IOS TRAINING

PREPARED BY FARHAJ AHMED

LECTURE 5
COLLECTION TYPES PART 2

CLOSURES

Earlier, you learned about functions. But Swift has another object you can use to break up code into reusable chunks: a **closure**. They become particularly useful when dealing with collections.

A closure is simply a function with no name; you can assign it to a variable and pass it around like any other value. This chapter shows you how convenient and useful closures can be.

CLOSURES

Closure basics

Closures are so named because they have the ability to "close over" the variables and constants within the closure's own scope. This simply means that a closure can access, store and manipulate the value of any variable or constant from the surrounding context. Variables and constants used within the body of a closure are said to have been **captured** by the closure.

You may ask, "If closures are functions without names, then how do you use them?" To use a closure, you first have to assign it to a variable or constant.

Here's a declaration of a variable that can hold a closure:

```
var multiplyClosure: (Int, Int) -> Int
```

`multiplyClosure` takes two `Int` values and returns an `Int`. Notice that this is exactly the same as a variable declaration for a function. Like I said, a closure is simply a function without a name. The type of a closure is a function type.

In order for the declaration to compile in a playground, you need to provide an initial definition like so:

```
var multiplyClosure = { (a: Int, b: Int) -> Int in
  return a * b
}
```

This looks similar to a function declaration, but there's a subtle difference. There's the same parameter list, -> symbol and return type. But in the case of closures, these elements appear inside braces, and there is an `in` keyword after the return type.

With your closure variable defined, you can use it just as if it were a function, like so:

```
let result = multiplyClosure(4, 2)
```

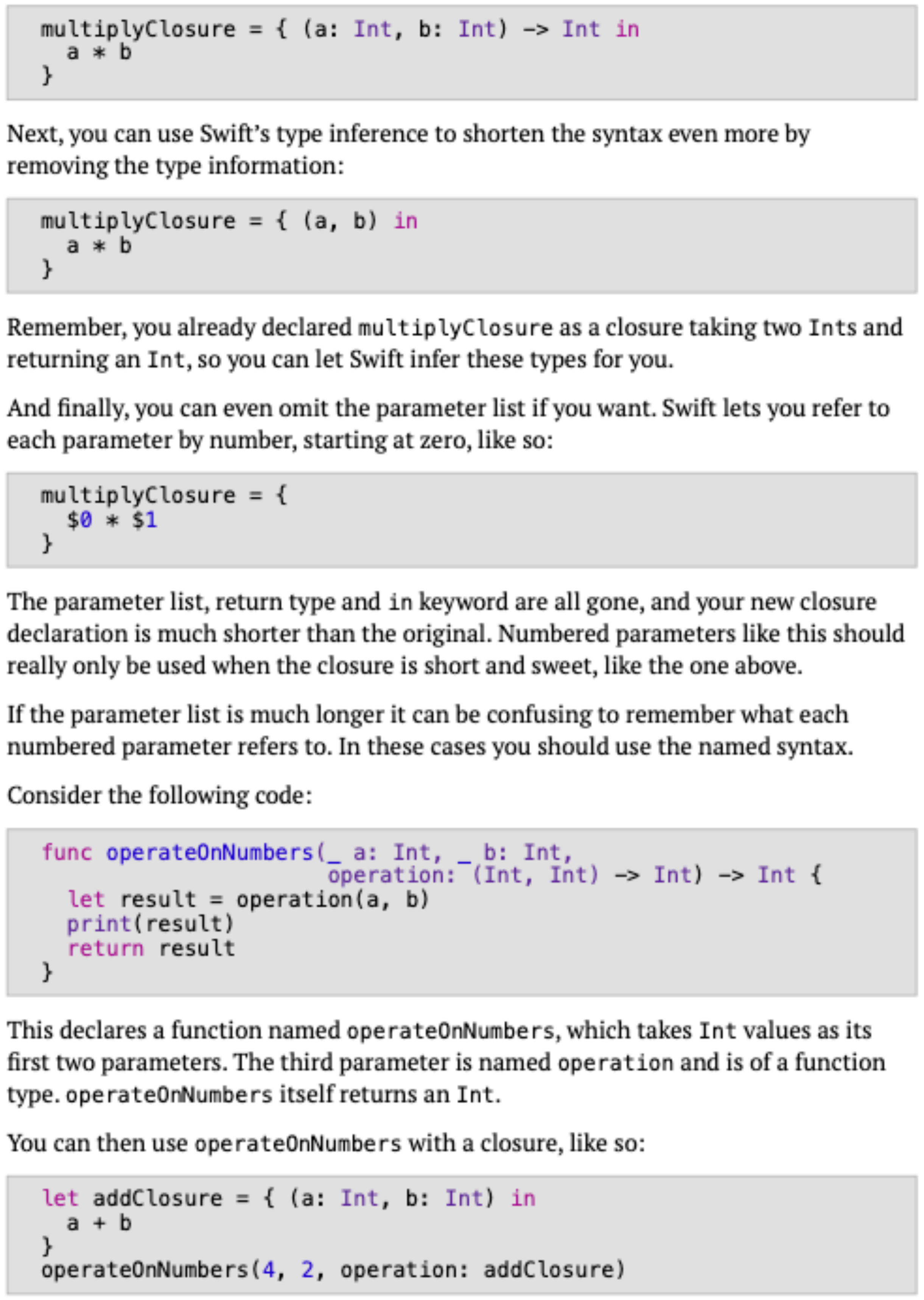
As you'd expect, `result` equals 8. Again, though, there's a subtle difference.

Notice how the closure has no external names for the parameters. You can't set them like you can with functions.

Shorthand syntax

Compared to functions, closures are designed to be lightweight. There are many ways to shorten their syntax. First, just like normal functions, if the closure consists of a single return statement, you can leave out the `return` keyword, like so:

CLOSURES



```
multiplyClosure = { (a: Int, b: Int) -> Int in
  a * b
}
```

Next, you can use Swift's type inference to shorten the syntax even more by removing the type information:

```
multiplyClosure = { (a, b) in
  a * b
}
```

Remember, you already declared `multiplyClosure` as a closure taking two `Int`s and returning an `Int`, so you can let Swift infer these types for you.

And finally, you can even omit the parameter list if you want. Swift lets you refer to each parameter by number, starting at zero, like so:

```
multiplyClosure = {
  $0 * $1
}
```

The parameter list, return type and `in` keyword are all gone, and your new closure declaration is much shorter than the original. Numbered parameters like this should really only be used when the closure is short and sweet, like the one above.

If the parameter list is much longer it can be confusing to remember what each numbered parameter refers to. In these cases you should use the named syntax.

