Closures | Documentation

Group code that executes together, without creating a named function.

Closures are self-contained blocks of functionality that can be passed around and used in your code. Closures in Swift are similar to blocks in C and Objective-C and to lambdas in other programming languages.

Closures can capture and store references to any constants and variables from the context in which they're defined. This is known as *closing over* those constants and variables. Swift handles all of the memory management of capturing for you.

Global and nested functions, as introduced in Functions, are actually special cases of closures. Closures take one of three forms:

- Global functions are closures that have a name and don't capture any values.
- Nested functions are closures that have a name and can capture values from their enclosing function.
- Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context.

Swift's closure expressions have a clean, clear style, with optimizations that encourage brief, clutter-free syntax in common scenarios. These optimizations include:

- Inferring parameter and return value types from context
- Implicit returns from single-expression closures
- Shorthand argument names
- Trailing closure syntax

Closure Expressions

Nested functions, as introduced in Nested Functions, are a convenient means of naming and defining self-contained blocks of code as part of a larger function. However, it's sometimes useful to write shorter versions of function-like constructs without a full declaration and name. This is particularly true when you work with functions or methods that take functions as one or more of their arguments.

Closure expressions are a way to write inline closures in a brief, focused syntax. Closure expressions provide several syntax optimizations for writing closures in a shortened form without loss of clarity or intent. The closure expression examples below illustrate these optimizations by refining a single example of the `sorted(by:)` method over several iterations, each of which expresses the same functionality in a more succinct way.

The Sorted Method

Swift's standard library provides a method called `sorted(by:)`, which sorts an array of values of a known type, based on the output of a sorting closure that you provide. Once it completes the sorting process, the `sorted(by:)` method returns a new array of the same type and size as the old one, with its elements in the correct sorted order. The original array isn't modified by the `sorted(by:)` method.

The closure expression examples below use the `sorted(by:)` method to sort an array of `String` values in reverse alphabetical order. Here's the initial array to be sorted:

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

The `sorted(by:)` method accepts a closure that takes two arguments of the same type as the array's contents, and returns a `Bool` value to say whether the first value should appear before or after the second value once the values are sorted. The sorting closure needs to return `true` if the first value should appear *before* the second value, and `false` otherwise.

This example is sorting an array of `String` values, and so the sorting closure needs to be a function of type `(String, String) -> Bool`.

One way to provide the sorting closure is to write a normal function of the correct type, and to pass it in as an argument to the `sorted(by:)` method:

```
func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
var reversedNames = names.sorted(by: backward)
// reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

If the first string (`s1`) is greater than the second string (`s2`), the `backward(_:_:)` function will return `true`, indicating that `s1` should appear before `s2` in the sorted array. For characters in strings, "greater than" means "appears later in the alphabet than". This means that the letter `"B"` is "greater than" the letter

`"A"`, and the string `"Tom"` is greater than the string `"Tim"`. This gives a reverse alphabetical sort, with `"Barry"` being placed before `"Alex"`, and so on.

However, this is a rather long-winded way to write what is essentially a single-expression function (`a > b`). In this example, it would be preferable to write the sorting closure inline, using closure expression syntax.

Closure Expression Syntax

Closure expression syntax has the following general form:

```
{ (<#parameters#>) -> <#return type#> in
   <#statements#>
}
```

The *parameters* in closure expression syntax can be in-out parameters, but they can't have a default value. Variadic parameters can be used if you name the variadic parameter. Tuples can also be used as parameter types and return types.

The example below shows a closure expression version of the `backward(_:_:)` function from above:

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

Note that the declaration of parameters and return type for this inline closure is identical to the declaration from the `backward(_:_:)` function. In both cases, it's written as `(s1: String, s2: String) -> Bool`. However, for the inline closure expression, the parameters and return type are written *inside* the curly braces, not outside of them.

The start of the closure's body is introduced by the `in` keyword. This keyword indicates that the definition of the closure's parameters and return type has finished, and the body of the closure is about to begin.

Because the body of the closure is so short, it can even be written on a single line:

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return
```

```
s1 > s2 } )
```

This illustrates that the overall call to the `sorted(by:)` method has remained the same. A pair of parentheses still wrap the entire argument for the method. However, that argument is now an inline closure.

