Cheatography

Swift 5 Reference Cheat Sheet

by zyqhi via cheatography.com/79780/cs/19305/

Closures

```
// Syntax
{ (parameters) -> return type in
    statements
}
```

1. Closures are self-contained blocks of functionality that can be passed around and used in your code.

Closures Syntax Optimizations

```
let names = ["Chris", "Alex", "Ewa", "Barry",
"Daniella"]
// 1
reversedNames = names.sorted(by: { (s1: String, s2:
String) -> Bool in
    return s1 > s2
})
// 2. Inferring Type From Context
reversedNames = names.sorted(by: { s1, s2 in return
s1 > s2 } )
// 3. Implicit Returns from Single-Expression
Closures
reversedNames = names.sorted(by: { s1, s2 in s1 >
s2 } )
// 4. Shorthand Argument Names
reversedNames = names.sorted(by: { $0 > $1 } )
// 5. Operator Methods, operator (>) is a method
reversedNames = names.sorted(by: >)
// 6. Trailing Closures
reversedNames = names.sorted() { $0 > $1 }
// function with only one arguement
reversedNames = names.sorted { $0 > $1 }
```

1. A trailing closure is written after the function call's parentheses, even though it is still an argument to the function.

Escaping Closures

```
var completionHandlers: [() -> Void] = []
func someFunctionWithEscapingClosure(handler:
@escaping () -> Void) {
    completionHandlers.append(handler)
}
func someFunctionWithNonescapingClosure(closure:
() -> Void) {
    closure()
}
class SomeClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure {
            self.x = 100
        }
        someFunctionWithNonescapingClosure {
            x = 200
        }
    }
}
```

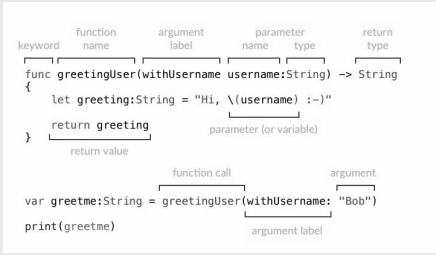
1. A closure is said to escape a function when the closure is passed as an argument to the function, but is called after the function returns.
2. Marking a closure with *@escaping* means you have to refer to self explicitly within the closure.

Autoclosures

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

1. An *autoclosure* is a closure that is automatically created to wrap an expression that's being passed as an argument to a function.
2. An autoclosure lets you delay evaluation.

Functions



1. Functions are self-contained chunks of code that perform a specific task.

Property

```
class SomeClass {
    // Type Property
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty:
Int {
        return 107
    }

    // Lazy Stored Property
    lazy var view = UIView()

    // Stored Property
    let data = [String]()

    // Computed Property
    var id: Int = 0 {
        willSet {
            print("About to set \(newValue)")
        }
        didSet {
            print("Did set id to \(id), oldValue is
\(oldValue)")
        }
    }

    var idStr: String {
        set {
            id = Int(idStr) ?? 0
        }
        get {
            return String(id)
        }
    }

    var desc: String {
        return "Readonly property."
    }
```

Property (cont)

```
}
```

1. Computed properties must be declared with the *var* keyword, because their value is not fixed.
2. A lazy stored property is a property whose initial value is not calculated until the first time it is used.

Enumeration

```
/// Use enum
enum Planet {
    case mercury, venus, earth, mars, jupiter,
saturn, uranus, neptune
}
var p = Planet.venus
p = .uranus
/// Iterating over Enumeration Cases
enum Beverage: CaseIterable {
    case coffee, tea, juice
}
let numberOfChoices = Beverage.allCases.count
print("\(numberOfChoices) beverages available")
for beverage in Beverage.allCases {
    print("bevergae is \(beverage)")
}
/// Associated Values
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```

1. An enumeration defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code.
2. Conforming to the *CaseIterable* protocol makes a enum iterable.