

함수를 호출 할 때 특정 매개변수(Parameter)에 이름(별명)을 붙임으로써 좀 더 읽기 쉬운 코드를 만들 수도 있다.
호출하는 쪽에서도 이름이 붙음으로써 가독성이 높아질 수 있다.
그냥 매개변수 이름 앞에다 쓰고 싶은 이름을 붙이면 된다.

```
func sumAandB(valueA a: Int, valueB b: Int) -> Int {
    return a + b
}

var result = sumAandB(valueA: 10, valueB: 20)
```

다음은 문자열을 합치는 함수 예제이다.

```
func join(s1: String, s2: String, joiner: String) -> String {

    return s1 + joiner + s2
```

```
}
```

위 함수를 호출 할 때 세 개의 문자열 인자의 목적이 뚜렷하지 않음.

```
join("hello", "world", ", ")

// returns "hello, world"
```

따라서 이들 문자열 값의 목적을 뚜렷하게 하기 위해 외부 인자 이름을 각 함수 인자에 정의한다.

```
func join(string s1: String, toString s2: String, withJoiner joiner: String)

    -> String {

        return s1 + joiner + s2

}
```

외부 인자 이름을 정의함에 따라 각 인자들의 목적이 뚜렷하게 나타남.

```
join(string: "hello", toString: "world", withJoiner: ", ")

// returns "hello, world"
```

함수 선언 시 매개변수 이름 앞에 샵(#)을 붙이면 그 이름 그대로 호출 시 인자의 별명으로 쓸 수 있게 된다.

매개변수 기본 값(Parameter Default Value)

특정 매개변수에는 기본값을 넣어서 필요하지 않을 경우 함수 호출을 단순화 시킬 수 있는 방법이 스위프트에서도 제공된다.

함수의 정의에 부분으로 인자에 기본값을 정의할 수 있다. 기본 값은 정의되면 함수 호출될 때 인자를 생략할 수 있다.

기본 값을 가지는 인자는 함수 인자 목록의 마지막에 위치한다. 이는 기본 값을 가지지 않는 인자가 같은 순서를 사용함을 보장하며 매 경우 같은 함수가 호출되도록 명확하게 한다.

```
func sumOrInc(value: Int, incValue: Int = 1) -> Int {
    return value + incValue
}

var result = 0;
```

```
result = sumOrInc(value: result)                // result = 1
result = sumOrInc(value: result, incValue: 5)   // result = 6
```

이 예에서는 incValue 라는 매개변수가 기본값 1 을 가지도록 설계되었다.

joiner 인자가 기본 값을 가지도록 하는 join 함수의 예제.

```
func join(string s1: String, toString s2: String,

    withJoiner joiner: String = " ") -> String {

        return s1 + joiner + s2

}
```

joiner 에 값이 있는 경우에 다음과 같이 함수를 호출함.

```
join(string: "hello", toString: "world", withJoiner: "-")

// returns "hello-world"
```

만약 joiner 에 값이 없는 경우 기본 값이 대신 사용하도록 다음과 같이 함수를 호출함.

```
join(string: "hello", toString: "world")

// returns "hello world"
```

동적인 매개변수(Variadic Parameters)

가변 인자는 특정 타입의 0 이상의 값을 받아 들임. 가변 갯수 파라메터를 사용함으로써 함수 호출시 입력 값들이 임의의 갯수가 될수 있다고 정할 수 있음.

가변 인자 타입 이름 뒤에 마침표 세개(...)를 추가하여 가변 인자를 작성함.

가변 인자에 넘긴 값은 적절한 타입의 배열로 만들어짐. Double… 타입인 numbers 이름 가변 인자는 함수 내에서 [Double]타입의 numbers 로 불리는 상수 배열로 만들어짐.

```
func printManyStrings(strs: String ...) {
    for str in strs {
        println(str)
    }
}

printManyStrings("A", "B", "C")
printManyStrings("A", "B", "C", "D", "E", "F")
```

인자 타입 뒤에 '...' 을 붙이면 자동으로 가변 매개변수 리스트가 완성된다. 그냥 Array 이터레이션 하듯이 쓰면 된다.

호출하는 쪽에서는 마음껏 (타입에 맞게) 인자를 집어 넣을 수 있다.

다만 모든 언어의 동적 매개변수의 특성과 비슷하게, **동적 매개변수는 매개변수 정의에서 가장 마지막 위치에만 정의가 가능하다**는 점에 주의하자.

함수는 대부분 한 개의 가변 인자를 가지며 이는 인자 목록에 마지막에 항상 위치를 하고, 함수 호출 시 여러 인자들과의 모호성을 피하기 위함. 하나 이상의 기본 값을 가지는 인자와 가변 인자를 가지고 있다면, 기본 값을 가지는 인자 뒤에 가변 인자 순으로 위치해야 함.