Inferring Type From Context

Because the sorting closure is passed as an argument to a method, Swift can infer the types of its parameters and the type of the value it returns. The `sorted(by:)` method is being called on an array of strings, so its argument must be a function of type `(String, String) -> Bool`. This means that the `(String, String)` and `Bool` types don't need to be written as part of the closure expression's definition. Because all of the types can be inferred, the return arrow (`->`) and the parentheses around the names of the parameters can also be omitted:

```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

It's always possible to infer the parameter types and return type when passing a closure to a function or method as an inline closure expression. As a result, you never need to write an inline closure in its fullest form when the closure is used as a function or method argument.

Nonetheless, you can still make the types explicit if you wish, and doing so is encouraged if it avoids ambiguity for readers of your code. In the case of the `sorted(by:)` method, the purpose of the closure is clear from the fact that sorting is taking place, and it's safe for a reader to assume that the closure is likely to be working with `String` values, because it's assisting with the sorting of an array of strings.

Implicit Returns from Single-Expression Closures

Single-expression closures can implicitly return the result of their single expression by omitting the `return` keyword from their declaration, as in this version of the previous example:

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```

Here, the function type of the `sorted(by:)` method's argument makes it clear that a `Bool` value must be returned by the closure. Because the closure's body contains a single expression (`s1 > s2`) that returns a `Bool` value, there's no ambiguity, and the `return` keyword can be omitted.

Shorthand Argument Names

Swift automatically provides shorthand argument names to inline closures, which can be used to refer to the values of the closure's arguments by the names `$0`, `$1`, `$2`, and so on.

If you use these shorthand argument names within your closure expression, you can omit the closure's argument list from its definition. The type of the shorthand argument names is inferred from the expected function type, and the highest numbered shorthand argument you use determines the number of arguments that the closure takes. The `in` keyword can also be omitted, because the closure expression is made up entirely of its body:

```
reversedNames = names.sorted(by: { $0 > $1 } )
```

Here, `$0` and `$1` refer to the closure's first and second `String` arguments. Because `$1` is the shorthand argument with highest number, the closure is understood to take two arguments. Because the `sorted(by:)` function here expects a closure whose arguments are both strings, the shorthand arguments `$0` and `$1` are both of type `String`.

Operator Methods

There's actually an even *shorter* way to write the closure expression above. Swift's `String` type defines its string-specific implementation of the greater-than operator (`>`) as a method that has two parameters of type `String`, and returns a value of type `Bool`. This exactly matches the method type needed by the `sorted(by:)` method. Therefore, you can simply pass in the greater-than operator, and Swift will infer that you want to use its string-specific implementation:

```
reversedNames = names.sorted(by: >)
```

For more about operator methods, see Operator Methods.

Trailing Closures

If you need to pass a closure expression to a function as the function's final argument and the closure expression is long, it can be useful to write it as a *trailing closure* instead. You write a trailing closure after the function call's parentheses, even though the trailing closure is still an argument to the function. When you use the trailing closure syntax, you don't write the argument label for the first closure as part of the

function call. A function call can include multiple trailing closures; however, the first few examples below use a single trailing closure.

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // function body goes here
}

// Here's how you call this function without using a trailing closure:

someFunctionThatTakesAClosure(closure: {
    // closure's body goes here
})

// Here's how you call this function with a trailing closure instead:

someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

The string-sorting closure from the Closure Expression Syntax section above can be written outside of the `sorted(by:)` method's parentheses as a trailing closure:

```
reversedNames = names.sorted() { $0 > $1 }
```

If a closure expression is provided as the function's or method's only argument and you provide that expression as a trailing closure, you don't need to write a pair of parentheses `()` after the function or method's name when you call the function:

```
reversedNames = names.sorted { $0 > $1 }
```

Trailing closures are most useful when the closure is sufficiently long that it isn't possible to write it inline on a single line. As an example, Swift's `Array` type has a `map(_:)` method, which takes a closure expression as its single argument. The closure is called once for each item in the array, and returns an alternative mapped value (possibly of some other type) for that item. You specify the nature of the mapping and the type of the returned value by writing code in the closure that you pass to `map(_:)`.

After applying the provided closure to each array element, the `map(_:)` method returns a new array containing all of the new mapped values, in the same order as their corresponding values in the original array.

Here's how you can use the `map(_:)` method with a trailing closure to convert an array of `Int` values into an array of `String` values. The array `[16, 58, 510]` is used to create the new array `["OneSix", "FiveEight", "FiveOneZero"]`:

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

The code above creates a dictionary of mappings between the integer digits and English-language versions of their names. It also defines an array of integers, ready to be converted into strings.

You can now use the `numbers` array to create an array of `String` values, by passing a closure expression to the array's `map(_:)` method as a trailing closure:

```
let strings = numbers.map { (number) -> String in
    var number = number
    var output = ""
    repeat {
        output = digitNames[number % 10]! + output
        number /= 10
    } while number > 0
    return output
}
// strings is inferred to be of type [String]
// its value is ["OneSix", "FiveEight", "FiveOneZero"]
```

The `map(_:)` method calls the closure expression once for each item in the array. You don't need to specify the type of the closure's input parameter, `number`, because the type can be inferred from the values in the array to be mapped.

In this example, the variable `number` is initialized with the value of the closure's `number` parameter, so that the value can be modified within the closure body. (The parameters to functions and closures are

always constants.) The closure expression also specifies a return type of `String`, to indicate the type that will be stored in the mapped output array.

The closure expression builds a string called `output` each time it's called. It calculates the last digit of `number` by using the remainder operator (`number % 10`), and uses this digit to look up an appropriate string in the `digitNames` dictionary. The closure can be used to create a string representation of any integer greater than zero.

The string retrieved from the `digitNames` dictionary is added to the *front* of `output`, effectively building a string version of the number in reverse. (The expression `number % 10` gives a value of `6` for `16`, `8` for `58`, and `0` for `510`.)

The `number` variable is then divided by `10`. Because it's an integer, it's rounded down during the division, so `16` becomes `1`, `58` becomes `5`, and `510` becomes `51`.

The process is repeated until `number` is equal to `0`, at which point the `output` string is returned by the closure, and is added to the output array by the `map(_:)` method.

The use of trailing closure syntax in the example above neatly encapsulates the closure's functionality immediately after the function that closure supports, without needing to wrap the entire closure within the `map(_:)` method's outer parentheses.

If a function takes multiple closures, you omit the argument label for the first trailing closure and you label the remaining trailing closures. For example, the function below loads a picture for a photo gallery:

```
func loadPicture(from server: Server, completion: (Picture) -> Void, onFailur
e: () -> Void) {
    if let picture = download("photo.jpg", from: server) {
        completion(picture)
    } else {
        onFailure()
    }
}
```

When you call this function to load a picture, you provide two closures. The first closure is a completion handler that displays a picture after a successful download. The second closure is an error handler that displays an error to the user.

```
loadPicture(from: someServer) { picture in
    someView.currentPicture = picture
```

```
} onFailure: {
    print("Couldn't download the next picture.")
}
```

In this example, the `loadPicture(from:completion:onFailure:)` function dispatches its network task into the background, and calls one of the two completion handlers when the network task finishes. Writing the function this way lets you cleanly separate the code that's responsible for handling a network failure from the code that updates the user interface after a successful download, instead of using just one closure that handles both circumstances.

Capturing Values

A closure can *capture* constants and variables from the surrounding context in which it's defined. The closure can then refer to and modify the values of those constants and variables from within its body, even if the original scope that defined the constants and variables no longer exists.

In Swift, the simplest form of a closure that can capture values is a nested function, written within the body of another function. A nested function can capture any of its outer function's arguments and can also capture any constants and variables defined within the outer function.

Here's an example of a function called `makeIncrementer`, which contains a nested function called `incrementer`. The nested `incrementer()` function captures two values, `runningTotal` and `amount`, from its surrounding context. After capturing these values, `incrementer` is returned by `makeIncrementer` as a closure that increments `runningTotal` by `amount` each time it's called.

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}
```

The return type of `makeIncrementer` is `() -> Int`. This means that it returns a *function*, rather than a simple value. The function it returns has no parameters, and returns an `Int` value each time it's called. To learn how functions can return other functions, see Function Types as Return Types.

The `makeIncrementer(forIncrement:)` function defines an integer variable called `runningTotal`, to store the current running total of the incrementer that will be returned. This variable is initialized with a

value of `0`.

The `makeIncrementer(forIncrement:)` function has a single `Int` parameter with an argument label of `forIncrement`, and a parameter name of `amount`. The argument value passed to this parameter specifies how much `runningTotal` should be incremented by each time the returned incrementer function is called. The `makeIncrementer` function defines a nested function called `incrementer`, which performs the actual incrementing. This function simply adds `amount` to `runningTotal`, and returns the result.

