PLAIN HTML VERSION: lecture3.161.txt.html
Xcode Project Archive: lecture3.161.zip

```
/*
file:    lecture3.161.swift
author:  Danny Abesdris
date:    January 27, 2016
purpose: MAP523AA/DPS923AA lecture #3
         Swift programming language:
         Functions: (parameter passing, label names, return values (optional type),
                    function overloading, preconditions, types (signature),
                    value vs reference (inout), variable argument lists).
         Arrays:    (initializers, instance variables (properties), instance methods,
                    operators, passing arrays to functons).
*/

import Foundation

var response : String = ""

/*
Swift Functions:
Functions in Swift are very powerful and flexible. Function basics are simple (especia
if you're famillar with functions in languages like C, Python, Javascript, etc).
However, because of the flexibility of functions in Swift, a good understanding of the
basics is essential in order to what can easily turn into complex looking code.

In Swift, a function is nothing more than a block of code that can be executed wheneve
it's needed as in:

func printSomeMessage(message: String) {
    print(message)
}

General syntax:

func funcname(Parameters) -> returntype {
   statement(s)
   return parameters
}

A function begins with the func keyword and is followed by the name of the function
(in the example above: printSomeMessage( ). As in most other C-based languages, the na
the function is followed by a pair of parentheses that contain the function's paramete
(i.e. the function's data).
The body of the function is enclosed in a pair of curly braces and is invoked by writi
the name of the function, followed by a pair of parentheses and adding the required (i
paremeters.

Function Parameters:
In the printSomeMessage(message: String) example above, the function accepts one param
named 'message' of type: String.
Swift functions may accept 0 or more parameters. The parameters are enclosed within th
function's parentheses ( ). The name of the parameter is followed by a colon and the
```

```
parameter's type (similar to declaring a variable or constant in Swift).

By default, Swift's function parameters are constants (i.e. their values cannot change
To change this default behavior, the var keyword can be added to the parameter name in
function definition. Swift will then create a variable copy of the parameter's value fo
you to work with in the function's body.

Swift also allows different functions to have the same name provided their signature is
unique (simillar to function overloding in C++).
In Swift all functions have a type consisting of the parameter types and the return typ


// Function overloading example
// NOTE: In Swift, even the function's return value is significant in determination of
//       its signature!
*/
func printMessage(message: String) {                  // signature (String) -> ( )
    print(message)
}

func printMessage(message: String, times: Int) {      // signature (String, Int) -> ( )
    for i in 0..<times {
        print("\(i) \(message)")
    }
}

func printMessage(var message: String) -> Double? {         // signature (String) -> Dou
    // the "message" String is now mutable (i.e. can be altered) because the var keywor
    // added to the declaration
    message += "0"
    print(message)
    // precondition(Double(message) != nil, "supplied string cannot be converted to Dou

    /* In Swift, using a precondition tests to see if an operation can be performed leg
       and if not, prevents your code from proceeding forward. This is different from a
       assert( ) in Swift which simply checks if a variable contains a specific value o
       operation can be execuuted, but DOES NOT prevent your code from proceeding forwa
    */
    return Double(message) != nil ? Double(message)! : nil
}

if var rv : Double = printMessage("none") { // testing the optional return value from
   print("rv is: \(rv)", terminator: "");
}

printMessage("Hi Ho ", times: 3)

/*
// Swift argument label naming rules when invoking functions:
As of Swift 2.1, the language's defaults for the presence of argument labels have chang
are now simpler. The default behavior can be summarized as follows:

1. First parameters to methods and functions should not have required argument labels.
2. Other parameters to methods and functions should have required argument labels.
3. All parameters to initializers should have required argument labels
   (see maximum( ) function below).

In order to force users to explicitly require the use of labels when invoking functions
using the label name twice in the function definition is now required.
*/
func startHtml(title title : String) -> String {
   let text : String = "<html><head><title>\(title)</title></head>\n" +
                       "<body>\n"
   return text;
}

print(startHtml(title: "my webpage title...")) // in this function call, the "title" la

/*
// Swift In-Out Parameters:
In addition to using the var keyword to change the mutability of function parameters, S
also provides a mechanism for sending arguments that can modifiy the orignal variables
into the function by using the "inout" keyword.
When a parameter is declared as inout, the calling function must supply an & before the
variable's name (in much the same way references are created and initialized in C++ or
the way variable addresses are sent to functions in the C language).
*/

func addToString(inout dest : String, source : String) {
    dest = dest + source
}

var str = "Hello "
//           +------------- variable sent as inout (by "address")
//           |
addToString(&str, source: "World!")
print("str is now: \(str)")

/*
// Swift Function return values:
In Swift, a function's return type is preceded by the -> symbol and may consist of any
type or else return no value at all.
Actually, Swift functions without a defined return type still return a special value o
type Void by default.
This is simply an empty tuple, in effect, a tuple with zero elements, which can be
written as ( ).