함수의 매개변수는 상수형

아래 예제를 보면 타 언어와는 다른 점이 보인다.

```
func someFunc(a: Int, b: Int) {
    a++      // ERROR
    b++      // ERROR
}
```

위 함수 안에 구현된 두 줄의 코드는 모두 오류가 발생한다. 스위프트의 함수 매개변수는 기본적으로 상수(Constant)로 넘어오기 때문에 직접 값을 바꾸는 것이 불가능하다. 호출하는 쪽에서 인자를 상수가 아닌 변수로 넣는다 해도 그 값이 복사되어서 상수로 전달되기 때문에 똑같다.

하지만 값을 바꾸게 할 수 있는 방법도 있다. 매개변수 이름 앞에 var 를 붙이면 된다.

```
func someFunc(var a: Int, var b: Int) {
    a++
    b++
}
```

물론 이 값을 바꾸더라도 호출한 쪽에는 아무런 영향이 없다. 함수가 호출될 때 매개변수로 넘어가는 값은 인자의 값에서 복사되어서 넘어가기 때문이다.

레퍼런스 매개변수?

일반적으로 함수를 호출할 때 전달하는 인자는 함수 실행 후 바뀌는 경우가 없어야 한다. 하지만 필요에 의해 인자로 넘기는 변수의 값이 함수에 의해서 바뀌어야 하는 경우가 있다. 스위프트도 이런 경우를 위해 inout 이라는 기능을 만들어 놨다. 역시 매개변수 이름 앞에 선언하면 된다.

```
func incValue(inout value: Int) {
```

```
    value++
}

var v = 0
incValue(&v)      // v == 1
incValue(&v)      // v == 2
incValue(&v)      // v == 3
```

자바나 C 에서 기본 타입변수의 포인터를 넘길 때와 비슷하게, inout 용 인자를 전달 할 때는 앞에 앰퍼선드(&)를 붙여서 호출해야 한다. 이러면 해당 변수가 복사가 아닌 변수의 레퍼런스가 바로 전달되어서 함수가 실행된다. 그리고 보다싶이 inout 으로 선언된 매개변수는 상수가 아닌 변수로 전달된다.

아마도 대표적인 레퍼런스 매개변수 기능을 소개하는 함수는 swap 이 있겠다.

함수 타입(Function Types)

모든 함수는 특정 함수 타입을 가지며, 함수 타입은 인자 타입과 반환 타입으로 만들어짐.

```
func addTwoInts(a: Int, b: Int) -> Int {

    return a + b

}

func multiplyTwoInts(a: Int, b: Int) -> Int {

    return a * b

}
```

위 예제는 두 개의 산수하는 함수이며 addTwoInts 와 multiplyTwoInts 함수로 호출된다. 이들 함수는 각각 두 개의 Int 값을 취하고 Int 값을 적절히 연산하여 결과로 반환한다.

이들 함수의 타입은 `(Int, Int) -> Int` 이며 다음과 같이 읽는다

함수 타입은 두 개의 Int 타입 인자를 가지며 Int 타입의 값을 반환한다.

다음은 인자도 반환 값도 없는 함수의 예제.

```
func printHelloWorld() {

    println("hello, world")
```

```
}
```

이 함수의 타입은 `() -> ()` 또는 함수에 인자는 없고 `Void`를 반환한다.

함수는 특별하지 않으면 `Void`를 항상 반환하며, 이는 Swift 에서 빈 튜플 `()`과 같은 의미이다.

함수 타입 사용(Using Function Types)

함수 타입은 Swift 에 다른 타입들 처럼 사용한다.

예를 들어 함수 타입을 변수나 상수에 할당할 수 있음.

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

위는 두 개의 Int 값을 취하고 Int 값을 반환하는 함수 타입을 가진 mathFunction 변수를 정의한다.

addTwoInts 함수가 참조하는 새로운 변수로 설정한다.

addTwoInts 함수는 mathFunction 변수와 같은 타입이며, Swift 에 타입 확인로 할당이 허용된다.

다음은 mathFunction 이름으로 할당된 함수를 호출하는 예제.

```
print("Result: \(mathFunction(2, 3))")

// prints "Result: 5"
```

타입은 같지만 다른 동작을 하는 함수는 타입이 같은 변수에 할당할 수 있으며, 다음은 해당 예제.

```
mathFunction = multiplyTwoInts

print("Result: \(mathFunction(2, 3))")

// prints "Result: 6"
```

Swift 에선 상수나 변수에 함수를 할당할 때 함수 타입을 추론하도록 내버려둔다.

```
let anotherMathFunction = addTwoInts

// anotherMathFunction is inferred to be of type (Int, Int) -> Int
```

인자 타입으로서 함수 타입(Function Types as Parameter Types)

`(Int, Int) -> Int` 같은 함수 타입은 다른 함수에 인자 타입으로 사용할 수 있다.

