Seneca College of Applied Arts & Technology

PLAIN HTML VERSION: lecture2.161.txt.html
Xcode Project Archive: lecture2.161.zip

```
/*
file:    lecture2.161.swift
author:  Danny Abesdris
date:    January 20, 2016
purpose: MAP523AA/DPS923AA lecture #2
         Swift programming language:
         Overview: How an iPhone app works.
         String formatting, string functions, date functions and formatting,
         Swift numeric types, initializers, arithmetic, relational, and logical operat
         decision control, ternary operator, switch/case, tuples, enums, loops,
         random numbers, complete program example.
*/

import Foundation
import Darwin // required for arc4random_uniform( ) function

/*
How does an iPhone app work?

 _________________
[+---------------+]    _________ 4 ----<------<---------------<---------<-----------<-----
||   +------+    ||   /
||   |Label |___||_/                      ____    / Event Loop
||   +------+    ||      _ 1 -->-->--> /      \/
||               ||    /              |  iOS _/|                    +=====================
||   +-------+   ||   /               | (UIKit)|                    |
||   |Button |__||_/       _ 6 -->--> \      /                     | +-----------------+
||   +-------+   ||       /             ----                  0 ----| viewDidLoad( ) |
||               ||      /               |                         | +-----------------+
||      __       ||     |                | 2                       |          |
||     /  \      ||     |     +----------|--------------+          | +--------|----------
||     \__/      ||     +-<-<-| func touchDownEvent( )  |---3---> | | func processButtonI
[+---------------+]           +----------------------\--+          | +--------------------
  iOS Simulator                          |            \_____       +=====================
                                +--------|--------------+   \         View Controller
                                | func processNxtAction( )|    \
                                +-------------------------+     \______ 5 -<----<---<------

0. viewDidLoad( ) function is invoked only once after UI objects are loaded on the scr
1. User clicks/taps Button/Screen.
2. Touch event (Touch Down) message sent.
3. Message triggers @IBAction function processButtonPress( ).
4. Update UI elements (Label, TextField/View, Images, etc).
5. Return from func processButtonPress( ).
6. Done! Go to background/sleep until next event is triggered!

An app is essentially made up of objects that can send messages to each other.
Many of the objects in your app are provided by iOS, for example the button
(a UIButton object and TextField object) and a Label. These objects communicate by
passing messages to each other. When the user taps or clicks/touches the button on
the app, for example, that UIButton object sends a message to your view controller.
In turn, the view controller may message more objects.
```

```
With iOS, apps are event-driven, which means that the objects listen for certain events
occur and then processes them.
As strange as it may sound, an app spends most of its time doing... absolutely nothing
It just sits there waiting for something to happen. When the user taps the screen, the
app springs to action for a few milliseconds and then it goes back to sleep again until
the next event arrives.
You, the programmer, are required to write the source code that will be performed when
your objects receive the messages for such events.
In the app from Lab #0, the buttons Touch Down event is connected to the view
controllers processButtonPress( ) action. So when the button recognizes it has been
tapped, it sends the "Touch Down" message to your view controller.
*/

// String formatting, indexing, and Date manipulation

let dateFormatter = NSDateFormatter( )
dateFormatter.dateFormat = "MMM-dd-yyyy HH:mm:ss a zzz"
let date2 = NSDate( )
var dateString = dateFormatter.stringFromDate(date2)
print("dateString: \(dateString)")


var response : String = ""

let timeOfDay = dateString[dateString.startIndex.advancedBy(21)...dateString.startIndex
print("timeOfDay: \(timeOfDay)")
response = readLine(stripNewline: true)!

/*
mutating func append(c: Character)
mutating func appendContentsOf(other: String)
func substringFromIndex(index: Index) -> String
   [Foundation] Returns a new string containing the characters of the String
   from the one at a given index to the end.

func substringToIndex(index: Index) -> String
   [Foundation] Returns a new string containing the characters of the String
   up to, but not including, the one at a given index.

func substringWithRange(aRange: Range<Index>) -> String
   [Foundation] Returns a string object containing the characters of the
   String that lie within a given range.
*/

dateString.append(Character("x"))
dateString.appendContentsOf(timeOfDay)
let index : String.Index = dateString.startIndex.advancedBy(5)
let subStr1 = dateString.substringToIndex(index)
print("subStr1: \(subStr1)")

let subStr2 = dateString.substringFromIndex(index)
print("suStr2: \(subStr2)")

let subStr3 = dateString.substringWithRange(Range<String.Index>(start: dateString.start
                                                                end:   dateString.start
print("subStr3: \(subStr3)")


/*       Some NSDateFormatter string formatting options:

"y"      A year with at least 1 digit.
```

```
"yy"     A year with exactly 2 digits.
         If less, it is padded with a zero.
"yyy"    A year with at least 3 digits.
         If less, it is padded with zeros.
"yyyy"   A year with at least 3 digits.
         If less, it is padded with zeros.

"M"      A month with at least 1 digit.
"MM"     A month with at least 2 digits.
         If less, it is padded with zeros.
"MMM"    Three letter month abbreviation.

"MMMM"   Full name of month.

"d"      A day with at least 1 digit.
"dd"     A day with at least 2 digits.
         If less, it is padded with a zero.

"E",
"EE", or
"EEE"    3 letter day abbreviation of day name.

