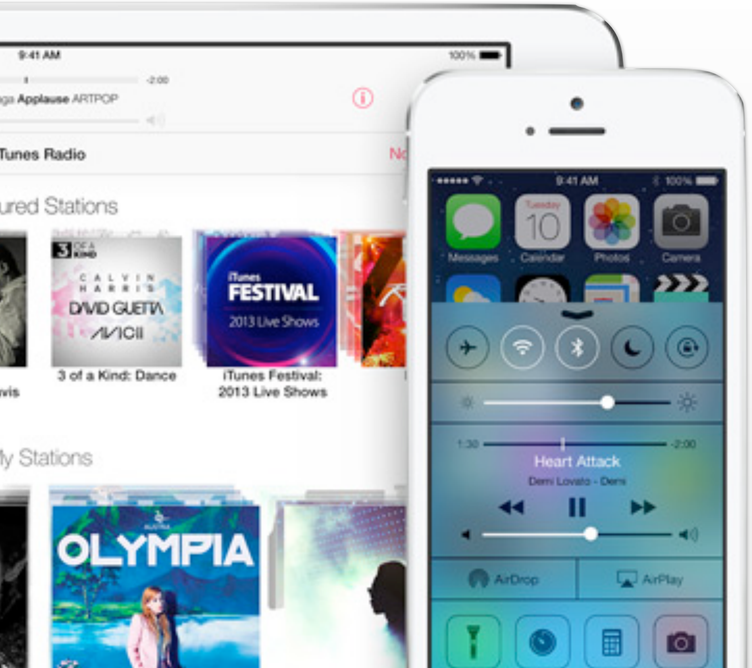



# iPhone Application Programming
# Lecture 2: Swift Part 1

Nur Al-huda Hamdan
Media Computing Group
RWTH Aachen University

Winter Semester 2015/2016

http://hci.rwth-aachen.de/iphone

Media Computing Group | RWTH AACHEN UNIVERSITY

# Review

- Mobile device characteristics?
- Differences between mobile device and desktop?
- Golden rules of interface design?
- Application types?
- Design themes?

# Hello World Demo

- Xcode IDE
- AppDelegate and ViewController
- App UI
- Outlets and Actions
- UIButton and UIAlert
- Documentation
- Check out L01 slides to understand the development environment


If you missed the lab:
(a) Check out how to setup iOS development environment here
(b) Check out the Jump Right In iOS programming tutorial from Apple (we covered until "Implement a Custom Control)"
(c) First assignment and reading requirement on our website

# iOS Apps

- Composed of objects that send messages to each other
  - UILabel is an object implemented in UIKit and is ready to be used
  - UIViewController is an object that you can implement
  - You can create your own objects, i.e., classes
- Event driven. The app does nothing until it receives an event from the user (e.g., a touch) or the system (e.g., full memory, incoming call, timer), or the data source (e.g., new data arrived)

# Swift

- Swift is the new programming language for iOS, OS X, watchOS, and tvOS apps
- Unifies the procedural and object-oriented portions of the language
- Safe programming patterns
- Seamless bridging between Swift and Objective-C types
- Unlike Objective-C, Swift is not compatible with C
- Playground feature

# Syntax

- Naming conventions
  - CamelCase for types (classe, enum, struct)
  - camelCase for variables, constants, functions, and properties
  - Case sensitive, i.e., myObject and MyObject are different identifiers
  - Escape keywords `` `func` ``
- Comments
- No semicolon, unless between several statements on the same line

**Naming convention**

```
class Report {}
func checkResults () {}
var grade:Float
```

**Comments**

```
//Comments can be single line

/* or multiple lines
    /* can be nested */
*/
```

# Basic Data Types

- Basic data types: `Int`, `Float`, `Double`, `Bool`, `String`, `Character`, optional`?`, collection types: tuples, arrays, dictionaries, sets
  - Types have different values, ranges, and methods
  - Type conversion, e.g., `Int(doubleValue)` or `Double(StringValue)`
- Type alias for renaming types
- Type safe: All variables need to have a type. If you define a `Bool` swift guarantees the value true/false, nothing else. No space for confusion. Checked at compile time
- Type inference: No need to declare types if they can be inferred from the context

**Types methods**

```
UInt16.max //65,535
```

**Type alias**

```
typealias MyInt = Int
var someInt:MyInt = 5
```

**Type safe**

```
let i = 1
if i { // compile time error}
 if i == 1 { // is valid}

someDouble+anotherDouble // works
someInt+someDouble // error
```

**Type inference**

```
let someInt = 5
let someDouble = 5.0
let someExplicitDouble:Double = 5
let someExplicitString:String
someExplicitString = "Hello there!"
```

# Variables and Constants (Mutability)

- Variables
  - `var variableName = <initial value>`
  - `var variableName:<data type> = <optional initial value>`
- Constant values cannot be modified after definition
  - Similar to variables but with `let`
- One either explicitly specifies the type of `var`/`let` or provides a default value
- All variables and constants must be initialized before use, expect for optionals
- String interpolation

```
var var1:Int, var2: Double, var3 = 7, 😊 = "Happy"
var1 = 8
var2 = 9.1
var2 = 0.3
let const1 = 2
let const2:Float
print(const1)
const2 = 1.8
const2 = 3.14 // error
```

```
var arrayOfInts:[Int]                    Invalid redeclaration of 'arrayOfInts'
arrayOfInts.isEmpty
```

