iOS Programming

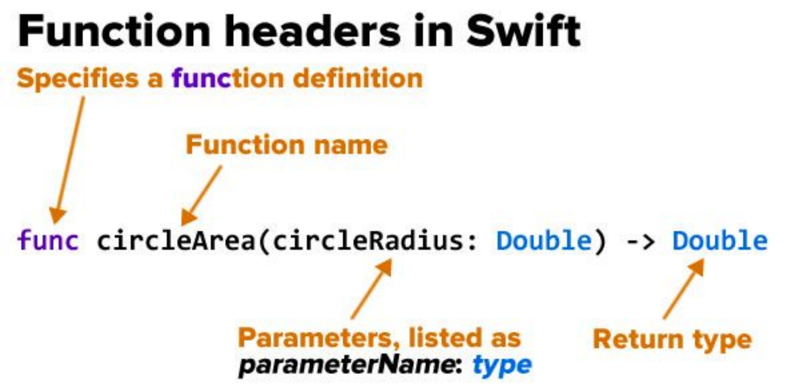
Lecture 5

Recap

Conditional Statements

Loops

Functions



Function headers in Swift

```
func circleArea(circleRadius: Double) -> Double
```

Today – Complex & Custom Data Types

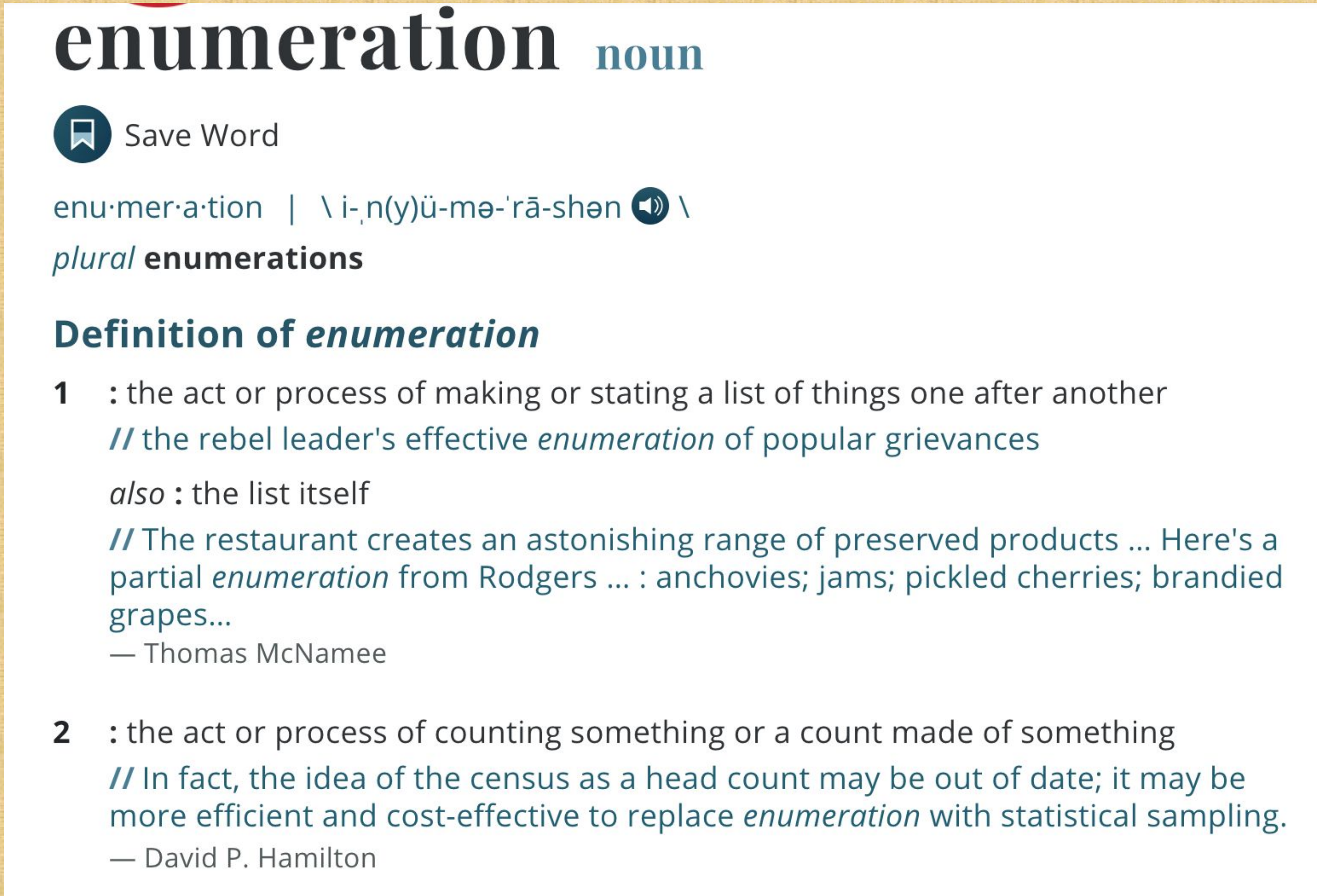
Enumeration

Structs

Tuples

Closures

Enumeration



enumeration noun

Save Word

enu·mer·a·tion | \ i-ˌn(y)ü-mə-ˈrā-shən 🔊 \

plural **enumerations**

Definition of *enumeration*

1 **:** the act or process of making or stating a list of things one after another

// the rebel leader's effective *enumeration* of popular grievances

also **:** the list itself

// The restaurant creates an astonishing range of preserved products … Here's a partial *enumeration* from Rodgers … : anchovies; jams; pickled cherries; brandied grapes…

— Thomas McNamee

2 **:** the act or process of counting something or a count made of something

// In fact, the idea of the census as a head count may be out of date; it may be more efficient and cost-effective to replace *enumeration* with statistical sampling.

— David P. Hamilton

Enumeration

```
//I am building an Calculator functionality to return result based on the operation user
passes

func calculate(firstParamter: Double, secondParameter: Double, usingOperator:
String) -> Double{
    switch (usingOperator){
    case "+":
        return firstParamter + secondParameter;
    case "-":
        return firstParamter - secondParameter;
    case "*":
        return firstParamter * secondParameter;
    case "/":
        return firstParamter / secondParameter;
    //But swift knows that your data type allows for more values, the below return
doesn't works
    default:
        return -1.00
    }
}
```

