Cheatsheets / **Learn Intermediate Swift**

# Closures

## Defining a Closure

Closures are self-contained blocks of functionality. Just like functions, closures take in arguments, execute instructions, and return a value or `Void`.

```
let displayWelcome = { () -> Void in
  print("Hello World!")
}

displayWelcome() // Prints: "Hello World"
```

## Inputs and Outputs

Closures can accept inputs and return a value. Unlike functions, closures cannot have argument labels, only internal argument names.

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

print(multiply(4,3)) // Prints: 12
```

## Passing Closures to Functions

Closures can be passed as arguments into functions. Passing closures to functions allows for specification about *how* the function should operate.

```
func combine(_ a: Int, _ b: Int, using combiner: (Int, Int) -> Int) -> Int {
  return combiner(a,b)
}

let add = { (a: Int, b: Int) -> Int in
  return a + b
}

let multiply = { (a: Int, b: Int) -> Int
```

```
in
  return a * b
}

print(combine(2,5, using: add)) //
Prints: 7
print(combine(2,5, using: multiply)) //
Prints: 10
```

## Trailing Closure Syntax

If a function's last argument is a closure, the function can be called using trailing closure syntax. Omit the last argument from the method call and close the parentheses. Then, define the closure immediately after the parentheses are closed.

```
func combine(_ a: Int, _ b: Int, using
combiner: (Int, Int) -> Int) -> Int {
  return combiner(a,b)
}

let sum = combine(2,5) { (a: Int, b:
Int) -> Int in
  return a + b
}

print(sum) // Prints: 7
```

## Shorthand Argument Names

When defining a closure, the arguments in parentheses, return type, and the keyword `in` can be omitted in exchange for shorthand argument labels. `$0` refers to the first argument and `$1` refers to the second argument.

```
func combine(_ a: Int, _ b: Int, using
combiner: (Int, Int) -> Int) -> Int {
  return combiner(a,b)
}

let sum = combine(2,5) { $0 + $1 }

print(sum) // Prints: 7
```

## Common Higher-order Functions

A higher-order function is a function that takes another function as an argument. The Swift standard library provides a number of useful higher-order methods. The most commonly used are `filter`, `map`, `reduce`, and `sorted`.

```
let scores = [4,10,3,7,5]

let evenScores = scores.filter { $0 % 2 == 0 }
print(evenScores) // Prints: [4,10]
let doubledScores = scores.map { $0 * 2 }
print(doubledScores) // Prints: [8, 20, 6, 14, 10]
let sumOfScores = scores.reduce(0) { $0 + $1 }
print(sumOfScores) // Prints: 29
let sortedScores = scores.sorted { $0 < $1 }
print(sortedScores) // Prints: [3, 4, 5, 7, 10]
```

## Escaping Closures

A closure *escapes* a function when it's called after the function returns. This can happen when the closure is assigned to a variable. Escaping closures must be marked with the `@escaping` tag in a function signature.

```
struct TextSaver {
  var saveAction: (String) -> Void = { print("Saving '\($0)' to disk") }

  mutating func setSaveAction(to newAction: @escaping (String) -> Void) {
    saveAction = newAction
  }
}

var saver = TextSaver()
saver.saveAction("Hello World!")
// Prints: Saving 'Hello World!' to disk
saver.setSaveAction(to: { print("Saving '\($0)' to the cloud") })
saver.saveAction("Hello World!")
```

```
// Prints: Saving 'Hello World!' to the
cloud
```

## Capturing Values

Closures can capture values from their surrounding scope. When a closure captures a value, it keeps track of it and can manipulate the value even if the original function returns.

```
func makeCountingPrinter(for str:
String) -> () -> Void {
  var printCount = 0
  func strPrinter() -> Void {
    printCount += 1
    print("\(str) print count: \
(printCount)")
  }
  return strPrinter
}

let printHello =
makeCountingPrinter(for: "Hello!")
let printGoodbye =
makeCountingPrinter(for: "Goodbye!")

printHello() // Prints: Hello! print
count: 1
printHello() // Prints: Hello! print
count: 2
printGoodbye() // Prints: Goodbye! print
count: 1
printHello() // Prints: Hello! print
count: 3
printGoodbye() // Prints: Goodbye! print
count: 2
```