**Interpolation**

```
print("I am " + 😊 + " with the \(var1 + const1)
friends I have on Facebook.”)
//“I am Happy with the 10 friends I have on
Facebook.\n"
```

# Optional?

- A new data type that handles the absence of a value "nil". Has 2 possible values:
  - (a) there is a value and its equal to x, or (b) there is no value
- `var perhapsInt : Int?` (optional Int)
- If you think a variable can have no value during execution, declare it optional
- Reading optionals requires unwrapping. Writing to optionals doesn't require a thing

```
let possibleString: String? //default is nil
possibleString = "An optional string."
let forcedString: String = possibleString! //unwarp

let assumedString: String! = "An implicitly unwrapped
optional string."
// no need for an exclamation mark but if assumedString
is nil, a runtime error occurs
let aString: String = assumedString
//alternatively, use optical binding
if let definiteString = assumedString
{
    print(definiteString)
}
```

# Tuples

- Tuples group multiple values of different types into a single compound value
- Access values in a tuple with deconstruction, indices, or names
- Useful as the return values of functions

```
let http404Error = (404, "Not Found")

let (statusCode, statusMessage) = http404Error //deconstruction
print("The status code is \(statusCode)")

let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)")

print("The status message is \(http404Error.1)”)//indices
// prints "The status message is Not Found"

let http200Status = (statusCode: 200, description: “OK”)//names
print("The status message is \(http200Status.description)")
```

# Control Flow - Decision Making

- Switch cases must be exhaustive or you will get a compile error and must add the `default` case
- No need for `break` between cases
- `fallthrough` allows you to execute the following case statements
- Can do pattern matching, e.g., `case (0,_,"hi")` or `case 0…5`
- Value binding, e.g., `case (200..<400, let description)`

```
if 10 > 7 {} //do
else {} //do something else
//can be nested

switch anyType
{
    case option1: //do this
    //no need to break, that is the default beahviour
    case option2, option3: //do that
        fallthrough //execute the next case too
    case option4: //do things
    default: //do default case
}
```

# Control Flow - Loops and Ranges

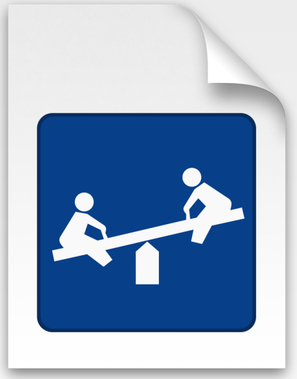
- Can use control statement `continue` and `break`
- Notice the range operator `#1..<#2` (half opened) and `#1...#2` (closed)
- Notice the wild card pattern `_` matches and ignores any value

```
for i in 5..<8 {print(i)} //iterate 5,6,7 (3 loops)
for _ in 0...4 {print("I forgot my homework")}//iterate 0,1,2,3,4 (5
loops)
for var i = 0; i<10; ++i
{
    print(Int(i)) //or print(i), same result
}

while condition
{statement(s)}

do{statement(s)} while condition
```

# String

- `String` is composed of extended grapheme clusters for `Character` values
  - String concatenation and modification may not always affect a string's character count
  - Example: cafe is 4 characters, if you append a COMBINING ACUTE ACCENT (U +0301) to the end it becomes café, but it's still 4 characters
- String interpolation constructs a new string value from other types using `"\(swift code)"`
- String concatenation constructs a new string from `String + String` but not with `Character`. You cab append a `Character` to a `String`
- Use the equal to `==` operator to check `Strings` or `Characters` equality. They are equal if they have the same linguistic meaning and appearance, e.g., `cafe !== café //true`

# Arrays



- Ordered list of values of the same type
- If declared with var it can be modified in content and size, if with let it cannot be changed in anyway after definition
- You access array elements using indices
- `var someArray : [SomeType]`
- `var someArray = [SomeType](count: NumbeOfElements, repeatedValue: InitialValue)`
- `var someInts : [Int] = [10, 20, 30]`

# Dictionaries

- Unordered list of elements (key-value pairs) of the same type
  - All values should have the same type, and all key should have the same type
- Unique identifier key is used to access and modify values (unlike arrays using indices)
- `var someDict : [KeyType: ValueType]`
- `let someDict : [Int:String] = [1:"One", 2:"Two", 3:”Three"]`
- `var` means mutable (can add, remove, and modify elements); `let` means immutable after the first definition
- Dictionaries methods that return the value of a key have optional return type

# Functions

- `func funcname(parameters) -> returntype { statment(s) }`
- Can have 0…N, parameters of any type, with local and external names, can be passes by value or by reference
- Can have 0…1 return type
- Functions can pass and return any data type, including optional, tuple, and function type
- Functions can have the same name if they have different definitions (parameter types and the return type) or external parameter names