Swift allows functions to return multiple values by specifying a tuple (list) return ty

*/
// required labels:     y:          z:              when invoking this function in your co
func maximum(x: Double, y: Double, z: Double) -> Double {
   var maximumValue = x // assume x is the largest to start
   // determine whether y is greater than maximumValue
   if y > maximumValue {
      maximumValue = y
   }
   // determine whether z is greater than maximumValue
   if z > maximumValue {
      maximumValue = z
   }
   return maximumValue;
}

print("Maximum of 3.3, 2.2 and 1.1 is: \(maximum(3.3, y:2.2, z:1.1))")
print("Maximum of 1.1, 3.3 and 2.2 is: \(maximum(1.1, y:3.3, z:2.2))")
print("Maximum of 2.2, 1.1 and 3.3 is: \(maximum(2.2, y:1.1, z:3.3))")

// omitting the function argument labels: y: and z: produces a compile error!
// let x : Double = maximum(2.2, 3.3, 4.4) // NOT PERMITTED
let x : Double = maximum(2.2, y: 3.3, z: 4.4)

func rollDice( ) -> (Int, Int, Int) {
   let die1 = Int(1 + arc4random_uniform(6)) // first die roll
   let die2 = Int(1 + arc4random_uniform(6)) // second die roll
   return (die1, die2, die1 + die2)
}

let dice = rollDice( )
print("die1: \(dice.0) die2: \(dice.1) sum: \(dice.2)")

/*
Swift functions may also return functions as a valid return value.
In the example below, the function compute( ) creates two embedded functions:
add( ) and subtract( ), each having a type of: (Int, Int) -> Int

The return value of compute must match this type: (Int, Int) -> Int

*/
func compute(x: Bool) -> (Int, Int) -> Int {
    func add(a: Int, b: Int) -> Int {
        return a + b
    }

    func subtract(a: Int, b: Int) -> Int {
        return a - b
    }
    return x ? add : subtract
}

let computeFunctionAdd = compute(true)
let resultAdd = computeFunctionAdd(1, 2)
print("result of function compute(true): \(resultAdd)")

let computeFunctionSubtract = compute(false)
let resultSubtract = computeFunctionSubtract(1, 2)
print("result of function compute(false): \(resultSubtract)")

// Swift function default values:
// Like C++, Swift allows functions to define default values for the parameters they ac

func printDate(date: NSDate, format: String = "YY/MM/dd") {
    let dateFormatter = NSDateFormatter( )
    dateFormatter.dateFormat = format
    print(dateFormatter.stringFromDate(date))
}

printDate(NSDate( ))
printDate(NSDate( ), format: "MMM-dd-yyyy HH:mm:ss a zzz")

/*
// Swift functions with variable (variadic) argument lists:
To define Swift functions that accept multiple numbers of arguments, the the ellipsis
must be included as the last argument and preceeded with the type of argument to accep
In order to specify a generic type, the <TYPE> directive can be specified after the
function name, but before the function's parentheses as:
   func functionName<TYPE>(members: TYPE...) {
      // your code here...
   }

*/

func variableArgs(members: Int...) {
   for i in members {
      print(i)
   }
}

variableArgs(4,3,5)
// variableArgs(4.5, 3.1, 5.6)                // not permitted given types Double
// vari("Swift", "Enumerations", "Closures")  // not permitted given types String

func variableArgs<TYPE>(list: TYPE...){
   for items in list {
      print(items)
   }
}

variableArgs(8.8, 3.14, -0.01953)                  // variable list of Double's now pe
variableArgs("Swift", "Functions", "Parameters")   // variable list of String's now pe

response = readLine(stripNewline: true)!

// Swift Arrays:

/*
struct Array<Element>
Array is an efficient, tail-growable random-access collection of arbitrary elements.
Swift arrays store ordered lists of values of the same type. Swift puts strict checkin
on arrays which does not allow programmers to enter a wrong type in an array
(even by accident).
Arrays are value types, meaning that they are copied when assigned, passed to function
or returned from functions.
An Arrays elements can be either value types or reference types. Every element of a
value-type Array contains a value of the Arrays declared element type. Similarly,
every element of a reference type Array is a reference to an object of the Arrays
declared element type. For example, every element of an Int Array is an Int value,
and every element of an Array of a class type is a reference to an object of that clas

An Array can be defined as a variable (with var) or a constant (with let). Arrays may
altered (changed or have elements added or removed from the list) only if the Array
was defined as a variable.

Syntax:
struct Array<Element>
An array is an efficient, tail-growable random-access collection of arbitrary
elements.

*/
var integers: Array<Int> = [ ]                     // an empty array of integers
var doubles: [Double] = [ ]                        // alternate array declaration forma

integers.append(6)                                 // adding values using the append(
integers.append(-1)
integers.append(8)
// integers.append(8.3)                            // error: cannot convert Double to
                                                   //        when appending to Array<I
doubles.append(2.3)
doubles.append(9.9)

print("integers array is: \(integers)")
print("doubles  array is: \(doubles)")

// The example below contains an array with mixed types. Is this even possible?
var mixedArray = [1, 2.2, 3, "Hello"]
var mixedArrayCopy = mixedArray
mixedArrayCopy[0] = 4

mixedArray.append(" World")

print("mixedArray=\(mixedArray), mixedArrayCopy=\(mixedArrayCopy)")

/*
The code above does in fact compile and run, but why? Arrays are supposed to
contain only homogeneous types of data!