Consider the following code:

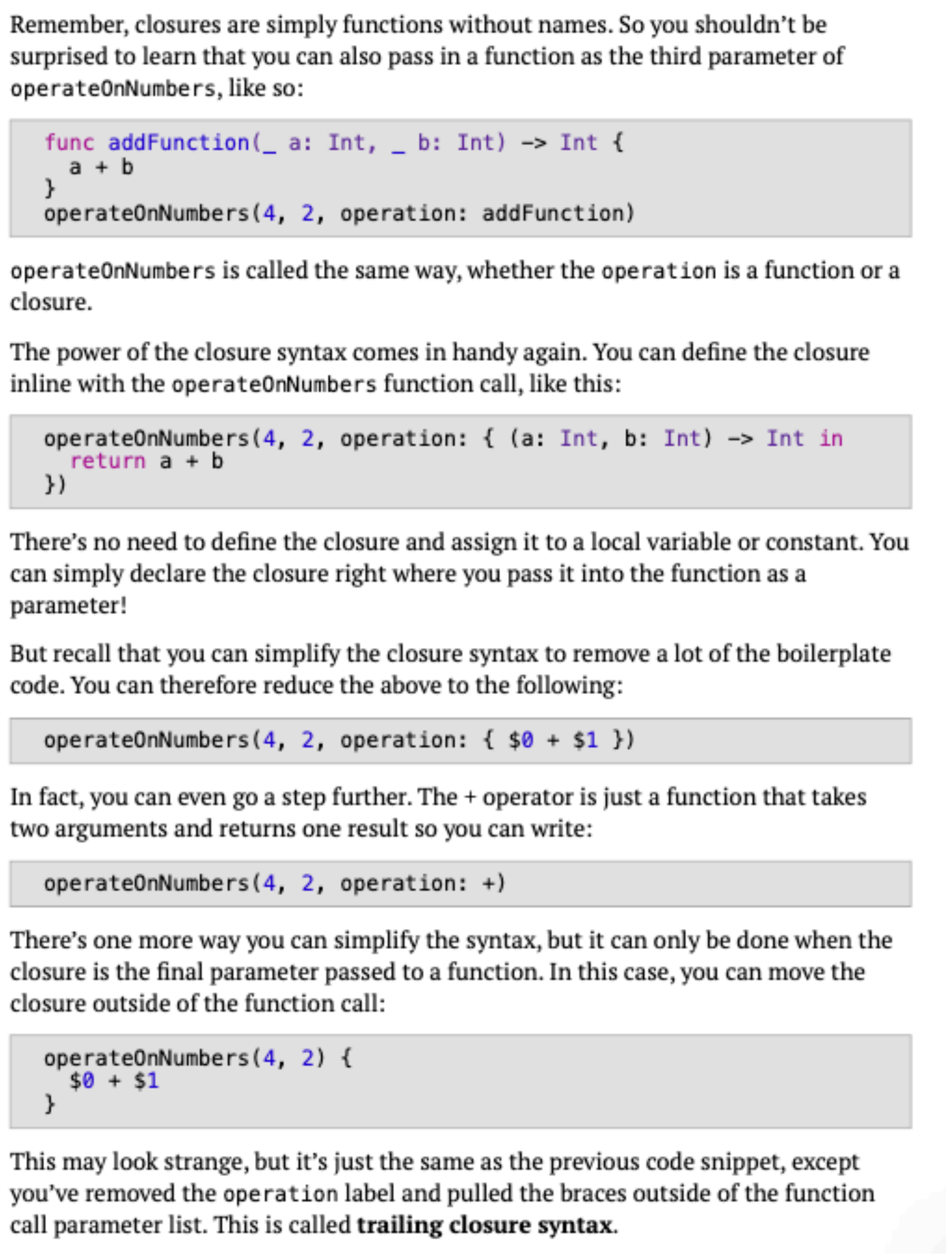
```
func operateOnNumbers(_ a: Int, _ b: Int,
                      operation: (Int, Int) -> Int) -> Int {
  let result = operation(a, b)
  print(result)
  return result
}
```

This declares a function named `operateOnNumbers`, which takes `Int` values as its first two parameters. The third parameter is named `operation` and is of a function type. `operateOnNumbers` itself returns an `Int`.

You can then use `operateOnNumbers` with a closure, like so:

```
let addClosure = { (a: Int, b: Int) in
  a + b
}
operateOnNumbers(4, 2, operation: addClosure)
```

CLOSURES



Remember, closures are simply functions without names. So you shouldn't be surprised to learn that you can also pass in a function as the third parameter of `operateOnNumbers`, like so:

```
func addFunction(_ a: Int, _ b: Int) -> Int {
  a + b
}
operateOnNumbers(4, 2, operation: addFunction)
```

`operateOnNumbers` is called the same way, whether the `operation` is a function or a closure.

The power of the closure syntax comes in handy again. You can define the closure inline with the `operateOnNumbers` function call, like this:

```
operateOnNumbers(4, 2, operation: { (a: Int, b: Int) -> Int in
  return a + b
})
```

There's no need to define the closure and assign it to a local variable or constant. You can simply declare the closure right where you pass it into the function as a parameter!

But recall that you can simplify the closure syntax to remove a lot of the boilerplate code. You can therefore reduce the above to the following:

```
operateOnNumbers(4, 2, operation: { $0 + $1 })
```

In fact, you can even go a step further. The + operator is just a function that takes two arguments and returns one result so you can write:

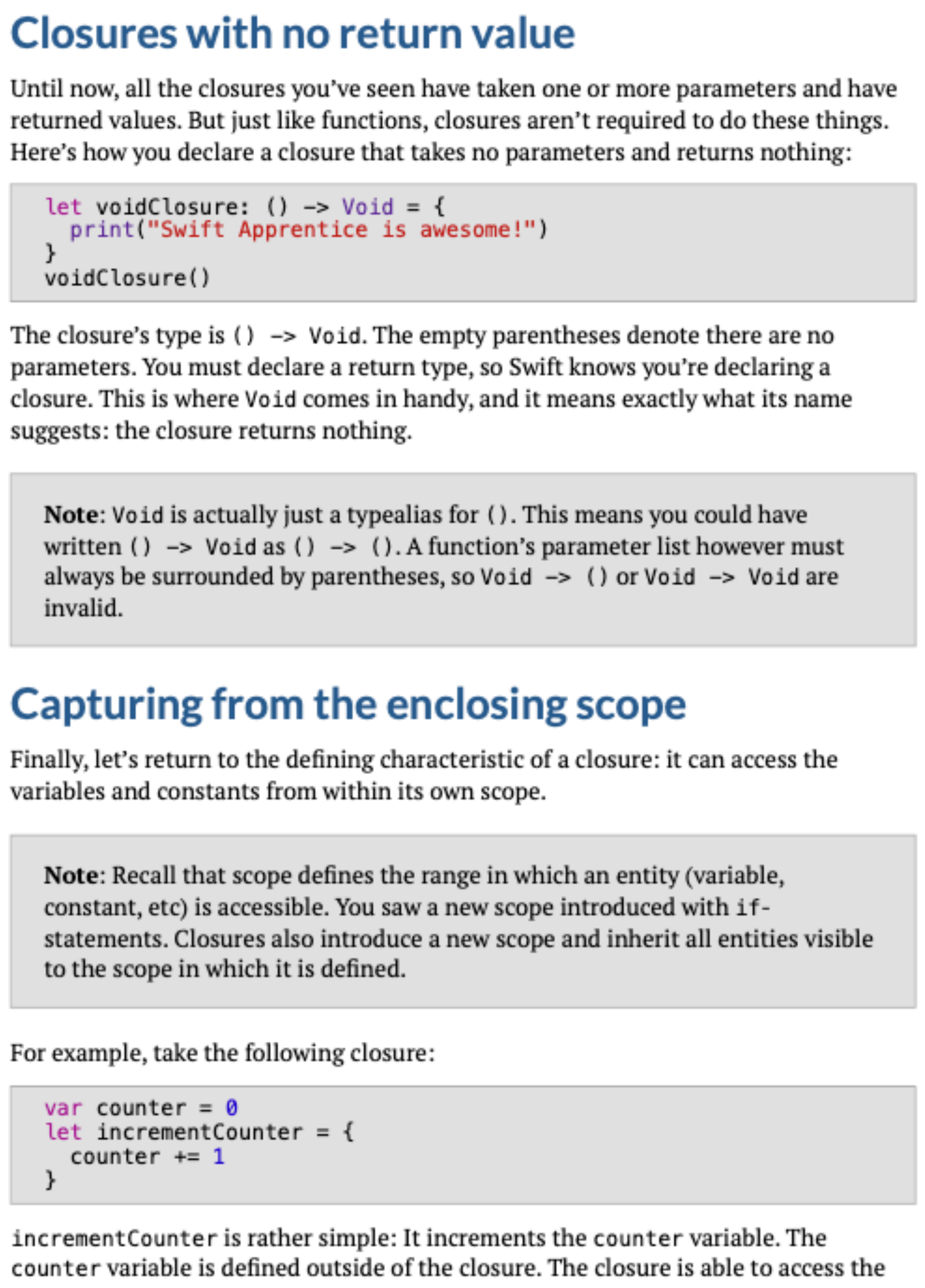
```
operateOnNumbers(4, 2, operation: +)
```

There's one more way you can simplify the syntax, but it can only be done when the closure is the final parameter passed to a function. In this case, you can move the closure outside of the function call:

```
operateOnNumbers(4, 2) {
  $0 + $1
}
```

This may look strange, but it's just the same as the previous code snippet, except you've removed the `operation` label and pulled the braces outside of the function call parameter list. This is called **trailing closure syntax**.

CLOSURES



Closures with no return value

Until now, all the closures you've seen have taken one or more parameters and have returned values. But just like functions, closures aren't required to do these things. Here's how you declare a closure that takes no parameters and returns nothing:

```
let voidClosure: () -> Void = {
  print("Swift Apprentice is awesome!")
}
voidClosure()
```

The closure's type is `() -> Void`. The empty parentheses denote there are no parameters. You must declare a return type, so Swift knows you're declaring a closure. This is where `Void` comes in handy, and it means exactly what its name suggests: the closure returns nothing.

> **Note**: `Void` is actually just a typealias for `()`. This means you could have written `() -> Void` as `() -> ()`. A function's parameter list however must always be surrounded by parentheses, so `Void -> ()` or `Void -> Void` are invalid.

Capturing from the enclosing scope

Finally, let's return to the defining characteristic of a closure: it can access the variables and constants from within its own scope.

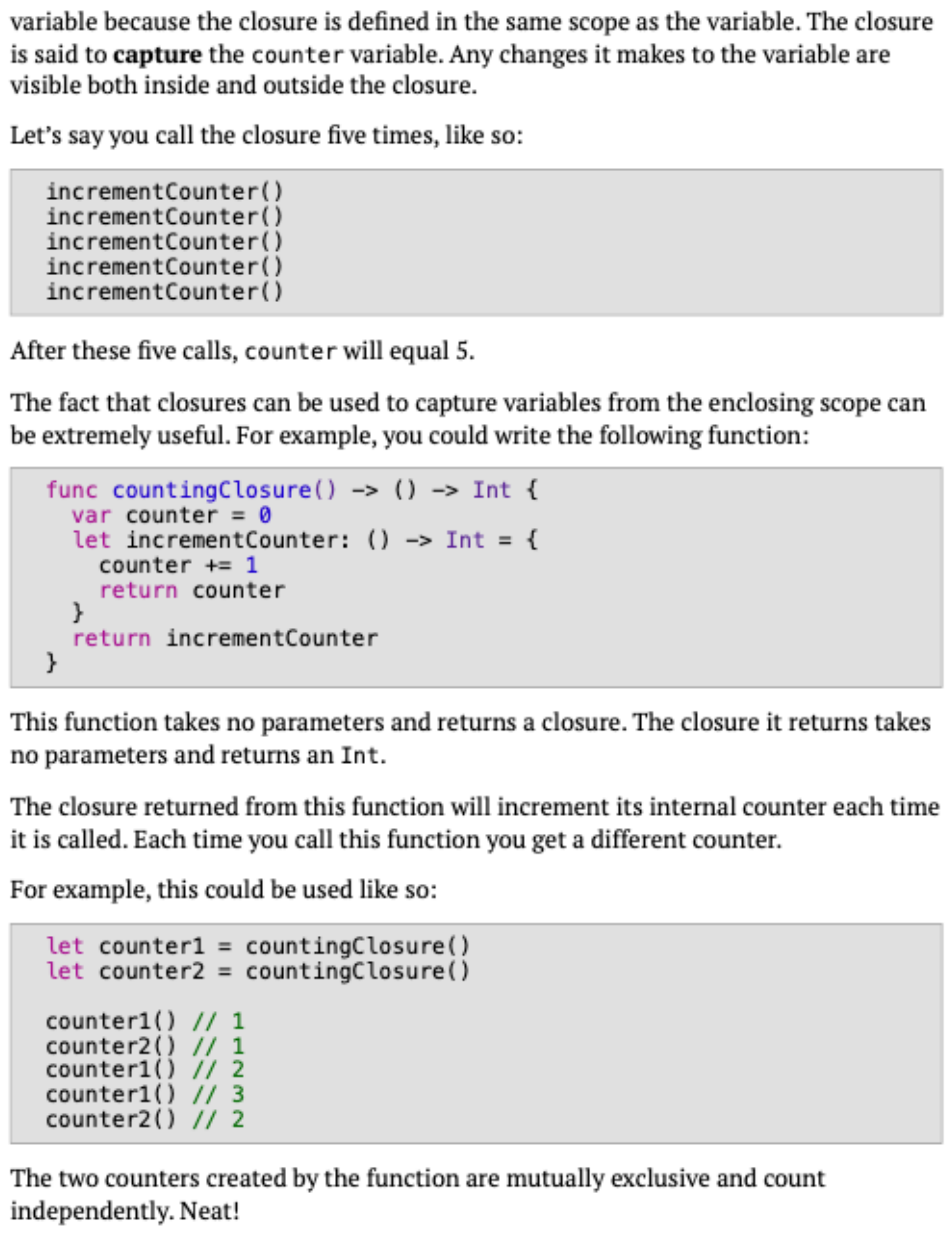
> **Note**: Recall that scope defines the range in which an entity (variable, constant, etc) is accessible. You saw a new scope introduced with `if`-statements. Closures also introduce a new scope and inherit all entities visible to the scope in which it is defined.

