Closures


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 are defined. This is known as closing over those constants and variables. Swift handles all of the memory management of capturing for you.
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
let names = ["Chris", "Alex", "Eva", "Barry", "Daniella"]

func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}

var reversedNames = names.sorted(by: backward)
// reversedNames is equal to ["Eva", "Daniella", "Chris", "Barry", "Alex"]
Closure Expression Syntax

{ (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.

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

since its short you can write them in one line

```swift
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2 } )
```
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 do not 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:

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

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

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

```swift
reversedNames = names.sorted(by: >)
```
Trailing Closures

let numbers = [16, 58, 510]
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"]
Capturing Values

```swift
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}
```
When considered in isolation, the nested incrementer() function might seem unusual:

```swift
func incrementer() -> Int {
    let runningTotal = 0
    return runningTotal
}
let incrementByTen = makeIncrementer(forIncrement: 10)
incrementByTen() // returns a value of 10
incrementByTen() // returns a value of 20
incrementByTen() // returns a value of 30
```
let incrementBySeven = makeIncrementer(forIncrement: 7)

incrementBySeven()  // returns a value of 7

incrementByTen()  // returns a value of 40
Closures Are Reference Types

```javascript
let alsoIncrementByTen = incrementByTen

alsoIncrementByTen() // returns a value of 50
```
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 is 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) {
    completionHandler.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.

Marking a closure with `@escaping` means you have to refer to self explicitly within the closure. For example, in the code below, the closure passed to `someFunctionWithEscapingClosure(_:)` is an escaping closure, which means it needs to refer 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"
Autoclosures

An autoclosure is a closure that is 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', 'Eva', '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 is not 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', 'Eva', '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 ["Eva", "Barry", "Daniella"]
func serve(customer customerProvider: @autoclosure () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: customersInLine.remove(at: 0))
// Prints "Now serving Eva!"

• Note

Overusing autoclosures can make your code hard to understand. The context and function name should make it clear that evaluation is being deferred.
If you want an autoclosure that is 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.