10
Higher-Order Functions

Written by Ehab Yosry Amer

A higher-order function takes one or more functions as a parameter. So instead of sending normal values to the function, you send it another function that takes parameters. The normal function is called a first-order function. Too many “functions” there.

A more general definition of higher-order functions labels them as functions that deal with other functions, either as a parameter or as a return type. In this chapter, you’ll start with sending a function as a parameter. Then you’ll move to having a function return another function.

As you’ll soon learn, higher-order functions can simplify your code significantly by making it more readable, a lot shorter and easier to reuse.

A simple text printer

Before going deeply into what higher-order functions are or how Swift makes them fun, consider this example for a text printer.

Create a new playground and add the following:

```
class TextPrinter {
  var formatter: ParagraphFormatterProtocol
  init(formatter: ParagraphFormatterProtocol) {
    self.formatter = formatter
  }

  func printText(_ paragraphs: [String]) {
    for text in paragraphs {
      let formattedText = formatter.formatParagraph(text)
      print(formattedText)
    }
  }
}

protocol ParagraphFormatterProtocol {
  func formatParagraph(_ text: String) -> String
```

```
}
```

The code above is for a text printer. All it does is receive an array of strings, run each through a formatter and then print the formatted value. `TextPrinter` doesn't specify how the formatting operation is done. But it does specify that it needs an object that conforms to `ParagraphFormatterProtocol`, which will implement this functionality.

Add this new class to your playground:

```
class SimpleFormatter: ParagraphFormatterProtocol {
  func formatParagraph(_ text: String) -> String {
    guard !text.isEmpty else { return text } // 1
    var formattedText =
      text.prefix(1).uppercased() + text.dropFirst() // 2
    if let lastCharacter = formattedText.last,
       !lastCharacter.isPunctuation {
      formattedText += "." // 3
    }
    return formattedText
  }
}
```

COPY

This simple formatter conforms to the required protocol, and its format function does the following:

1. Makes sure the provided string is not empty. If it is, then there's nothing to format and the formatter returns the string as-is.
2. Capitalizes the first character
3. Checks whether the last character is punctuation. If it isn't, then it adds a full-stop ".".

Now, use this new class in your playground:

```
let simpleFormatter = SimpleFormatter()
let textPrinter = TextPrinter(formatter: simpleFormatter)

let exampleParagraphs = [
  "basic text example",
  "Another text example!!",
  "one more text example"
]

textPrinter.printText(exampleParagraphs)
```

COPY

You created an instance of the simple formatter you just defined and used it to create an object of the printer. The output in the console from the code above should be:

```
Basic text example.
Another text example!!
One more text example.
```

COPY

First-order functions

The two main methods here are **first-order functions**. Both `ParagraphFormatterProtocol.formatParagraph(_:)` and `TextPrinter.printText(_:)` take normal values. You didn't go into the higher-order ones yet.

Another way to perform the formatting is by adding a new first-order function to string arrays. Add this new extension at the end of your playground:

```
extension Array where Element == String {
  func printFormatted(formatter: ParagraphFormatterProtocol) {
    let textPrinter = TextPrinter(formatter: formatter)
    textPrinter.printText(self)
  }
}
```

COPY

This extension adds a method to `Array` only if the type of the stored elements is `String`. So this method won't be available to an array of `Int` or an array of `Any` but only to `String`.

Use it by calling:

```
exampleParagraphs.printFormatted(formatter: simpleFormatter)
```

COPY

This will print the same result as before.

There's nothing wrong at all with the implementations above. They deliver the needed operations and offer a clear integration method to provide different formatting operations. But creating a formatter every time or using the same one over and over is not particularly Swifty. What if there were a way to send the formatting operation itself as a parameter instead of packaging it in an object?

Your first higher-order function

Create a new playground and try this in a new way. Extract the formatting function you created in `SimpleFormatter` from the previous example and add it as a function:

```
func formatParagraph(_ text: String) -> String {
  guard !text.isEmpty else { return text }
  var formattedText =
    text.prefix(1).uppercased() + text.dropFirst()
  if let lastCharacter = formattedText.last,
     !lastCharacter.isPunctuation {
    formattedText += "."
  }
  return formattedText
}
```

Next, create this new extension:

```
extension Array where Element == String {
  func printFormatted(formatter: ((String) -> String)) {
    for string in self {
      let formattedString = formatter(string)
      print(formattedString)
    }
  }
}
```

This extension has the same idea only for `[String]`. But in its method, notice the different type of the parameter. `((String) -> String)` means that this is not a property. Instead, it's a method that takes a `String` as a parameter and returns a `String`.