For example, take the following closure:

```
var counter = 0
let incrementCounter = {
  counter += 1
}
```

`incrementCounter` is rather simple: It increments the `counter` variable. The `counter` variable is defined outside of the closure. The closure is able to access the

CLOSURES



variable because the closure is defined in the same scope as the variable. The closure is said to **capture** the `counter` variable. Any changes it makes to the variable are visible both inside and outside the closure.

Let's say you call the closure five times, like so:

```
incrementCounter()
incrementCounter()
incrementCounter()
incrementCounter()
incrementCounter()
```

After these five calls, `counter` will equal 5.

The fact that closures can be used to capture variables from the enclosing scope can be extremely useful. For example, you could write the following function:

```
func countingClosure() -> () -> Int {
  var counter = 0
  let incrementCounter: () -> Int = {
    counter += 1
    return counter
  }
  return incrementCounter
}
```

This function takes no parameters and returns a closure. The closure it returns takes no parameters and returns an `Int`.

The closure returned from this function will increment its internal counter each time it is called. Each time you call this function you get a different counter.

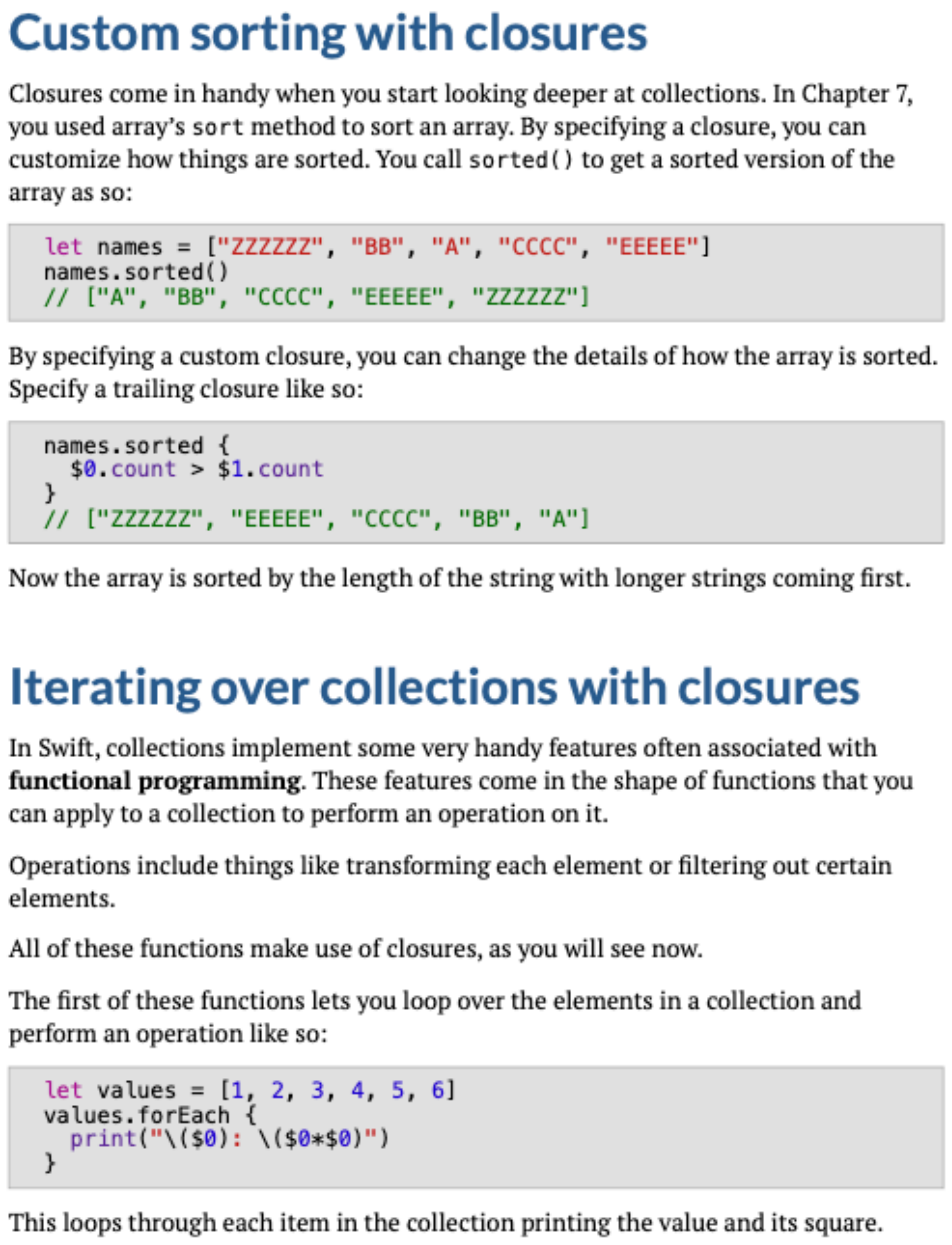
For example, this could be used like so:

```
let counter1 = countingClosure()
let counter2 = countingClosure()

counter1() // 1
counter2() // 1
counter1() // 2
counter1() // 3
counter2() // 2
```

The two counters created by the function are mutually exclusive and count independently. Neat!

CLOSURES



Custom sorting with closures

Closures come in handy when you start looking deeper at collections. In Chapter 7, you used array's `sort` method to sort an array. By specifying a closure, you can customize how things are sorted. You call `sorted()` to get a sorted version of the array as so:

```
let names = ["ZZZZZZ", "BB", "A", "CCCC", "EEEEE"]
names.sorted()
// ["A", "BB", "CCCC", "EEEEE", "ZZZZZZ"]
```

By specifying a custom closure, you can change the details of how the array is sorted. Specify a trailing closure like so:

```
names.sorted {
  $0.count > $1.count
}
// ["ZZZZZZ", "EEEEE", "CCCC", "BB", "A"]
```

Now the array is sorted by the length of the string with longer strings coming first.

Iterating over collections with closures

In Swift, collections implement some very handy features often associated with **functional programming**. These features come in the shape of functions that you can apply to a collection to perform an operation on it.

Operations include things like transforming each element or filtering out certain elements.