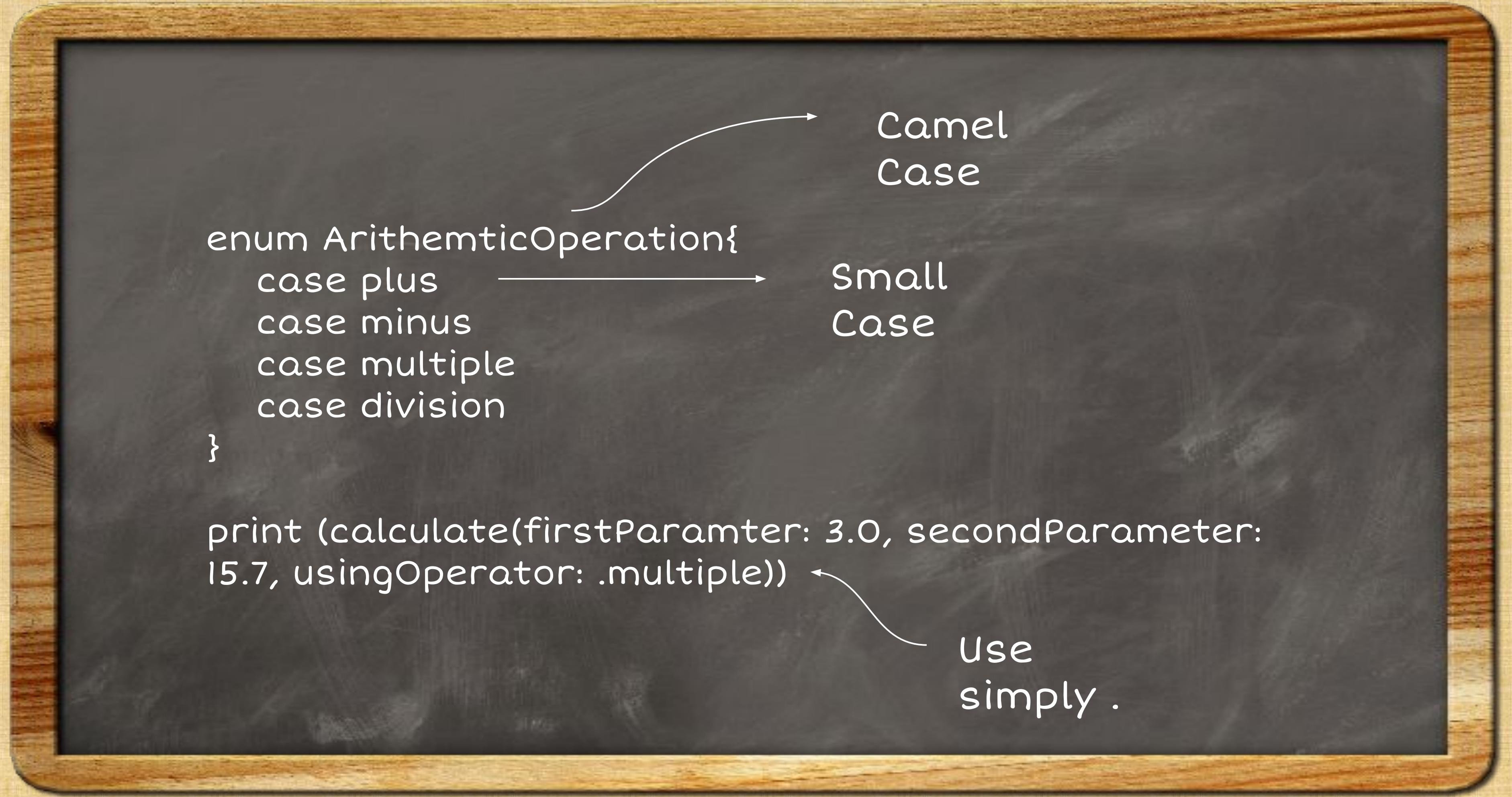
Enumeration

```
enum ArithemticOperation{
    case plus
    case minus
    case multiple
    case division
}

func calculate(firstParamter: Double, secondParameter: Double, usingOperator:
ArithemticOperation) -> Double{
    switch (usingOperator){
    case .plus:
        return firstParamter + secondParameter;
    case .minus:
        return firstParamter - secondParameter;
    case .multiple:
        return firstParamter * secondParameter;
    case .division:
        return firstParamter / secondParameter;
    }
}

print (calculate(firstParamter: 3.0, secondParameter: 15.7, usingOperator:
ArithemticOperation.multiple))
```

Enumeration - Swifty



```
enum ArithemticOperation{
    case plus
    case minus
    case multiple
    case division
}

print (calculate(firstParamter: 3.0, secondParameter:
15.7, usingOperator: .multiple))
```

Camel Case

Small Case

Use simply .

