Swift Function Parameters and Return Values

In this tutorial, we will learn about function parameters and return values in Swift with the help of examples.

Swift Function Parameters

A function parameter is a value that is accepted by a function.

Before you learn about function parameters and return values, make sure to know about Swift functions (/swift-programming/functions).

Let's see an example,

```
func addNumbers(a: Int, b: Int) {
  var sum = a + b
  print("Sum:", sum)
}

addNumbers(a: 2, b: 3)
```

Output

```
Sum: 5
```

In the above example, the function `addNumbers()` takes two parameters: `a` and `b`. Notice the line,

> **Note**: The values **2** and **3** passed during the function call are also called as arguments.

Function Parameter with Default Values

In Swift, we can provide default values to function parameters.

We use the `=` operator to provide default values. For example,

```
func addNumbers( a: Int = 7,  b: Int = 8) -> Int {
  var sum = a + b
  print("Sum:", sum)
}

// function call with two arguments
addNumbers(a: 2, b: 3)

// function call with one argument
addNumbers(a: 2)

// function call with no arguments
addNumbers()
```

Output

```
Sum: 5
Sum: 10
Sum: 15
```

In the above example, notice the function definition

Here's how this program works

1. addNumbers(a: 2, b: 3)

Both values are passed during the function call. Hence, these values are used instead of the default values.

2. addNumbers(a: 2)

Only one the value for parameter `a` is passed during the function call. So, the default value is used for parameter `b`.

3. addNumbers()

No value is passed during the function call. Hence, default value is used for both parameters `a` and `b`.

Function With Argument Label

In Swift, we can use the argument labels to define a function in an expressive and sentence-like manner. For example,

```
func sum(of a: Int, and b: Int) {
  ...
}
```

Here, the `sum()` function has argument labels: `of` and `and`.

While calling a function, we can use the argument label instead of parameter names. For example,

```
sum(of: 2, and: 3)
```

Here, we are calling `sum()` with values **2** and **3** using argument labels.

```
  print("Sum:", a + b)
}

sum(of: 2, and: 3)
```

Output

```
Sum: 5
```

> **Note:** It makes more sense while calling the function as `sum(of: 2, and: 3)` instead of `sum(a: 2, b: 3)`.

Omit Argument Labels

We can also omit the argument label by writing a `_` before the parameter name. For example,

```
func sum(_ a: Int, _ b: Int) {
  print("Sum:", a + b)
}

sum(2, 3)
```

Output

```
Sum: 5
```

> **Note:** If we use `_` before the parameter name, then we can call the function without an argument label or parameter name.

parameters in Swift.

Variadic parameters allow us to pass a varying number of values during a function call.

We use the `...` character after the parameter's type to denote the variadic parameters. For example,

```
  for num in numbers {
    result = result + num
  }

  print("Sum = \(result)")
}

// function call with 3 arguments
sum(numbers: 1, 2, 3)

// function call with 2 arguments
sum (numbers: 4, 9)
```

Output

```
Sum = 6
Sum = 13
```

In the above example, we have created the function `sum()` that accepts variadic parameters. Notice the lines,

```
sum(numbers: 1, 2, 3)

sum(numbers: 4, 9)
```

Here, we are able to call the same function with different arguments.

> **Note:** After getting multiple values, `numbers` behave as an array so we are able to use the `for` loop to access each value.

Function With inout Parameters

When we define a function parameter, the function parameter cannot be modified inside the function body. For example,

Here, the above code will generate an error: `Error: cannot assign to value: 'name' is a 'let' constant`. This is because the parameter behaves as a constant value.

To overcome this, we need to define the parameter as an `inout` parameter. For example,

```
func changeName(name: inout String) {
  if name == "Ross" {
      name = "Joey"
  }
}

var userName = "Ross"
print("Before: ", userName)

changeName(name: &userName)

print("After: ", userName)
```

Output

```
Before:  Ross
After:  Joey
```

In the above example, we have declared a function `changeName()` that accepts `inout` parameter `name`.

While calling a function with inout parameter, we must use the `ampersand(&)` sign before a variable name passed as an argument.

```
changeName(name: &userName)
```

Swift Function Return Values

A function may or may not return value. If we want our function to return some value, we use the **return statement** and **return type**. For example,

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

let result = addNumbers(a: 2, b: 3)

print("Sum:", result)
```

Output

```
Sum: 5
```

In the above example, we have created a function named `addNumbers()`. The function adds two numbers and returns the `sum`.

```
return sum
```

The returned value is stored in the `result` variable.

Function with Return Multiple Values

A function can also return multiple values. For example,

Here, the return statement returns two values: `message` and `marks`.

Also, `-> (String, Int)` specifies the return type `message` and `marks`. And, the order should be the same. That is, the first returned value should be a string and second should be an integer.

Example: Multiple Return Values

```
func compute(number: Int) -> (Int, Int, Int) {

  var square = number * number

  var cube = square * number

  return (number, square, cube)
}

var result = compute(number: 5)

print("Number:", result.0)
print("Square:", result.1)
print("Cube:", result.2)
```

Output

```
Number: 5
Square: 25
Cube: 125
```

In the above example, the `compute()` function accepts a number and computes the square and the cube of the number. Notice the line,

```
return (number, square, cube)
```

Here, the function is returning the number, its square and the cube value at once.

Next Tutorial:
Swift Nested Functions (/swift-programming/nested-functions)

Previous Tutorial:
Swift Functions (/swift-programming/functions)

Swift Tutorial

Swift Functions

(/swift-programming/functions)

Swift Tutorial

Swift Function Overloading

(/swift-programming/function-overloading)

Swift Tutorial

Swift Nested Functions

(/swift-programming/nested-functions)

Swift Tutorial

Swift Closures

(/swift-programming/closures)