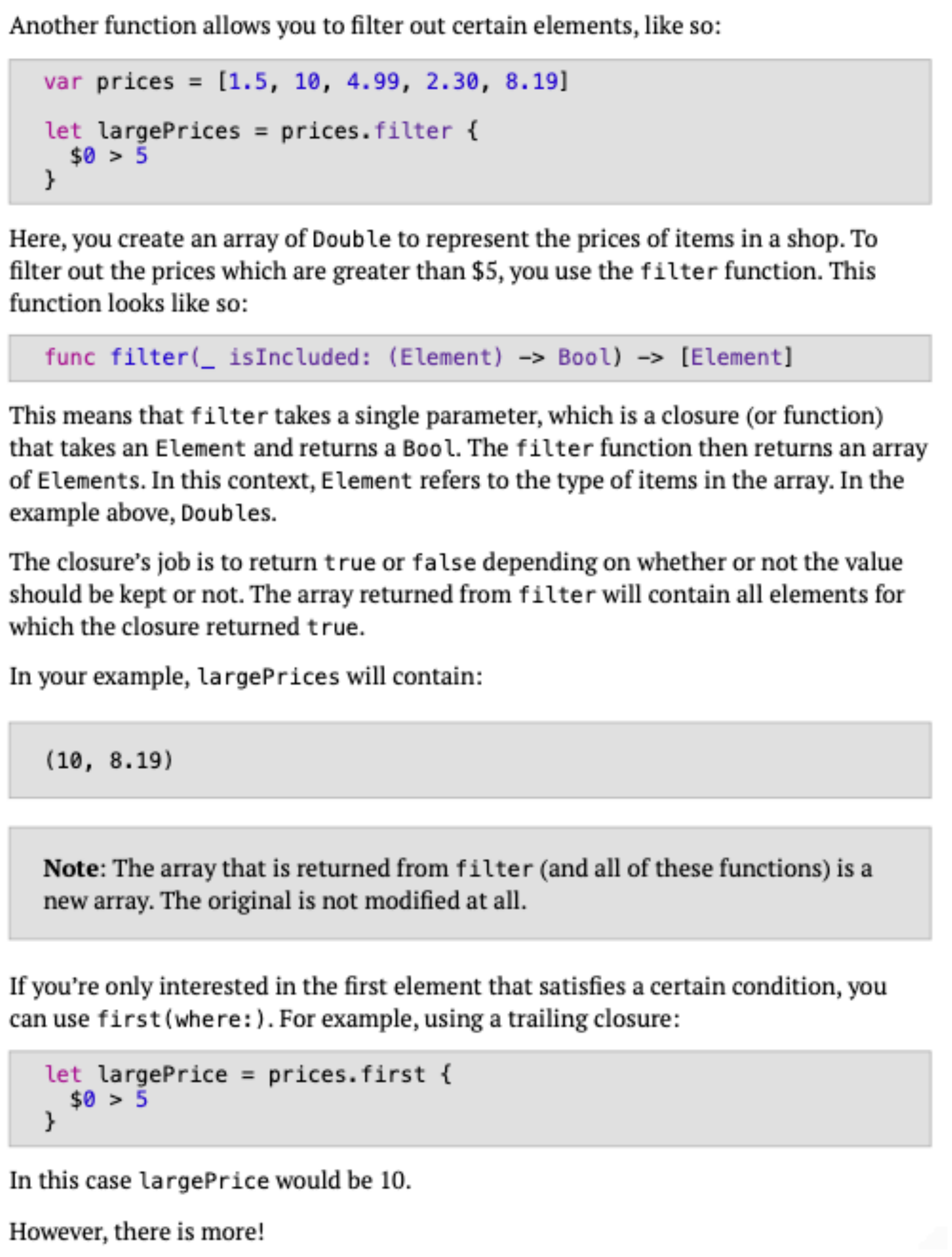
All of these functions make use of closures, as you will see now.

The first of these functions lets you loop over the elements in a collection and perform an operation like so:

```
let values = [1, 2, 3, 4, 5, 6]
values.forEach {
  print("\($0): \($0*$0)")
}
```

This loops through each item in the collection printing the value and its square.

CLOSURES



Another function allows you to filter out certain elements, like so:

```
var prices = [1.5, 10, 4.99, 2.30, 8.19]

let largePrices = prices.filter {
  $0 > 5
}
```

Here, you create an array of `Double` to represent the prices of items in a shop. To filter out the prices which are greater than $5, you use the `filter` function. This function looks like so:

```
func filter(_ isIncluded: (Element) -> Bool) -> [Element]
```

This means that `filter` takes a single parameter, which is a closure (or function) that takes an `Element` and returns a `Bool`. The `filter` function then returns an array of `Element`s. In this context, `Element` refers to the type of items in the array. In the example above, `Double`s.

The closure's job is to return `true` or `false` depending on whether or not the value should be kept or not. The array returned from `filter` will contain all elements for which the closure returned `true`.

In your example, `largePrices` will contain:

```
(10, 8.19)
```

Note: The array that is returned from `filter` (and all of these functions) is a new array. The original is not modified at all.

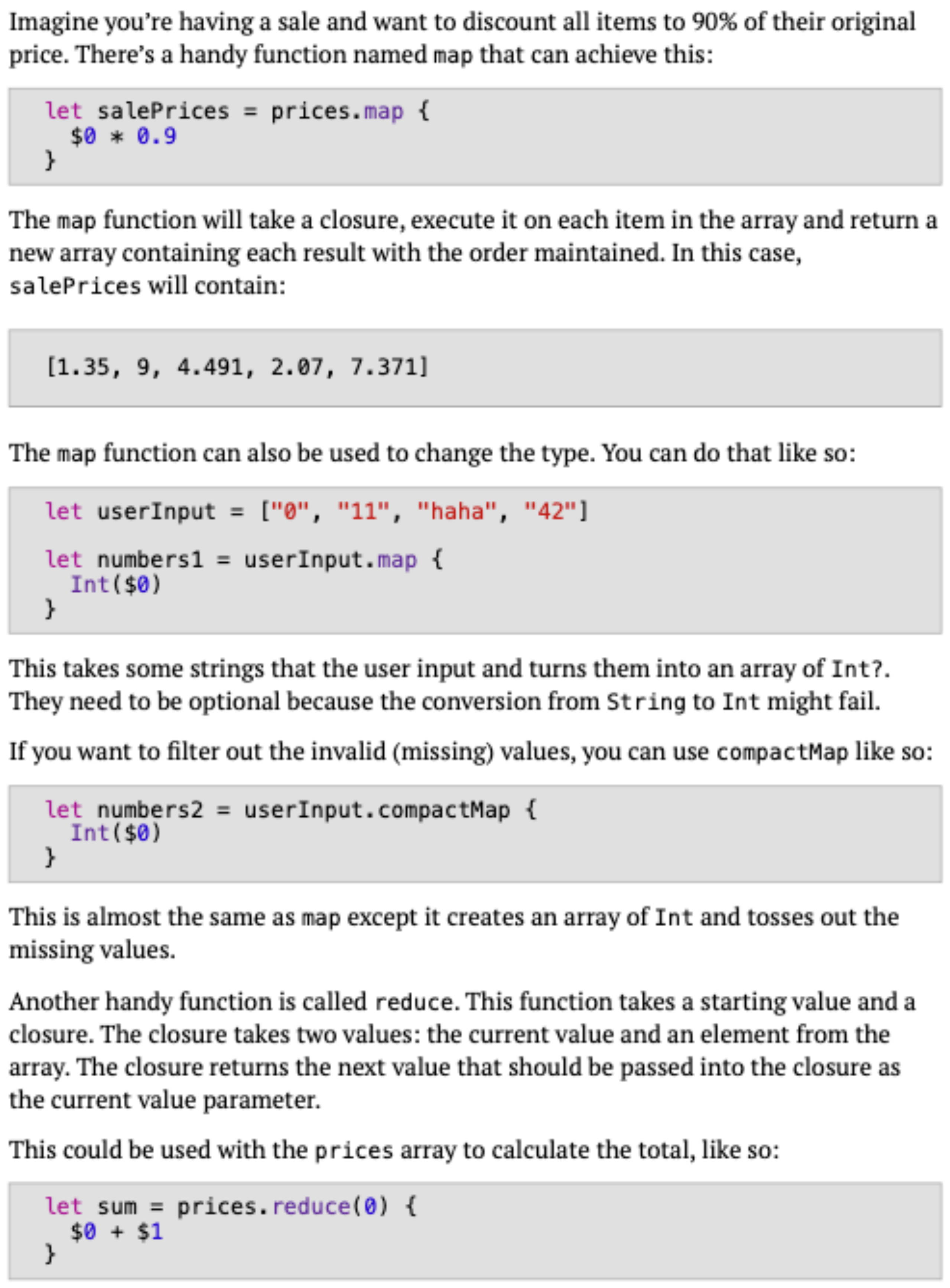
If you're only interested in the first element that satisfies a certain condition, you can use `first(where:)`. For example, using a trailing closure:

```
let largePrice = prices.first {
  $0 > 5
}
```