In the parentheses, anything that precedes the arrow describes parameters, and what follows the arrow defines a return type.

That method you added in the extension is a **higher-order function**. For it to work, it takes another function as a parameter and not a normal property. You can use it like this:

```
let exampleParagraphs = [
  "basic text example",
  "Another text example!!",
  "one more text example"
]

exampleParagraphs.printFormatted(formatter: formatParagraph(_:))
```

You sent the actual function itself as a parameter. Another way to pass the function as a parameter is like this:

```
let theFunction = formatParagraph
exampleParagraphs.printFormatted(formatter: theFunction)
```

The variable `theFunction` is actually a function and not a normal property. You can see its type as `((String) -> String)` and can describe it as a

reference to a function.

Closures

There's also another form of passing over the function as a parameter that you're familiar with:

```
exampleParagraphs.printFormatted { text in
  guard !text.isEmpty else { return text }
  var formattedText = text.prefix(1).uppercased() +
text.dropFirst()
  if let lastCharacter = formattedText.last,
     !lastCharacter.isPunctuation {
    formattedText += "."
  }
  return formattedText
}
```

COPY

In this call, you sent a closure, not a function. Practically, both are the same. This is identical to sending the function reference directly. Another way is to call the function in the closure.

```
exampleParagraphs.printFormatted { formatParagraph($0) }
```

COPY

This form is perfect when the function you want to pass in as a parameter to the higher order function doesn't match the parameter signature. An example of this is if the function you're going to call or use has two parameters, but you'll send one of them as a constant, and the other will be from the main operation. Then you can have a format as follows:

```
aHigherOrderFunction { someOperation($0, "a constant") }
```

COPY

All those examples will work the same, but they each have a slight difference. Think of a closure as if it's a function that's created only for the current scope. The last two examples used a closure, but the first gave a ready-made function directly to the `printFormatted`.

You can still use functions directly as a parameter, but the signature of `someOperation(::)` will need to change to match what `aHigherOrderFunction()` expects. This change is called **Currying** and is covered in the second half of the chapter.

The Swift standard library has many of those operations that use this concept. Each of them does some operation, but there's a small detail inside it that's left out. Take the `sort` function as an example. There are many sorting

algorithms. Some perform better in a small set but not as well in a large one. However, in the end, they all have a comparison between two items that define if the final ordering will be ascending or descending. Which algorithm is used and how it's used aren't relevant when calling the function, except for how the final result is ordered.

A function that returns a function is also considered a higher-order function. The return type is defined by a data-type after the arrow `->`. So instead of defining a class or a struct, you can have something like `(()->())`, which is a function that doesn't take any parameters and doesn't return anything, or as many parameters as you want with any return type you want: `((p1: Type1, p2: Type2, p3: Type3) -> ReturnType)`.

Higher-order functions in the standard library

Swift brought you a few higher-order functions, which deliver several common operations and have a neat way to call them in your code. Some of those functions are:

- map
- compactMap
- flatMap
- filter
- reduce
- sorted
- split

You'll go through them in more detail now to see an example of their usage.

map

`Array.map(_:)` applies an operation on all the elements of the array, and the result is a new array of the same size. This is a much shorter version of iterating over the elements and adding the new items of an operation to a new array. Try this example in a playground:

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
var newNumbers: [Int] = []

for number in numbers {
  newNumbers.append(number * number)
}
```

COPY

```
print(newNumbers)
```

This creates a new array that contains the square of each number. An alternative way to do this using `map(_:)` is this:

```
let newNumbers2 = numbers.map { $0 * $0 }
print(newNumbers2)
```

COPY

This code is much simpler and shorter than the previous one. This is because it uses the shorthand provided by Swift and all the code reductions possible for a Swift closure. But as you learned above, higher-order functions can receive a function instead of a closure. Try this example:

```
func squareOperation(value: Int) -> Int {
  print("Original Value is: \(value)")
  let newValue = value * value
  print("New Value is: \(newValue)")
  return newValue
}
```

COPY

This function squares an `Int` value and prints its original value and its square value. You can use `map(_:)` by sending the new function to it like this:

```
let newNumbers3 = numbers.map(squareOperation(value:))
```

COPY

compactMap

`Array.compactMap(_:)` is like the previous one, but the result array doesn't need to be the same size as the original array. This one filters out nil values from the resulting operation. The operation in the parameter has an optional result, but `map(_:)` didn't allow for that.

A common usage for this method is to transform the items in the array to a different type, knowing that not all the items can be transformed and that you'll get only the elements that were successfully transformed. Consider the following example:

```
func wordsToInt(_ str: String) -> Int? {
  let formatter = NumberFormatter()
  formatter.numberStyle = .spellOut
  return formatter.number(from: str.lowercased()) as? Int
}
```

COPY

The function receives a string and finds out if this string is a number

represented in letters. For example, the word “one” is the number “1”. If it is, then the function returns that number. Otherwise, it returns nil.

```
wordsToInt("Three") // 3
wordsToInt("Four") // 4
wordsToInt("Five") // 5
wordsToInt("Hello") // nil
```

Next, add this function at the end of your playground:

```
func convertToInt(_ value: Any) -> Int? {
  if let value = value as? String {
    return wordsToInt(value)
  } else {
    return value as? Int
  }
}
```

This function receives any value and attempts to convert it to an integer. If it’s a string, it passes it over to the previous function that you added. Otherwise, it attempts to cast it to an integer.

```
convertToInt("one") // 1
convertToInt(1.1) // nil
convertToInt(1) // 1
```

The value “1.1” failed to convert because it’s a `Double`. Now, try it with this sample array:

```
let sampleArray: [Any] = [1, 2, 3.0, "Four", "Five", "sixx", 7.1,
"Hello", "World", "!"]

let newArray = sampleArray.compactMap(convertToInt(_:)) // [1, 2,
4, 5]
```

So far, it looks good. But while working on the project, you learned the values “3.0” and “7.1” should be converted to integers. The problem was not related at all to `compactMap(_:)`, but it’s in the method that you’re using for conversion all over your project. Update it to the following:

```
func convertToInt(_ value: Any) -> Int? {
  if let value = value as? String {
    return wordsToInt(value)
  } else if let value = value as? Double {
    return Int(value)
  } else {
    return value as? Int
  }
}
```

This gave your function the ability to convert doubles to integers. The result changed to “[1, 2, 3, 4, 5, 7]”.

Later, you can refactor and improve `convertToInt(_:)` to convert more values without going through many places in your project. This specific example shows a better usage for a function instead of a code block or a closure as a parameter to a higher-order function.

flatMap

As you saw in `compactMap(_:)`, you can have fewer results than in your original array. But what if the operation you want will provide an array for each item? Consider that you want to calculate the square, the cube and the fourth power of each number in an array. Consider the following function:

```
func calculatePowers(_ number: Int) -> [Int] {
  var results: [Int] = []
  var value = number
  for _ in 0...2 {
    value *= number
    results.append(value)
  }
  return results
}
calculatePowers(3) // [9, 27, 81]
```

COPY

Now, use this function on a sample array of integers:

```
let exampleList = [1, 2, 3, 4, 5]
let result = exampleList.map(calculatePowers(_:))
// [[1, 1, 1], [4, 8, 16], [9, 27, 81], [16, 64, 256], [25, 125,
625]]
result.count // 5
```

COPY

The result contains all the values you want, but it still has the same size as the original array. The only difference is that the array is now an array of arrays `[[Int]]`. For each element in the original collection, you created another collection for it.

If you want to have this in a single, flat array, you can do the additional step of joining them together:

```
let joinedResult = Array(result.joined())
// [1, 1, 1, 4, 8, 16, 9, 27, 81, 16, 64, 256, 25, 125, 625]
```

COPY

But the Swift standard library provides a more convenient way to do that using

fewer steps. Instead of using `map(_:)` use `flatMap(_:)`:

```
let flatResult = exampleList.flatMap(calculatePowers(_:))
// [1, 1, 1, 4, 8, 16, 9, 27, 81, 16, 64, 256, 25, 125, 625]
```

The two ways are equivalent and provide the same result. The latter is just simpler to write.

filter

`filter(_:)` is one of the simplest higher-order functions. As its name suggests, you want to filter a collection of many items based on criteria. If the element meets the criteria, keep it. But if it doesn't, remove it from the list.

This function expects criteria in the form of a function that returns `true` or `false`. To be in the list or not to be in the list.

Say you want to get the list of number words (numbers in English words, not in digits) that contain only four letters for the numbers between zero and one hundred. First, build the whole list of words:

```
func intToWord(_ number: Int) -> String? {
  let formatter = NumberFormatter()
  formatter.numberStyle = .spellOut
  return formatter.string(from: number as NSNumber)
}

let numbers: [Int] = Array(0...100)
let words = numbers.compactMap(intToWord(_:))
// ["zero", "one", "two", ....., "ninety-nine", "one hundred"]
```

Next, create the function that will decide if this string should stay or not:

```
func shouldKeep(word: String) -> Bool {
  return word.count == 4
}
```

Finally, filter out your array:

```
let filteredWords = words.filter(shouldKeep(word:))
// ["zero", "four", "five", "nine"]
```

As mentioned before, you can use a closure here for simplicity:

```
let filteredWords = words.filter { $0.count == 4 }
```

It's correct. But if your actual check is complex and used all over your application, then it would be better to define it as a function rather than as a closure. Your code will look better and have fewer places to maintain if any changes are needed.

reduce

`reduce(_:_:)` is a handy method when you want to merge a group of elements. Common examples include adding or multiplying a group of numbers. But this doesn't mean it works only on numbers. You also can use it on your own custom types. Say you have scores for a soccer team and you want to merge them all to see the team's totals. Consider this `Score` type:

```
struct Score {
  var wins = 0, draws = 0, losses = 0
  var goalsScored = 0, goalsReceived = 0

  init() {}

  init(goalsScored: Int, goalsReceived: Int) {
    self.goalsScored = goalsScored
    self.goalsReceived = goalsReceived

    if goalsScored == goalsReceived {
      draws = 1
    } else if goalsScored > goalsReceived {
      wins = 1
    } else {
      losses = 1
    }
  }
}
```

`Score` can represent a single match or a group of matches. The initializer receives the goals scored and received by the team in one match and sets the `wins`, `draws` and `losses` values according to this score.

Now, build an array of individual match scores:

```
var teamScores = [
  Score(goalsScored: 1, goalsReceived: 0),
  Score(goalsScored: 2, goalsReceived: 1),
  Score(goalsScored: 0, goalsReceived: 0),
  Score(goalsScored: 1, goalsReceived: 3),
  Score(goalsScored: 2, goalsReceived: 2),
  Score(goalsScored: 3, goalsReceived: 0),
  Score(goalsScored: 4, goalsReceived: 3)
]
```

Next, create a function to easily add the scores of two matches. Using operator

overloading is ideal for this:

```
extension Score {
  static func +(left: Score, right: Score) -> Score {
    var newScore = Score()

    newScore.wins = left.wins + right.wins
    newScore.losses = left.losses + right.losses
    newScore.draws = left.draws + right.draws
    newScore.goalsScored =
      left.goalsScored + right.goalsScored
    newScore.goalsReceived =
      left.goalsReceived + right.goalsReceived

    return newScore
  }
}
```

Finally, reduce this team's scores into a single `Score` object using `reduce(_:_:)` and the new operator method you created:

```
let firstSeasonScores = teamScores.reduce(Score(), +)
// Score(wins: 4, draws: 2, losses: 1, goalsScored: 13,
goalsReceived: 9)
```

A nice thing about operators is that you don't need to write the whole signature. The operator symbol by itself is enough, and there's no need to write it like this: `+(left:right:)`.

The first parameter in `reduce(_:_:)` is the initial value you want to add on top of. In this case, you don't have anything other than the existing matches, so an empty `Score` object is enough. But if you wanted to add the second season's scores to this season's scores, then this parameter would be quite handy:

```
var secondSeasonMatches = [
  Score(goalsScored: 5, goalsReceived: 3),
  Score(goalsScored: 1, goalsReceived: 1),
  Score(goalsScored: 0, goalsReceived: 2),
  Score(goalsScored: 2, goalsReceived: 0),
  Score(goalsScored: 2, goalsReceived: 2),
  Score(goalsScored: 3, goalsReceived: 2),
  Score(goalsScored: 2, goalsReceived: 3)
]

let totalScores = secondSeasonMatches.reduce(firstSeasonScores, +)
// Score(wins: 7, draws: 4, losses: 3, goalsScored: 28,
goalsReceived: 22)
```

Instead of supplying an empty score object, you provided the ones from the first season as an initial value, and all the matches from the second season are

added to it.

In this example, you couldn't have created separate types for match scores and season scores. The type returned by `reduce(_:_:)` matches the type of the elements in the array. This is why a single type was used to represent both.

sorted

This might be one of the most frequently used functions in day-to-day work. It sorts an array of elements for you.

Multiple sorting algorithms exist, and each has different complexities and usages. Some are better than others in different situations. For example, the insertion-sort is better for small sets. However, for a larger set, it's not as efficient as others. You can learn about the different sorting algorithms from our book "Data Structures & Algorithms in Swift" — it explains the differences between algorithms.

In the Swift standard library, the sort implementation has changed over the years. With Swift 5.0, the implementation for it uses "Timsort", which is a stable sorting algorithm. Here, stable means that if two elements have the same ordering score, then the order in which they appeared in the original set is maintained in the final ordered result.

To see how sorting works in action, try it on an array of the English words for numbers from zero to ten.

```
let words = ["zero", "one", "two", "three", "four", "five", "six",
"seven", "eight", "nine", "ten"]

let stringOrderedWords = words.sorted()
// ["eight", "five", "four", "nine", "one", "seven", "six", "ten",
"three", "two", "zero"]
```

The sort worked in ascending alphabetical order, starting with "E" and ending with "Z". When the items in the array are comparable, the default ordering is ascending, which is equivalent to `words.sorted(<)`.

`sorted(_:)` expects an expression that takes two parameters and defines if the first should appear before the second.

Using the same `Score` as previously defined, create a new method to check if the two provided match scores are ordered:

```
func areMatchesSorted(first: Score, second: Score) -> Bool {
  if first.wins != second.wins { // 1
    return first.wins > second.wins
```

```
    } else if first.draws != second.draws { // 2
      return first.draws > second.draws
    } else { // 3
      let firstDifference = first.goalsScored - first.goalsReceived
      let secondDifference = second.goalsScored -
second.goalsReceived

      if firstDifference == secondDifference {
        return first.goalsScored > second.goalsScored
      } else {
        return firstDifference > secondDifference
      }
    }
}
```

1. This method gives priority to the winning match if only one of them was a win.
2. If there were no winning matches, give priority to the draw match.
3. If both matches were wins, draws, or losses, get the difference between the goals scored and goals received. If both differences were the same, then return the match that had more goals or the match that had a higher difference. Loss matches with a higher difference give a -ve value so they stay at the end of the ordering.

Sorting a set of matches with this function gives the following result:

```
var teamScores = [
  Score(goalsScored: 1, goalsReceived: 0),
  Score(goalsScored: 2, goalsReceived: 1),
  Score(goalsScored: 0, goalsReceived: 0),
  Score(goalsScored: 1, goalsReceived: 3),
  Score(goalsScored: 2, goalsReceived: 2),
  Score(goalsScored: 3, goalsReceived: 0),
  Score(goalsScored: 4, goalsReceived: 3)
]

let sortedMatches = teamScores.sorted(by:
areMatchesSorted(first:second:))
//[Score(wins: 1, draws: 0, losses: 0, goalsScored: 3,
goalsReceived: 0),
// Score(wins: 1, draws: 0, losses: 0, goalsScored: 4,
goalsReceived: 3),
// Score(wins: 1, draws: 0, losses: 0, goalsScored: 2,
goalsReceived: 1),
// Score(wins: 1, draws: 0, losses: 0, goalsScored: 1,
goalsReceived: 0),
// Score(wins: 0, draws: 1, losses: 0, goalsScored: 2,
goalsReceived: 2),
// Score(wins: 0, draws: 1, losses: 0, goalsScored: 0,
goalsReceived: 0),
// Score(wins: 0, draws: 0, losses: 1, goalsScored: 1,
goalsReceived: 3)]
```


The final result has matches with the largest scoring difference, followed by

draws with the highest scoring and finally losses with the smallest difference and then larger difference.

Sorting is not specific to numbers or comparable types. You can implement `comparable` on any type and use the `<` and `>` operators. Or you can have your own comparison method or methods.

As explained earlier, higher-order functions can do a lot for you. They can simplify a complex functionality for you while keeping a tiny, simple part of its overall operation to be provided at the time of calling.

So far, you went deeply through one of the forms of higher-order functions: functions that expect part of their functionality as a parameter. This is the easy part. Before starting the next section, take a break, grab a cup of coffee (strong coffee is advisable) and do whatever you need to give your mind a break. It will help.

Function as a return type

Functions that return other functions are also higher-order functions. This might not be something you're accustomed to. But it can heavily empower your code and make it simple, despite using an indirect approach and looking complicated at first. Make sure you have cleared your mind so it doesn't feel too complicated.

Currying

Earlier, you learned that sending a function directly to a higher-order function requires the signatures to match. Thus, the example below is best done through a closure because the signatures don't match.