Enumeration - Swifty

More Swifty

```
enum ArithemticOperation{
    case plus
    case minus
    case multiple
    case division
}

enum ArithemticOperation{
    case plus, minus, multiple, division
}
```

Enumeration – Raw Values

```
enum ArithemticOperation: String{
    case plus = "+"
    case minus = "-"
    case multiple = "*"
    case division = "/"
}

func calculate(firstParamter: Double, secondParameter: Double, usingOperator: ArithemticOperation) -> Double{
    switch (usingOperator){
    case .plus:
        return firstParamter + secondParameter;
    case .minus:
        return firstParamter - secondParameter;
    case .multiple:
        return firstParamter * secondParameter;
    case .division:
        return firstParamter / secondParameter;
    }
}

var firstParameter = 3.0
var secondParameter = 15.7
var operation: ArithemticOperation = .multiple
var calculatedResult = calculate(firstParamter: 3.0, secondParameter: 15.7, usingOperator:ArithemticOperation.multiple)
print ("\(firstParameter) \(operation.rawValue) \(secondParameter) = \(calculatedResult)")
```

Enumeration – Associated Values

```
enum Actor{
    case age(Int)
    case fullName(String)
    case netWorth(Double)
}


let arnoldAge: Actor = .age(56)
let arnoldWorth: Actor = .netWorth(10000000.23)


switch (arnoldAge){
case .age(let age):
    print ("Actor's age: \(age)")
case .fullName(let fullName):
    print ("Actor's full name: \(fullName)")
case .netWorth(let netWorth):
    print ("Actor's net worth: \(netWorth)")
}
```

Structs

Did you realize that you have been already been using structs all along.

All the Data Types we have been using so far are actually structs in Swift:

- Int
- Double
- String

Structs are powerful in swift

Structs

We can group properties under a common logical structure

```
struct Actor{
    var age: Int
    var fullName: String
    var netWorth: Double
}


let actor: Actor = Actor(age: 51, fullName: "Brad Pitt", netWorth: 32000000.45)


print (actor)
```

Structs

We can add methods to structs

```
struct Actor{
    var age: Int
    var fullName: String
    var netWorth: Double

    func isRich() -> Bool{
        if netWorth > 1000000{
            return true
        }
        return false
    }
}

let actor: Actor = Actor(age: 51, fullName: "Brad Pitt", netWorth: 32000000.45)

print ("\(actor.fullName) is rich: \(actor.isRich())")
```

Structs

No inheritance in structs

Structs are pass by value and not by reference

Tuples

Tuples is a mechanism to group multiple properties together

```
let antonyms = [ "Hot": "Cold",
                 "Sunny": "Dark",
                 "Happy": "Sad"
]

for (key, value) in antonyms {
    print ("\(key):\(value)")
}
```

Tuples

You can read individual elements of a tuple

```
func customerInfo() -> (String, String) {
    return ("Joe", "Doe")
}


var customer = customerInfo()
print ("Customer Name: \(customer.0) \(customer.1)")
```

Tuples

You can read individual elements by assigned names rather than indexes

```
func customerInfo() -> (firstName: String, lastName: String) {
    return ("Joe", "Doe")
}


var customer = customerInfo()
print ("Customer Name: \(customer.firstName) \(customer.lastName)")
```

Closures vs Functions

```
func customerInfo() -> (firstName: String, lastName: String) {
    return ("Joe", "Doe")
}


var customer = customerInfo()
```

Both are
Re-usable blocks of code

```
//Closure
{
    //Code to pass
}
```

Closures are nameless blocks of code

Remember Function Types???

Closures

Where can they useful???

Other languages have had or adopted similar constructs:

- Blocks
- Lambdas
- Anonymous Functions

Closures are nameless blocks of code that you pass

Closures – In Action

```
struct Student{
    var name: String
    var swiftScore: Int
    var kotlinScore: Int
}


var alan = Student.init(name: "Alan", swiftScore: 100, kotlinScore: 25)
var chrissy = Student.init(name: "Chrissy", swiftScore: 25, kotlinScore: 100)
var susan = Student.init(name: "Susan", swiftScore: 60, kotlinScore: 60)


let students = [alan, chrissy, susan]

//The below function provides sort criteria
func sortStudents(firstStudent: Student, secondStudent: Student)->Bool{
    if (firstStudent.kotlinScore > secondStudent.kotlinScore){
        return true;
    }
    return false;
}


//Swift will take care of the actual sorting for us we simply need to tell which element is
students.sorted(by: sortStudents(firstStudent:secondStudent:))
```