In this case `largePrice` would be 10.

However, there is more!

CLOSURES



Imagine you're having a sale and want to discount all items to 90% of their original price. There's a handy function named map that can achieve this:

```
let salePrices = prices.map {
  $0 * 0.9
}
```

The map function will take a closure, execute it on each item in the array and return a new array containing each result with the order maintained. In this case, salePrices will contain:

```
[1.35, 9, 4.491, 2.07, 7.371]
```

The map function can also be used to change the type. You can do that like so:

```
let userInput = ["0", "11", "haha", "42"]

let numbers1 = userInput.map {
  Int($0)
}
```

This takes some strings that the user input and turns them into an array of Int?. They need to be optional because the conversion from String to Int might fail.

If you want to filter out the invalid (missing) values, you can use compactMap like so:

```
let numbers2 = userInput.compactMap {
  Int($0)
}
```

This is almost the same as map except it creates an array of Int and tosses out the missing values.

Another handy function is called reduce. This function takes a starting value and a closure. The closure takes two values: the current value and an element from the array. The closure returns the next value that should be passed into the closure as the current value parameter.

This could be used with the prices array to calculate the total, like so:

```
let sum = prices.reduce(0) {
  $0 + $1
}
```

CLOSURES

The initial value is 0. Then the closure calculates the sum of the current value plus the current iteration's value. Thus you calculate the total of all the values in the array. In this case, `sum` will be:

```
26.98
```

Now that you've seen `filter`, `map` and `reduce`, hopefully you're realizing how powerful these functions can be, thanks to the syntax of closures. In just a few lines of code, you have calculated quite complex values from the collection.

These functions can also be used with dictionaries. Imagine you represent the stock in your shop by a dictionary mapping the price to number of items at that price. You could use that to calculate the total value of your stock like so:

```
let stock = [1.5: 5, 10: 2, 4.99: 20, 2.30: 5, 8.19: 30]
let stockSum = stock.reduce(0) {
  $0 + $1.key * Double($1.value)
}
```

In this case, the second parameter to the `reduce` function is a named tuple containing the `key` and `value` from the dictionary elements. A type conversion of the value is required to perform the calculation.

Here, the result is:

```
384.5
```

There's another form of `reduce` named `reduce(into:_:)`. You'd use it when the result you're reducing a collection into is an array or dictionary, like so:

```
let farmAnimals = ["🐂": 5, "🐄": 10, "🐑": 50, "🐶": 1]
let allAnimals = farmAnimals.reduce(into: []) {
  (result, this: (key: String, value: Int)) in
  for _ in 0 ..< this.value {
    result.append(this.key)
  }
}
```

It works in exactly the same way as the other version, except that you don't return something from the closure. Instead, each iteration gives you a mutable value. In this way, there is only ever one array in this example that is created and appended to,

CLOSURES

The initial value is 0. Then the closure calculates the sum of the current value plus the current iteration's value. Thus you calculate the total of all the values in the array. In this case, `sum` will be:

```
26.98
```

Now that you've seen `filter`, `map` and `reduce`, hopefully you're realizing how powerful these functions can be, thanks to the syntax of closures. In just a few lines of code, you have calculated quite complex values from the collection.

These functions can also be used with dictionaries. Imagine you represent the stock in your shop by a dictionary mapping the price to number of items at that price. You could use that to calculate the total value of your stock like so:

```
let stock = [1.5: 5, 10: 2, 4.99: 20, 2.30: 5, 8.19: 30]
let stockSum = stock.reduce(0) {
  $0 + $1.key * Double($1.value)
}
```

In this case, the second parameter to the `reduce` function is a named tuple containing the `key` and `value` from the dictionary elements. A type conversion of the value is required to perform the calculation.

Here, the result is:

```
384.5
```

There's another form of `reduce` named `reduce(into:_:)`. You'd use it when the result you're reducing a collection into is an array or dictionary, like so:

```
let farmAnimals = ["🐎": 5, "🐄": 10, "🐑": 50, "🐶": 1]
let allAnimals = farmAnimals.reduce(into: []) {
  (result, this: (key: String, value: Int)) in
  for _ in 0 ..< this.value {
    result.append(this.key)
  }
}
```

It works in exactly the same way as the other version, except that you don't return something from the closure. Instead, each iteration gives you a mutable value. In this way, there is only ever one array in this example that is created and appended to,

CLOSURES

making `reduce(into:_:)` more efficient in some cases.