```
aHigherOrderFunction { someOperation($0, "a constant") }
```

COPY

Usually, this is fine. But it's not impossible to remove the usage of the closure and send a function directly.

To remove the closure, you'll first need to change the signature of `someOperation(_:_:)` slightly. The new usage will be something like this:

```
aHigherOrderFunction { curried_SomeOperation("a constant")($0) }
```

COPY

And if having the inner function be passed as a parameter instead of closure:

```
aHigherOrderFunction(curried_SomeOperation("a constant"))
```
COPY

Now, take a step back and observe `curried_SomeOperation`. There are two major changes:

1. The parameters are now sent separately, each in its own brackets.
2. The parameters are swapped. The constant is passed first.

Currying is breaking down a function that takes multiple parameters into a chain of functions in which each takes a single parameter. To see this in action, you'll practice by implementing the above example, going from `someOperation(_:_:)` to `curried_SomeOperation(_:)`.

Create a new playground page and add the following:

```
func aHigherOrderFunction(_ operation: (Int) -> ()) {
  let numbers = 1...10
  numbers.forEach(operation)
}

func someOperation(_ p1: Int, _ p2: String) {
  print("number is: \(p1), and String is: \(p2)")
}

aHigherOrderFunction { someOperation($0, "a constant") }
```
COPY

Now that the abstract has become more concrete, start with the signature of `curried_SomeOperation`. First, break it down: Instead of taking two parameters, create a chain that takes one parameter at a time. Worry about the re-ordering of the two parameters later.

```
func curried_SomeOperation(_ p1: Int) -> ((String) -> ())
```
COPY

The first step is done. Well, it's done, but it could still use a cleanup. The signature looks straightforward. The function takes a single parameter and returns another function with the signature `(String) -> ()`.

Now, it's time for the actual body:

```
func curried_SomeOperation(_ p1: Int) -> (String) -> () {
  return { str in
    print("number is: \(p1), and String is: \(str)")
  }
}
```
COPY

The body returns a closure that takes one parameter of type `String`. This closure is `(String) -> ()`, and inside it, the `p1` that was sent to the original

function is captured and used.

> **Note**: To avoid retain cycles, you'll need to pay attention to the capture list *if* the properties are of reference types and not value types.

So far, you didn't break anything. Try it out:

```
aHigherOrderFunction { curried_SomeOperation($0)("a constant") }
```

COPY

The output is:

```
number is: 1, and String is: a constant
.
.
.
number is: 10, and String is: a constant
```

COPY

Like before, you're using a closure but it's still needed. To eliminate the usage of the closure, you'll *need* to re-order the parameters. But before doing that, you should know *why*.

`aHigherOrderFunction(_:)` expects the type `(Int) -> ()`. Your function after the `Int` parameter is provided doesn't return `()`. Instead, it returns `(String) -> ()`.

If you change the order of the parameters, you'll have the `(Int) -> ()` signature that you need to avoid using the closure. Change `curried_SomeOperation` to this:

```
func curried_SomeOperation(_ str: String) -> (Int) -> () {
  return { p1 in
    print("number is: \(p1), and String is: \(str)")
  }
}
```

COPY

Now that the string is expected first, the closure takes the `Int` as a parameter. This closure now matches the signature expectations of `aHigherOrderFunction(:)`. The usage now is:

```
aHigherOrderFunction { curried_SomeOperation("a constant")($0) }
```

COPY

You can safely reduce it and eliminate the closure:

```
aHigherOrderFunction(curried_SomeOperation("a constant"))
```

COPY

You now understand what currying is and why flipping parameters is useful. But what if there were a way to make a generic `curry` and `flip` so you didn't need to create a curried/flipped version for each method you encounter?

Glad you asked!

A generic currying function

Using an original function's signature `originalMethod(A, B) -> C`, you want to transform it to: `(A) -> (B) -> C`.

Notice that you mention a return type (C), but in `someOperation`, there was no return type. Technically, the return type is `Void`.

The signature of your generic two parameter currying method is:

```
func curry<A, B, C>(
  _ originalMethod: (A, B) -> C
) -> (A) -> (B) -> C
```

COPY

Before going through the implementation, there's one *important* difference between what this generic method is doing and what you did in the previous example. Here, you're *generating* what you did yourself in the example, meaning that this will transform the signature of the original function. Although this might be obvious, it's worth pointing out because it introduces two key differences from the previous currying example.

Type the full curry implementation as follows:

```
func curry<A, B, C>(
  _ originalMethod: @escaping (A, B) -> C
) -> (A) -> (B) -> C {
  return { a in
    { b in
      originalMethod(a, b)
    }
  }
}
```

COPY

The first difference is that there's an additional closure level. Previously, you returned the closure `(Int) -> ()`. Here, you return a longer chain because the first set of arguments is the actual function that you'll transform.

The second difference is the explicit keyword `@escaping`. This is needed because the function you pass to it will execute after the curry function itself finishes.

`curry` is a higher-order function for both of these reasons: It's taking a function as a parameter and it returns a function, too.

Try using this function and compare it with the original `someOperation`:

```
someOperation(1, "number one")
curry(someOperation)(1)("number one")
```

COPY

Using the new method in the playground will give you this expected result, proving that the original function and the curried one are working as expected:

```
number is: 1, and String is: number one
number is: 1, and String is: number one
```

COPY

The return type from `curry(someOperation)` is `(Int) -> (String) -> ()`. This is identical to the first step you did before. Next, you need to flip the arguments to be `(String) -> (Int) -> ()`.

Generic argument flipping

Flipping won't be as confusing as currying because you don't need to make any drastic changes to the signature. Add the following function:

```
func flip<A, B, C>(
  _ originalMethod: @escaping (A) -> (B) -> C
) -> (B) -> (A) -> C {
  return { b in { a in originalMethod(a)(b) } }
}
```

COPY

The type of `flip(curry(someOperation))` is `(String) -> (Int) -> ()`, which is identical to the final `curried_SomeOperation` from your previous example. Try this in your playground:

```
aHigherOrderFunction(flip(curry(someOperation))("a constant"))
```

COPY

You don't need to write any more code to adapt your existing functions to pass them directly to higher-order functions.

For the specific case of flip, you might feel you can design your APIs to completely avoid needing it. But Swift contains some higher-order functions that make flipping necessary.

Generated class methods by Swift

For each method or instance-function you create, Swift creates a class higher-order function for this method. In a new playground page, add this extension:

```
extension Int {
  func word() -> String? {
    let formatter = NumberFormatter()
    formatter.numberStyle = .spellOut
    return formatter.string(from: self as NSNumber)
  }
}
```

COPY

This extension on `Int` generates the word for that number:

```
1.word() // one
10.word() // ten
36.word() // thirty-six
```

COPY

By default, Swift will generate a higher-order function for this method for you:

```
Int.word // (Int) -> () -> Optional<String>
```

COPY

To use it with the same three numbers used above, your code will look like this:

```
Int.word(1)() // one
Int.word(10)() // ten
Int.word(36)() // thirty-six
```

COPY

If you have an array of integers and want to map them to their word equivalents, you can either create a closure and call `word()` on each object or restructure `Int.word` to match the signature requirements for `map(_:)`.

In the same playground, create a new version of `flip`:

```
func flip<A, C>(
  _ originalMethod: @escaping (A) -> () -> C
) -> () -> (A) -> C {
  return { { a in originalMethod(a)() } }
}
```

COPY

This overload moves the part that doesn't take any parameters to the beginning, allowing the parameter to be sent at the end.