Functions are first class data types in Swift

Closures – Let’s make it Swifty

```
struct Student{
    var name: String
    var swiftScore: Int
    var kotlinScore: Int
}

var alan = Student.init(name: "Alan", swiftScore: 100, kotlinScore: 25)
var chrissy = Student.init(name: "Chrissy", swiftScore: 25, kotlinScore: 100)
var susan = Student.init(name: "Susan", swiftScore: 60, kotlinScore: 60)

let students = [alan, chrissy, susan]

//Swift will take care of the actual sorting for us we simply need to tell which element is of interest
let sortedStudents = students.sorted(by: {
    (firstStudent: Student, secondStudent: Student)->Bool
    in
    if (firstStudent.kotlinScore > secondStudent.kotlinScore){
        return true;
    }
    return false;
})
print (sortedStudents)
```

Closures – More Swifty

```
struct Student{
    var name: String
    var swiftScore: Int
    var kotlinScore: Int
}

var alan = Student.init(name: "Alan", swiftScore: 100, kotlinScore: 25)
var chrissy = Student.init(name: "Chrissy", swiftScore: 25, kotlinScore: 100)
var susan = Student.init(name: "Susan", swiftScore: 60, kotlinScore: 60)

let students = [alan, chrissy, susan]

//Swift will take care of the actual sorting for us we simply need to tell which element is of interest
let sortedStudents = students.sorted(by: {
    if ($0.kotlinScore > $1.kotlinScore){
        return true;
    }
    return false;
})
print (sortedStudents)
```

Closures – Even More

```
struct Student{
    var name: String
    var swiftScore: Int
    var kotlinScore: Int
}

var alan = Student.init(name: "Alan", swiftScore: 100, kotlinScore: 25)
var chrissy = Student.init(name: "Chrissy", swiftScore: 25, kotlinScore: 100)
var susan = Student.init(name: "Susan", swiftScore: 60, kotlinScore: 60)

let students = [alan, chrissy, susan]

//Swift will take care of the actual sorting for us we simply need to tell which element is of interest
let sortedStudents = students.sorted{
    if ($0.kotlinScore > $1.kotlinScore){
        return true;
    }
    return false;
}
print (sortedStudents)
```

Closures – Still More

```swift
struct Student{
    var name: String
    var swiftScore: Int
    var kotlinScore: Int
}

var alan = Student.init(name: "Alan", swiftScore: 100, kotlinScore: 25)
var chrissy = Student.init(name: "Chrissy", swiftScore: 25, kotlinScore: 100)
var susan = Student.init(name: "Susan", swiftScore: 60, kotlinScore: 60)

let students = [alan, chrissy, susan]

//Swift will take care of the actual sorting for us we simply need to tell which element is of interest
let sortedStudents = students.sorted{ return $0.kotlinScore > $1.kotlinScore }
print (sortedStudents)
```

Closures – Last Bit

```
struct Student{
    var name: String
    var swiftScore: Int
    var kotlinScore: Int
}

var alan = Student.init(name: "Alan", swiftScore: 100, kotlinScore: 25)
var chrissy = Student.init(name: "Chrissy", swiftScore: 25, kotlinScore: 100)
var susan = Student.init(name: "Susan", swiftScore: 60, kotlinScore: 60)

let students = [alan, chrissy, susan]

//Swift will take care of the actual sorting for us we simply need to tell which element is of interest
let sortedStudents = students.sorted{ $0.kotlinScore > $1.kotlinScore }
print (sortedStudents)
```

For single line closures even return is optional

Parting Notes

Practice:

- Enumerations
- Structs
- Tuples
- Closures

Simple Exercise:

Create your Student type with Enum defining the type as Mobile, Web, Analytics, Services and then use the enum to filter the students.

Obviously, you need to use closures.