Should you need to chop up an array, there are a few more functions that can be helpful. The first function is `dropFirst`, which works like so:

```
let removeFirst = prices.dropFirst()
let removeFirstTwo = prices.dropFirst(2)
```

The `dropFirst` function takes a single parameter that defaults to 1 and returns an array with the required number of elements removed from the front. Results are as follows:

```
removeFirst = [10, 4.99, 2.30, 8.19]
removeFirstTwo = [4.99, 2.30, 8.19]
```

Just like `dropFirst`, there also exists `dropLast` which removes elements from the end of the array. It works like this:

```
let removeLast = prices.dropLast()
let removeLastTwo = prices.dropLast(2)
```

The results of these are as you would expect:

```
removeLast = [1.5, 10, 4.99, 2.30]
removeLastTwo = [1.5, 10, 4.99]
```

You can select just the first or last elements of an array as shown below:

```
let firstTwo = prices.prefix(2)
let lastTwo = prices.suffix(2)
```

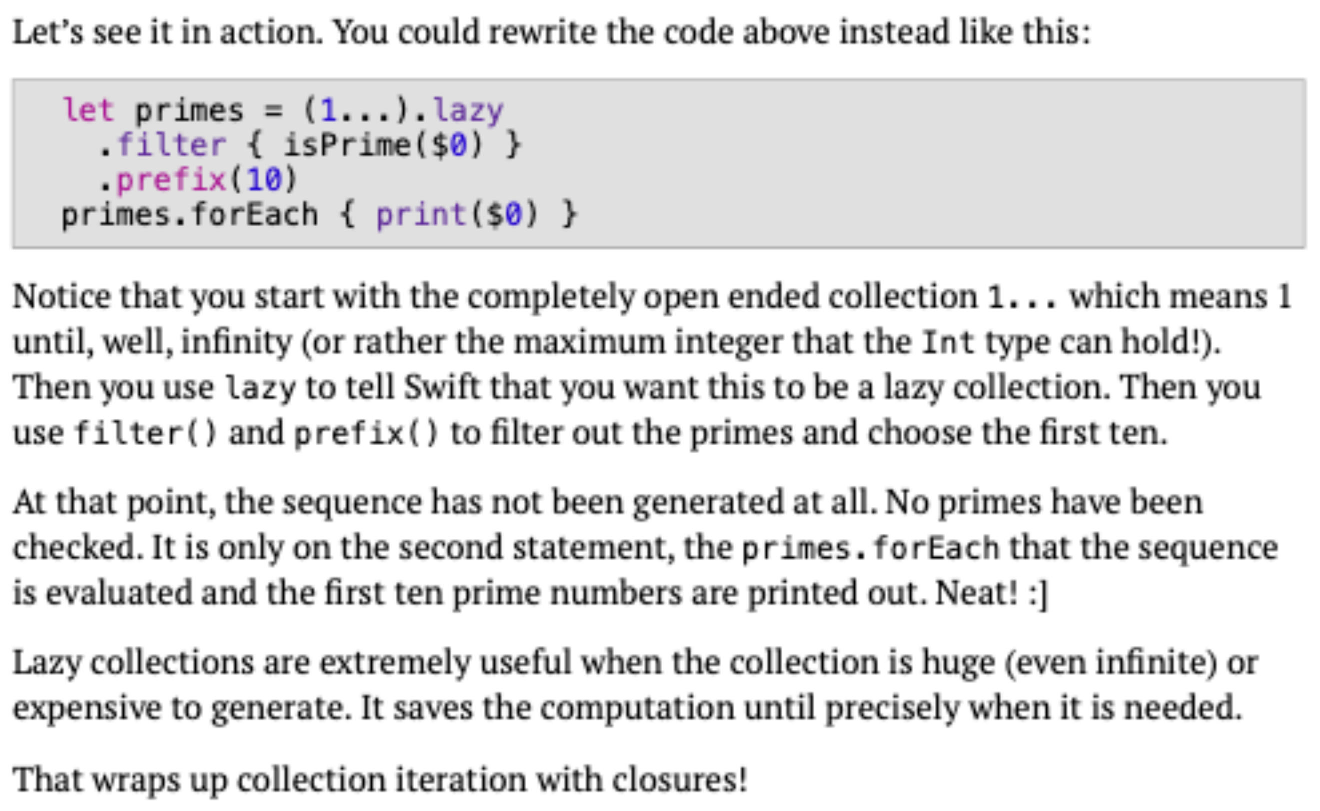
Here, `prefix` returns the required number of elements from the front of the array, and `suffix` returns the required number of elements from the back of the array. The results of this function are:

```
firstTwo = [1.5, 10]
lastTwo = [2.30, 8.19]
```

And finally, you can remove all elements in a collection by using `removeAll()` qualified by a closure, or unconditionally:

```
prices.removeAll() { $0 > 2 } // prices is now [1.5]
prices.removeAll() // prices is now an empty array
```

CLOSURES



Let’s see it in action. You could rewrite the code above instead like this:

```
  let primes = (1...).lazy
    .filter { isPrime($0) }
    .prefix(10)
  primes.forEach { print($0) }
```

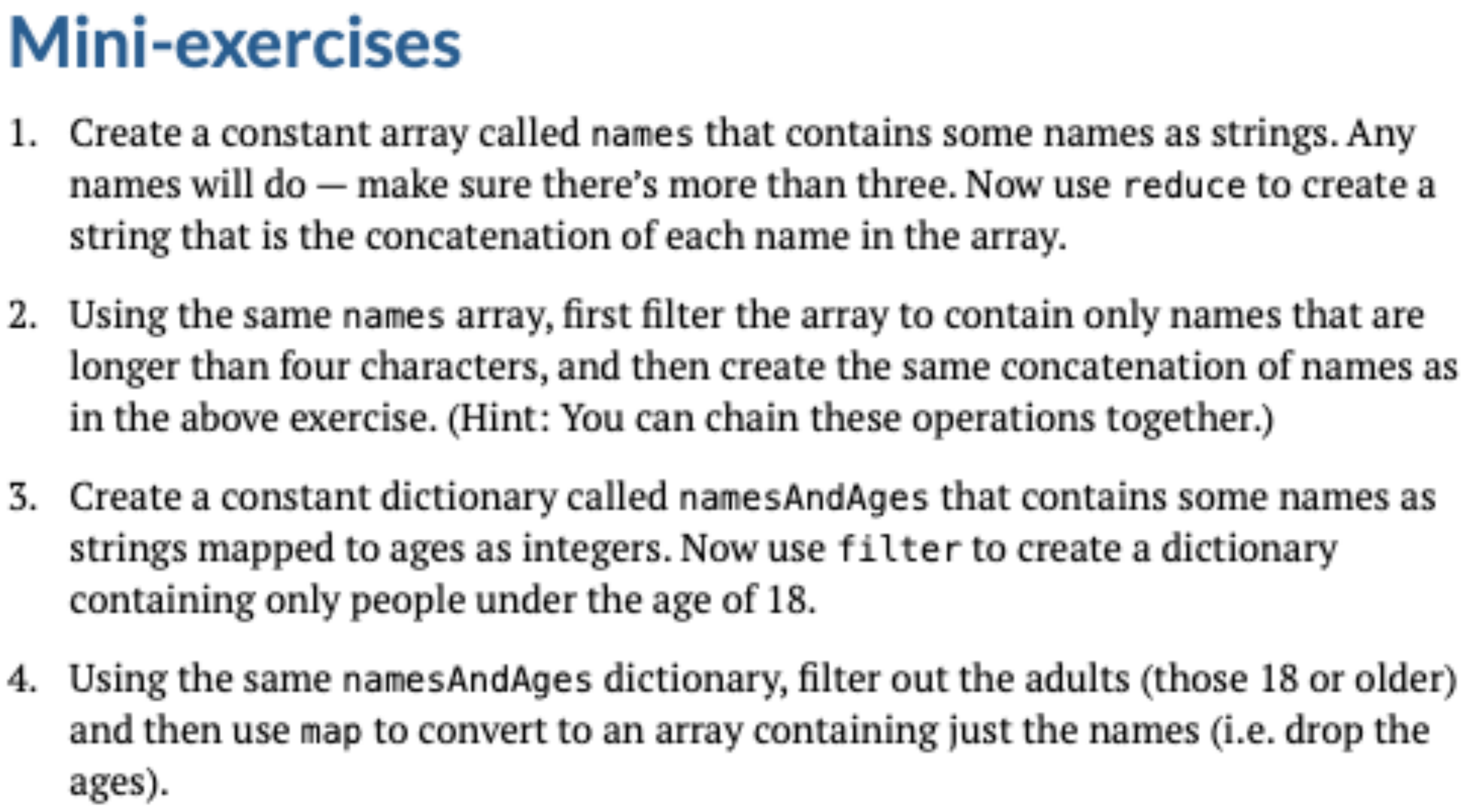
Notice that you start with the completely open ended collection `1...` which means 1 until, well, infinity (or rather the maximum integer that the `Int` type can hold!). Then you use `lazy` to tell Swift that you want this to be a lazy collection. Then you use `filter()` and `prefix()` to filter out the primes and choose the first ten.