```
flip(Int.word)()(1) // one
```

COPY

You can avoid repeating `flip` by creating a new variable that carries the function `flip(Int.word)()`:

```
var flippedWord = flip(Int.word)()
```

You can use your new property directly as if it were a function:

```
[1, 2, 3, 4, 5].map(flippedWord)
// ["one", "two", "three", "four", "five"]
```

On some occasions, the extra brackets `()` can be inconvenient. You might want to convert any generated higher-order function for a method/instance-function that doesn't take any parameters and use it if it was a standard class function. Create this new function:

```
func reduce<A, C>(
  _ originalMethod: @escaping (A) -> () -> C
) -> (A) -> C {
  return { a in originalMethod(a)() }
}

var reducedWord = reduce(Int.word)
```

`reducedWord` and `flippedWord` are identical and are both `(Int) -> String?`. But if you pay attention to the declaration of the latter, you'll find that the brackets were already added as if it contains the result of the outer closure of `flip`. But `reduce` doesn't have an outer closure in the first place.

Merging higher-order functions

An interesting trick you can do with higher-order functions is to merge them. Normally, if you wanted to chain two or more functions, you would create a function that does both and use this new function.

Consider the following extension:

```
extension Int {
  func word() -> String? {
    let formatter = NumberFormatter()
    formatter.numberStyle = .spellOut
    return formatter.string(from: self as NSNumber)
  }

  func squared() -> Int {
    return self * self
  }
}
```

If you wanted to have one function that would do both, it would look like this:

```
func squareAndWord() -> String? {
  self.squared().word()
}
```

COPY

It's definitely not wrong. But you might have many more than just two functions that you want to bundle. There's a nice way to do that. Add this generic function merger:

```
func mergeFunctions<A, B, C>(
  _ f: @escaping (A) -> () -> B,
  _ g: @escaping (B) -> () -> C
) -> (A) -> C {
  return { a in
    let fValue = f(a)()
    return g(fValue)()
  }
}
```

COPY

This function is tailored for Swift-generated higher-order functions from methods that take one parameter. It's important to notice how the data types between parameters and return types relate. The return type of the first function matches the parameter of the second. If they don't match, then it wouldn't make sense to chain them. Try it out:

```
var mergedFunctions = mergeFunctions(Int.squared, Int.word)
mergedFunctions(2) // four
```

COPY

You might be wondering if there isn't a nicer way to use it. This almost looks like you **added** the two functions. How about doing the same but with operator overloading?

Add the following to your playground:

```
func +<A, B, C>(
  left: @escaping (A) -> () -> B,
  right: @escaping (B) -> () -> C
) -> (A) -> C {
  return { a in
    let leftValue = left(a)()
    return right(leftValue)()
  }
}
```

COPY

Now, the interesting part. Try it out:

```
var addedFunctions = Int.squared + Int.word
addedFunctions(2) // four
(Int.squared + Int.word)(2) // four
```

COPY

This is called function composition. You **composed** a function using two smaller functions. This is a bigger topic, but once you understand how you can play with higher-order functions, function composition becomes a lot clearer.

You're free to define the operator you want. But know that once you start having too many operators in your code, it will look unfamiliar to others. Your proprietary conventions will make it complicated for other eyes.

Key points

- A higher-order function is a function that deals with other functions, either as a parameter or as a return type.
- Swift allows the use of a closure or a function signature in a higher-order function, as long as the number of parameters and the return type are identical to the original higher-order function declaration.
- Using a function signature instead of a closure can simplify your code if the operation is complex or gets repeated across your code.
- `map`, `compactMap`, `flatMap`, `filter`, `reduce`, `sorted` and `split` all are examples of higher-order functions in the standard library.
- Higher-order functions also describe functions that return functions as return types.
- Function currying means breaking down a function that takes multiple parameters into a chain of functions that each takes one parameter.
- Currying and argument flipping are ways to alter a function's signature to fit a higher-order function.
- Each instance method can be used as a higher-order function through its containing type.
- Function composition is when you merge higher-order functions to create larger functions.
- You can use operator overloading to create an adding function for higher-order functions, making function composition easier.

Where to go from here?

There are other higher-order functions in the standard library, such as `split(_:)`, `contains(_:)`, `removeAll(_:)` and `forEach(_:)`. The intention

of this chapter is not to explain all the functions in the library, but to show how they can make your code shorter and simpler.

You can read about different algorithms in the book **"Data Structures & Algorithms in Swift"** or from the **swift-algorithms-club** on https://github.com/raywenderlich/swift-algorithm-club, which the book is based on.

Mark Complete

11. Functional Reactive Programming 9. Unsafe

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.

Have feedback to share about the online reading experience? If you have feedback about the UI, UX, highlighting, or other features of our online readers, you can send them to the design team with the form below:

Feedback about the UI, UX, or other features of the online reader? Leave them here!

Send Feedback