When you declare an array that contains the values [1, 2.2, 3, "Hello"], Swift rules ou
Int, Double, and String in this case, because none of them to conform to ALL values wit
the array. The only type that would work is something that adheres to the IntegerLitera
FloatLiteralConvertible and StringLiteralConvertible types, but Nothing in the standar
library of types does that. So why does the example above work?
When you import UIKit or Foundation, it includes a type known as NSObject that Swift
can use to accomodate all of the different types listed and so that type is used.

// Array Initializers:
init( )
   Construct an empty Array.

init(_:)
   Construct from an arbitrary sequence with elements of type Element.

init(count:repeatedValue:)
   Construct a Array of count elements, each initialized to repeatedValue.

// Array instance variables:
capacity            // The number of elements the Array can store without reallocation
count               // The number of elements the Array stores.
endIndex            // A "past-the-end" element index; the successor of the last valid
first               // Returns the first element of self, or nil if self is empty.
last                // Returns the first element of self, or nil if self is empty.
indices             // Return the range of valid index values.
isEmpty             // Returns true iff self is empty.


// Array instance methods (functions):
mutating func append(newElement: Element)
   Append the elements of newElements to self.

func dropFirst( )
   Returns a subsequence containing all but the first element.

func dropFirst(n: Int) -> ArraySlice<Element>
   Returns a subsequence containing all but the first n elements.

func dropLast( )
   Returns a subsequence containing all but the last element.

func dropLast(_:)
   Returns a subsequence containing all but the last n elements.

func indexOf(_:)
   Returns the first index where predicate returns true for the corresponding value,
   or nil if such value is not found.

mutating func insert(newElement: Element, atIndex i: Int)
   Insert newElements at index i.
```

```
func maxElement(_:)
   Returns the maximum element in self or nil if the sequence is empty.

func minElement(_:)
   Returns the minimum element in self or nil if the sequence is empty.

mutating func popLast( )
   If !self.isEmpty, remove the last element and return it, otherwise return nil.

func prefix(_:)
   Returns a subsequence, up to maxLength in length, containing the initial elements.
   If maxLength exceeds self.count, the result contains all the elements of self.

mutating func removeAtIndex(_:)
   Remove and return the element at index i.

mutating func removeAtIndex(index: Int) -> Element
   Remove the element at startIndex and return it.

mutating func removeFirst() -> Element

mutating func removeFirst(_:)
   Remove the first n elements.

mutating func removeLast()
   Remove an element from the end of the Array in O(1).

mutating func removeRange(_:)
   Remove the indicated subRange of elements.

mutating func removeRange(subRange: Range<Int>)

mutating func replaceRange(_:with:)
   Replace the given subRange of elements with newElements.

func sort(@noescape isOrderedBefore: (Element, Element) -> Bool) -> [Element]
mutating func sortInPlace(_:)
   Sort self in-place according to isOrderedBefore.

*/

print(mixedArray.contains(2))          // check if the array contains the value 2

print(mixedArray.indexOf("Hello"))     // display the array index of element "Hello"

mixedArray[2] = 99                     // assign a new value to array at index 2

for i in mixedArray.indices {          // for in loop
    print("mixedArray[\(i)] is: \(mixedArray[i])")
}

var arrayOfInts = [1, 2, 888, -9, 0]  // inferred array of Int
arrayOfInts.append(3)
arrayOfInts += [4, 5, 13, 7, 18, -2]
print(arrayOfInts)

for (index, element) in arrayOfInts.enumerate( ) {
   print(String(format:"%5d%7d", index, element))
}

arrayOfInts.removeAtIndex(6)
arrayOfInts.removeLast( )


print("current  value of arrayOfInts: \(arrayOfInts)")
print("reverse  value of arrayOfInts: ", terminator:"")
for element in arrayOfInts.reverse( ) {
  print("\(element) ", terminator:"")
}


var cells = [Bool](count: 12, repeatedValue: false)
print("number of elements in cells: \(cells.count)")
cells[10] = true

// array ranges and accessing the description instance variable
cells[4...6] = [true, false, true]
print("cells: \(cells)")

let subset = cells[3...7]
print("subset of cells: \(subset.description)")

func myArrayFunc<T>(inputArray: Array<T>, element: T) -> Array<T> {
   var newArray : Array<T> = inputArray
   newArray.append(element)
   return newArray
}
/*
The function declared above accepts a generic type T, which is a placeholder. Because
has no requirements, T can be replaced by any type (when the function is called).
So the function could be called as:
*/

let newArrayOfInts = myArrayFunc(arrayOfInts, element: 1234) // calling the function
var sortedArrayOfInts = newArrayOfInts.sort(>)              // sort(>) descending '>'

for (index, element) in sortedArrayOfInts.enumerate( ) {
   print(String(format:"%5d%7d", index, element))
}

sortedArrayOfInts = newArrayOfInts.sort(<)                  // sort(<) ascending '<'
print("ascending order sort: \(sortedArrayOfInts)")
```