Sun*

Closures

Outline

1. Definition
2. Closure Expressions
3. Trailing Closures
4. Capturing Values
5. Escaping Closures
6. Autoclosures

1. Definition

1. Definition
2. Forms of Closures

1.1 Definition

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

1.2 Forms of Closures

- Closures take one of three forms:
  - ➔ Global functions are closures that have a name and do not 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.

1.2 Forms of Closures (cont)

- Global functions are closures that have a name and do not capture any values:

```
var a = 10 // global variable
// Global function with name
func globalClosure() {
    print("value of a: \(a)")
}

func someFunctionWithGlobalClosure() {
    var a = 5
    globalClosure()
}

a = 14
someFunctionWithGlobalClosure() // Prints: value of a: 14, not 5
```

1.2 Forms of Closures (cont)

- Nested functions are closures that have a name and can capture values from their enclosing function:

```
let a = 10

// Thí function returns a function of type () → Void
func someFunctionWithNestedClosure() → () → Void {
    var a = 5
    func nestedFunction() {
        print("value of a: \(a)")
        a += 1
    }
    return nestedFunction
}

var closure = someFunctionWithNestedClosure()
closure() // Prints: value of a: 5
closure() // Prints: value of a: 6
```

1.2 Forms of Closures (cont)

- Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context:

```
let addTwoInts = { (_ a: Int, _ b: Int) → Int in a + b }

print(addTwoInts(1, 2)) // Prints "3"
```

2. Closure Expressions

1. Closure Expression Syntax
2. Inferring Type From Context
3. Implicit Returns from Single-Expression Closures
4. Shorthand Argument Names

2.1 Closure Expression Syntax

- Closure expression syntax has the following general form:

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

2.1 Closure Expression Syntax (cont)

- You can define a closure like this:

```
let sayHello: (String) → String = { (name: String) in
    return "Greeting \(name)"
}

print(sayHello("John")) // Prints "Greeting John"
```

2.2 Inferring Type From Context

- If you have defined type of closures, Swift can infer the types of its parameters and the type of the value it returns:

```
let addTwoInts: (Int, Int) → Int = { a, b in
    return a + b
}
```

2.3 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:

```
let addTwoInts: (Int, Int) → Int = { a, b in a + b }
```

2.4 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:

```
let addTwoInts: (Int, Int) → Int = { $0 + $1 }
```

3. Trailing Closures

- A *Trailing Closure* is a concept which states that if, a function takes the ***last parameter*** as a closure, then while calling that function:
  - ➔ We can omit/remove the name of the closure parameter inside the function
  - ➔ We can close the function parameter list with `)` and then write our closure body within `{ }` after the closing `)`

3. Trailing Closures (cont)

```
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
}
```

4. Capturing Values

- A closure can *capture* constants and variables from the surrounding context in which it is defined.
- In Swift, the simplest form of a closure that can capture values is a nested 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.

4. Capturing Values (cont)

```swift
func makeIncrementer(forIncrementer amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
// incrementer() capture a reference to `runningTotal` and `amount`
        return runningTotal
    }
    return incrementer
}

let incrementByTen = makeIncrementer(forIncrementer: 10)
// incrementer() keeps references of `runningTotal` and `amount`
print(incrementByTen()) // Prints "10"
print(incrementByTen()) // Prints "20"

let incrementBySeven = makeIncrementer(forIncrementer: 7)
// incrementer keeps other references of `runningTotal` and `amount`
print(incrementBySeven()) // Prints "7"
print(incrementBySeven()) // Prints "14"
```

4. Capturing Values (cont)

- A weak/unowned reference is a pointer to object which is not protected from being deallocated by ARC.

- Strong reference increase the retain count of object by 1, weak/unowned references ***do not***.

- **`weak`** and **`unowned`** keyword:
  - ➔ All weak references are non-constant Optionals
  - ➔ An unowned reference has the added benefit of ***not being an Optional***

4. Capturing Values (cont)

*“Use a **weak** reference whenever it is valid for that reference to become nil at some point during its lifetime.*
*Conversely, use an **unowned** reference when you know that the reference will never be nil once it has been set during initialization”*

5. 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.
- You can write **`@escaping`** before the parameter's type to indicate that the closure is allowed to escape.

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

6. 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 returns the value of the expression that's wrapped inside of it when it's called.
- An *autoclosure* lets you delay evaluation, because the code inside isn't run until you call the closure.
- `@autoclosure` attribute enables you to define an argument that automatically gets wrapped in a closure.

6. Autoclosures

```
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"
```

6. Autoclosures

- If you want an *autoclosure* that is allowed to escape, use both the **`@autoclosure`** and **`@escaping`** attributes.

```
// 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!"
```

Question & Answer?