# Functions - Parameter Names

- By default, the first parameter omits its external name, and subsequent parameters use their local name as their external name
  - External name then local name
- All parameters *must* have unique local names but not unique external names
- Use underscore (_) for subsequent parameters to avoid using parameter name in function call
- External names must always be used when calling the function
- Functions of the same type but different external names are considered unique

```
func combineValues(value1:Int, _ value2:Int, valueThree value3:Int)
-> (_:Int,secondValue:Int, _:Int?)
{
    return (value1,value2,value3)
}
combineValues(10, 20, valueThree:30).secondValue //20

if let thirdValue = combineValues(10, 20, valueThree:30).2 //access
tuple by index
{
    print("That was an Optional type in a tuple type")
}
```

# Functions - Parameter Mutability

- By default all function parameters are constants
- `var` parameters are passed by value and mutable within the function body
- `inout` parameters are passed by reference and mutable
  - `inout` parameters cannot be constants, literals, or have default values, be variadic, or be defined as var or let

```
func manipulateValues(value1:Int, var _ value2:Int, inout valueThree
value3:Int, valueFour _:Int) -> (_:Int,secondValue:Int, _:Int, _:Int?)
{
    value1 += 1 //error, this is a constant (let) by default
    value2 += 1
    value3 += 1
    return (value1,value2,value3,3)
}

var someValue = 2
var anotherValue = 3
manipulateValues(1, someValue, valueThree:&anotherValue, valueFour:4)
someValue // 2
anotherValue //4
```

# Functions - Variadic Parameters and Default Values

- A variadic parameter can pass 0…N values of the same type
- In the function body a variadic is treated as an array
- Each function can have at most one variadic and it should appear as the last parameter
- Functions can have parameters with default values

**Variadic**

```
//<N> means one can pass any type to the
variadic
func vari<N>(members: N...){
    for i in members {
        print(i)
    }
}
vari(4,3,5)
vari(“4”,”3”,”5”)
```

**Default**

```
func addToContactList(name:String,phone:String,
list:String = "Friends")
{
    print("New contact "+name+" with phone number:
\(phone), was added to list \(list)")
}

addToContactList("Lara", phone: "01234567")
addToContactList("Moe", phone: "76543210",
list:"Work")
```

# Functions - Functions as Types

- Function type is defined by the function's parameter types and return type (not name)
  - `sum` function is of type `(Int, Int) -> Int`
  - `another` function is of type `((Int, Int) -> Int, Int, Int)`
- Functions (using their names) can be passed as function parameters

```
func sum(a: Int, b: Int) -> Int {
    return a + b
}
var addition: (Int, Int) -> Int = sum

print("Result: \(addition(40, 89))”) //129
print("Result: \(sum(40, 89))”) //129

//function as a parameter type
func another(add: (Int, Int) -> Int, a: Int, b: Int)
{
    print("Result: \(add(a, b))")
}
another(sum, a: 10, b: 20) //30
```

# Functions - Nesting

- A nested function (`decrementer`) is only accessible from within its enclosing function (`calcDecrement`)
- The nested captures a reference to any of its outer function's arguments, or constants and variables defined within the outer function
- Capturing by reference ensures the variables do not disappear when the call to outer function ends, and that the variables are available the next time the nested function is called

```
func calcDecrement(forDecrement total: Int) -> () -> Int
{
    var overallDecrement = 0
    func decrementer() -> Int { //nested function
        overallDecrement -= total
        return overallDecrement
    }
    return decrementer //function as return type
}
let decrem30 = calcDecrement(forDecrement: 30)
print(decrem30()) //-30
let decrem10 = calcDecrement(forDecrement: 10)
print(decrem10()) //-10
print(decrem30()) //-60 decrements its own
overallDecrement and is not effected by decrem10
```

# Classes

- Single inheritance
- Type casting
- Deinitialization for memory management
- Reference type
- Class instances are always passed by reference
- === is true if two constants or variables point to the same instance
- Properties store values, Subscripts give access to values (check these out!), methods define behavior, initializers, (later extensions and protocols)

# Classes

- Class definition
- Object instantiation
- Read/write to a property
- Call a method

```
class Recipe:Inheritance
{
    var ingredients = [String]()
    var levelsOfDifficulty = 1
    var takeMoreThanThirtyMins = true
    func cookingTimeInMins() -> Int
    {
        if (takeMoreThanThirtyMins)
        {
            return (30 + (ingredients.count * levelsOfDifficulty))
        }
        return 30
}    }
var myRecipe = Recipe()
myRecipe.ingredients = ["Rice", "Meat", "Salt"]
print(myRecipe.levelsOfDifficulty)
myRecipe.cookingTimeInMins()
```

# Next Time

- The slides and playgrounds from this lecture will be uploaded to our website
- This week's reading assignment is on the website
- Next week we do not have a lecture, but we have a lab
- On 10.11. we will continue with Swift syntax and talk about: properties, methods, inheritance, initialization, memory management, extensions, protocols, access control