At that point, the sequence has not been generated at all. No primes have been checked. It is only on the second statement, the `primes.forEach` that the sequence is evaluated and the first ten prime numbers are printed out. Neat! :]

Lazy collections are extremely useful when the collection is huge (even infinite) or expensive to generate. It saves the computation until precisely when it is needed.

That wraps up collection iteration with closures!

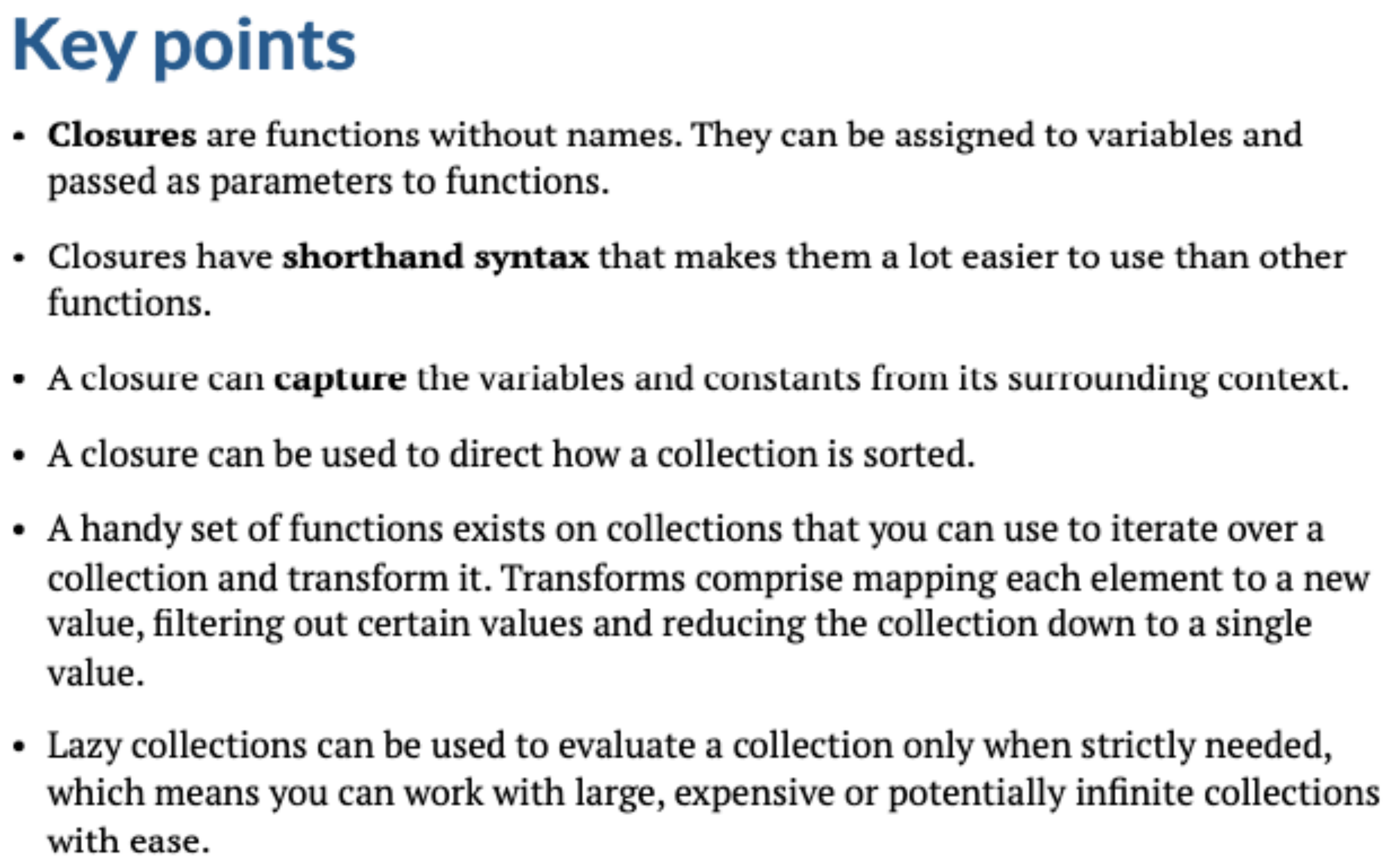
CLOSURES



Mini-exercises

1. Create a constant array called `names` that contains some names as strings. Any names will do — make sure there's more than three. Now use `reduce` to create a string that is the concatenation of each name in the array.
2. Using the same `names` array, first filter the array to contain only names that are longer than four characters, and then create the same concatenation of names as in the above exercise. (Hint: You can chain these operations together.)
3. Create a constant dictionary called `namesAndAges` that contains some names as strings mapped to ages as integers. Now use `filter` to create a dictionary containing only people under the age of 18.
4. Using the same `namesAndAges` dictionary, filter out the adults (those 18 or older) and then use `map` to convert to an array containing just the names (i.e. drop the ages).

CLOSURES



Key points

- **Closures** are functions without names. They can be assigned to variables and passed as parameters to functions.
- Closures have **shorthand syntax** that makes them a lot easier to use than other functions.
- A closure can **capture** the variables and constants from its surrounding context.
- A closure can be used to direct how a collection is sorted.
- A handy set of functions exists on collections that you can use to iterate over a collection and transform it. Transforms comprise mapping each element to a new value, filtering out certain values and reducing the collection down to a single value.
- Lazy collections can be used to evaluate a collection only when strictly needed, which means you can work with large, expensive or potentially infinite collections with ease.

FEEDBACK TIME

http://bit.ly/iOSFeedback

THANK YOU