다음은 함수 타입을 인자 타입으로 가지는 함수 예제이다.

```
func printMathResult(mathFunction: (Int, Int) -> Int, a: Int, b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(mathFunction: addTwoInts, a: 3, b: 5)
// prints "Result: 8"
```

printMathResult 함수는 세 개의 인자로 정의되며, 첫번째 인자는 mathFunction 로 불리며 `(Int, Int) -> Int` 타입임.

printMathResult 가 호출되면 addTwoInts 함수, 정수 3 과 5 가 넘어오고 addTwoInts 함수에 정수 3 과 5 를 넘겨 8 의 값을 얻음. 넘어온 함수가 어떤 행동을 하는지는 중요한 것이 아니고 어떤 타입이냐가 중요함.(type-safe)

반환 타입으로서 함수 타입(Function Type as Return Types)

다른 함수에서 반환 타입을 함수 타입으로 사용할 수 있다.

반환하는 함수의 반환 화살표(`->`) 뒤에 완전한 함수 타입을 붙여 작성한다.

다음 예제는 값을 증가 또는 감소시키는 함수 예제.

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

이 함수의 반환 타입은 (Int) -> Int 를 반환하는 함수이다.

chooseStepFunction 은 backwards 논리값 인자에 따라 stepForward 함수와 stepBackward 함수중 하나를 반환.

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

```
var currentValue = 3

let moveNearerToZero = chooseStepFunction(currentValue > 0)

// moveNearerToZero now refers to the stepBackward() function
```

이제 moveNearerToZero 상수는 stepBackward 함수를 참조하도록 할당됨.

다음은 moveNearerToZero 상수에 stepBackward 함수를 참조하는지 확인하는 예제.

```
println("Counting to zero:")

// Counting to zero:

while currentValue != 0 {

    println("W(currentValue)... ")

    currentValue = moveNearerToZero(currentValue)

}

println("zero!")

// 3...

// 2...

// 1...

// zero!
```

함수 속의 함수(Nested Function)

함수 내부에서 또다른 함수를 정의할 수 있으며 이를 중첩 함수라고 함.

중첩 함수는 기본적으로 밖에서는 숨겨져 있으며 중첩 함수 중 하나를 반환하여 다른 범위에서 함수가 사용할 수 있게 함.

다음은 chooseStepFunction 에 중첩 함수로 작성된 예제.

```
func someFunc() {
    func incAandB(a: Int, b: Int) -> Int {
        return a + b
    }
    println(incAandB(0, 0))    // 콘솔에 0이 찍힌다
    println(incAandB(1, 1))    // 콘솔에 2가 찍힌다
}
someFunc()
```

함수 속에서 함수 정의하는거야 뭐 굳이 어려울 것은 없다고 생각된다. 하지만 이 부분이 클로져로 확대되면 좀 난해해 질 수도 있다.

익명함수

#
일반적으로#함수의#경우#ixqf 키워드와#함수면을#선언하고#사용하지만#효율적인#코드를#
작성하기#위해#함수명을#선언하지#않고#바로#함수#몸체만#만들어#사용하는#일회용#함수를#
익명함수+Dqrq|p rxv#Ixqfwlrq,#혹은#클로저+Forvxuh,라고#한다1#
함수의#파라미터로#값이나#변수가#아닌#함수를#사용하고#싶을#때/#함수명을#사용하지#않고#
함수의#몸체만#이용할#때#사용한다1#
함수명을#가지지#않는#함수/#호출함수를#사용할#수#없다1
#
일반#함수#
ixqf#함수명+파라미터명-#자료형/ll,#DA#반환형,##
#　　살행문#
c#
#
익명함수#
~#파라미터명-#자료형/ll,#DA#반환형,#q#실행문#c#
#
~#파라미터명-#자료형/ll,#q#실행문#c#
#
~#파라미터명,#q#실행문#c#
#
~#파라미터명#1q#실행문#c#
#
#
일반함수#예제#
ixqf#wrgd|+b#p rqwk##Vwulqj/#b#gd|##Vwulqj,#DA#Vwulqj~
####uhwxuq#%오늘은#_+p rqwk,월#_+gd|,일입니다1%
c#

익명 함수

ydu#p dwkIxqfwlrq=#Vwulqj/#Vwulqj,DAVwulqj#@#~+p rqwk##Vwulqj/#gd|##Vwulqj,#DA#Vwulqj#q

```
####uhwxuq#%오늘은#_+p rqwk,월#_+gd|,일입니다1%
¢
subqw+%Uhvxow#+p dwkIxqfwlrq+%4%/#%5%,,%,


ydu#p dwkIxqfwlrq=+Vwulqj/#Vwulqj,0A+,#0#-
####+p rqwk##Vwulqj/#gd|##Vwulqj,#0A#-,#q
####subqw##%오늘은#_+p rqwk,월#_+gd|,일입니다1%,
¢
p dwkIxqfwlrq+%4%/#%5%,#
#
```