When considered in isolation, the nested `incrementer()` function might seem unusual:

```
func incrementer() -> Int {
    runningTotal += amount
    return runningTotal
}
```

The `incrementer()` function doesn't have any parameters, and yet it refers to `runningTotal` and `amount` from within its function body. It does this by capturing a *reference* to `runningTotal` and `amount` from the surrounding function and using them within its own function body. Capturing by reference ensures that `runningTotal` and `amount` don't disappear when the call to `makeIncrementer` ends, and also ensures that `runningTotal` is available the next time the `incrementer` function is called.

Here's an example of `makeIncrementer` in action:

```
let incrementByTen = makeIncrementer(forIncrement: 10)
```

This example sets a constant called `incrementByTen` to refer to an incrementer function that adds `10` to its `runningTotal` variable each time it's called. Calling the function multiple times shows this behavior in action:

```
incrementByTen()
// returns a value of 10
incrementByTen()
// returns a value of 20
incrementByTen()
// returns a value of 30
```

If you create a second incrementer, it will have its own stored reference to a new, separate `runningTotal` variable:

```
let incrementBySeven = makeIncrementer(forIncrement: 7)
incrementBySeven()
// returns a value of 7
```

Calling the original incrementer (`incrementByTen`) again continues to increment its own `runningTotal` variable, and doesn't affect the variable captured by `incrementBySeven` :

```
incrementByTen()
// returns a value of 40
```

Closures Are Reference Types

In the example above, `incrementBySeven` and `incrementByTen` are constants, but the closures these constants refer to are still able to increment the `runningTotal` variables that they have captured. This is because functions and closures are *reference types*.

Whenever you assign a function or a closure to a constant or a variable, you are actually setting that constant or variable to be a *reference* to the function or closure. In the example above, it's the choice of closure that `incrementByTen` *refers to* that's constant, and not the contents of the closure itself.

This also means that if you assign a closure to two different constants or variables, both of those constants or variables refer to the same closure.

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// returns a value of 50

incrementByTen()
// returns a value of 60
```

The example above shows that calling `alsoIncrementByTen` is the same as calling `incrementByTen` . Because both of them refer to the same closure, they both increment and return the same running total.

Escaping Closures

A closure is said to *escape* a function when the closure is passed as an argument to the function, but is called after the function returns. When you declare a function that takes a closure as one of its parameters, you can write `@escaping` before the parameter's type to indicate that the closure is allowed to escape.

One way that a closure can escape is by being stored in a variable that's defined outside the function. As an example, many functions that start an asynchronous operation take a closure argument as a completion handler. The function returns after it starts the operation, but the closure isn't called until the operation is completed — the closure needs to escape, to be called later. For example:

```
var completionHandlers: [() -> Void] = []
func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void)
{
    completionHandlers.append(completionHandler)
}
```

The `someFunctionWithEscapingClosure(_:)` function takes a closure as its argument and adds it to an array that's declared outside the function. If you didn't mark the parameter of this function with `@escaping`, you would get a compile-time error.

An escaping closure that refers to `self` needs special consideration if `self` refers to an instance of a class. Capturing `self` in an escaping closure makes it easy to accidentally create a strong reference cycle. For information about reference cycles, see Automatic Reference Counting.

Normally, a closure captures variables implicitly by using them in the body of the closure, but in this case you need to be explicit. If you want to capture `self`, write `self` explicitly when you use it, or include `self` in the closure's capture list. Writing `self` explicitly lets you express your intent, and reminds you to confirm that there isn't a reference cycle. For example, in the code below, the closure passed to `someFunctionWithEscapingClosure(_:)` refers to `self` explicitly. In contrast, the closure passed to `someFunctionWithNonescapingClosure(_:)` is a nonescaping closure, which means it can refer to `self` implicitly.

```
func someFunctionWithNonescapingClosure(closure: () -> Void) {
    closure()
}

class SomeClass {
    var x = 10
    func doSomething() {
```

```
        someFunctionWithEscapingClosure { self.x = 100 }
        someFunctionWithNonescapingClosure { x = 200 }
    }
}

let instance = SomeClass()
instance.doSomething()
print(instance.x)
// Prints "200"

completionHandlers.first?()
print(instance.x)
// Prints "100"
```

Here's a version of `doSomething()` that captures `self` by including it in the closure's capture list, and then refers to `self` implicitly:

```
class SomeOtherClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure { [self] in x = 100 }
        someFunctionWithNonescapingClosure { x = 200 }
    }
}
```

If `self` is an instance of a structure or an enumeration, you can always refer to `self` implicitly. However, an escaping closure can't capture a mutable reference to `self` when `self` is an instance of a structure or an enumeration. Structures and enumerations don't allow shared mutability, as discussed in Structures and Enumerations Are Value Types.

```
struct SomeStruct {
    var x = 10
    mutating func doSomething() {
        someFunctionWithNonescapingClosure { x = 200 }  // Ok
        someFunctionWithEscapingClosure { x = 100 }     // Error
    }
}
```

The call to the `someFunctionWithEscapingClosure` function in the example above is an error because it's inside a mutating method, so `self` is mutable. That violates the rule that escaping closures

can't capture a mutable reference to `self` for structures.

Autoclosures

An *autoclosure* is a closure that's automatically created to wrap an expression that's being passed as an argument to a function. It doesn't take any arguments, and when it's called, it returns the value of the expression that's wrapped inside of it. This syntactic convenience lets you omit braces around a function's parameter by writing a normal expression instead of an explicit closure.

It's common to *call* functions that take autoclosures, but it's not common to *implement* that kind of function. For example, the `assert(condition:message:file:line:)` function takes an autoclosure for its `condition` and `message` parameters; its `condition` parameter is evaluated only in debug builds and its `message` parameter is evaluated only if `condition` is `false`.

An autoclosure lets you delay evaluation, because the code inside isn't run until you call the closure. Delaying evaluation is useful for code that has side effects or is computationally expensive, because it lets you control when that code is evaluated. The code below shows how a closure delays evaluation.

```
var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
print(customersInLine.count)
// Prints "5"

let customerProvider = { customersInLine.remove(at: 0) }
print(customersInLine.count)
// Prints "5"

print("Now serving \(customerProvider())!")
// Prints "Now serving Chris!"
print(customersInLine.count)
// Prints "4"
```

Even though the first element of the `customersInLine` array is removed by the code inside the closure, the array element isn't removed until the closure is actually called. If the closure is never called, the expression inside the closure is never evaluated, which means the array element is never removed. Note that the type of `customerProvider` isn't `String` but `() -> String` — a function with no parameters that returns a string.

You get the same behavior of delayed evaluation when you pass a closure as an argument to a function.

```
// customersInLine is ["Alex", "Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: { customersInLine.remove(at: 0) } )
// Prints "Now serving Alex!"
```

The `serve(customer:)` function in the listing above takes an explicit closure that returns a customer's name. The version of `serve(customer:)` below performs the same operation but, instead of taking an explicit closure, it takes an autoclosure by marking its parameter's type with the `@autoclosure` attribute. Now you can call the function as if it took a `String` argument instead of a closure. The argument is automatically converted to a closure, because the `customerProvider` parameter's type is marked with the `@autoclosure` attribute.

```
// customersInLine is ["Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: @autoclosure () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: customersInLine.remove(at: 0))
// Prints "Now serving Ewa!"
```

If you want an autoclosure that's allowed to escape, use both the `@autoclosure` and `@escaping` attributes. The `@escaping` attribute is described above in Escaping Closures.

```
// customersInLine is ["Barry", "Daniella"]
var customerProviders: [() -> String] = []
func collectCustomerProviders(_ customerProvider: @autoclosure @escaping () -
> String) {
    customerProviders.append(customerProvider)
}
collectCustomerProviders(customersInLine.remove(at: 0))
collectCustomerProviders(customersInLine.remove(at: 0))

print("Collected \(customerProviders.count) closures.")
// Prints "Collected 2 closures."
for customerProvider in customerProviders {
    print("Now serving \(customerProvider())!")
}
```

```
// Prints "Now serving Barry!"
// Prints "Now serving Daniella!"
```

In the code above, instead of calling the closure passed to it as its `customerProvider` argument, the `collectCustomerProviders(_:)` function appends the closure to the `customerProviders` array. The array is declared outside the scope of the function, which means the closures in the array can be executed after the function returns. As a result, the value of the `customerProvider` argument must be allowed to escape the function's scope.

- Closures
- Closure Expressions
- The Sorted Method
- Closure Expression Syntax
- Inferring Type From Context
- Implicit Returns from Single-Expression Closures
- Shorthand Argument Names
- Operator Methods
- Trailing Closures
- Capturing Values
- Closures Are Reference Types
- Escaping Closures
- Autoclosures