"a"      Period of day (AM/PM).

"h"      A 1-12 based hour with at least 1 digit.

"hh"     A 1-12 based hour with at least 2 digits.
         If less, it is padded with a zero.

"H"      A 0-23 based hour with at least 1 digit.
"HH"     A 0-23 based hour with at least 2 digits.

"m"      A minute with at least 1 digit.
"mm"     A minute with at least 2 digits.
         If less, it is padded with a zero.

"s"      A second with at least 1 digit.
"ss"     A second with at least 2 digits.
         If less, it is padded with a zero.

"zzz"    Time zone (3 letter abbr)
*/

// Swift numeric types:
/*
Type:            Storage size:          Bits:     Value range:
Int              4 or 8 bytes           16 or 32  -32,768 to 32,767 or
                                                  -2,147,483,648 to 2,147,483,647 (de:


Int8             1 byte                 8         -128 to 127
Int16            2 bytes                16        -32,768 to 32,767
Int32            4 bytes                32        -2,147,483,648 to 2,147,483,647
Int64            8 bytes                64        -9,223,372,036,854,775,808 to
                                                   9,223,372,036,854,775,807
UInt8            1 byte                 8          0 to 255
UInt16           2 bytes                16         0 to 65535
UInt32           4 bytes                32         0 to 4,294,967,295
UInt64           8 bytes                64         0 to 18,446,744,073,709,551,615

                                                    (negative range)
Float            4 bytes                32        -3.4028234663852886e+38 to
```

```
                                                    -1.40129846432481707e-45

                                                     (positive range)
                                                     1.40129846432481707e-45 to
                                                     3.40282346638528860e+38

                                                     (negative range)
Double              8 bytes                 64      -1.79769313486231570e+308 to
                                                    -4.94065645841246544e-324

                                                     (positive range)
                                                     4.94065645841246544e-324 to
                                                     1.79769313486231570e+308
Bool                true or false

The data types listed above are actually aggregate types declared as structs and conta
properties and methods (including initializers). Some initializers include:
init(Int)
init(Int8)
init(Int16)
init(Int32)
init(Int64)
init(Float)
init(Double)

init?(_ text: String, radix radix: Int = default)
(eg.)
*/
let x = Int("FF", radix: 16)       // sets x to 255
let y = Int("10000001", radix: 2)  // sets y to 129
print("x: \(x)")
print("y: \(y)")

/*
Arithmetic, Increment/Decrement, Compound, and Relational operators:
+, -, *, /, %, =
++, -- (pre and postfix, although prefix usage is recommended)
+=, -=, *=, /=, %=
<, <=, >, >=, ==, !=

Same usage as in C, C++, Java.


Decision making and control statements in Swift using if, if/else, if/else if/else:
Branching and decision making is performed in Swift in exactly the same way as other
languages (like C, C++, Java, etc) with the exception that enclosing brackets ( )
around test conditions are optional.
However, unlike other languages, Swift requires braces { } for every control statement
body, even if the body contains only one statement. This is one of several Swift
requirements that eliminate common errors that occur in other languages.
(eg.) Leap year determiniation:
*/

let year : Int = 2016
if year % 400 == 0 {
   print("\(year) is a LEAP year")
}
else if year % 4 == 0 && year % 100 != 0 {
   print("\(year) is a LEAP year")
}
else {
```

```
    print("\(year) is NOT a LEAP year")
}

/*
Ternary operator: lvalue = testCondition ? TRUE : FALSE
Like C and C++, Swift allows the use of the ternary conditional operator (?:) in place
of an if...else statement. This can make your code shorter and clearer.
The conditional operator ? : is the only operator that takes three operands. The first
operand (to the left of the ?) is a boolean expression (test condition), the second op
(between the ? and :) is the value of the conditional expression if the test condition
true and the third operand (to the right of the :) is the value of the conditional
expression if test condition is false.
(eg.) Leap year string (using conditional operator):

let strYear : String = (year % 400 == 0 || year % 4 == 0 && year % 100 != 0) ? "Leap Y

Switch/Case:
The switch conditional statement performs different actions based on the possible valu
of a control expression. However, unlike many other C-based languages, you can use val
of any type for the control expression, not simply integral types.

(eg.)
*/

let grade = 87.5
var letterGrade = "Invalid grade"

switch grade {
    case 90...100: // grade was 90-100
        letterGrade = "A"
    case 80...89: // grade was between 80 and 89
        letterGrade = "B"
    case 70...79: // grade was between 70 and 79
        letterGrade = "C"
    case 60...69: // grade was between 60 and 69
        letterGrade = "D"
    case 0...59: // grade was between 0 and 59
        letterGrade = "F"
    default: // grade was out of range
        break
}

/*
In the case statements (above), the ... represents the closed range operator
which represents a sequential collection of values within the ranges specified.
For example, the expression 90...100 epresents the sequential collection 90, 91,
92, 93, 94, 95, 96, 97, 98, 99, and 100.

NOTE: With the closed range operator ... both staring and ending values are included.
However, Swift also provides the half-open range operator ..< for which the
ending value in the range is not included in the sequential collection. Thus,
the expression 1..<8 produces seven values containing: 1, 2, 3, 4, 5, 6, and 7.

Other possible patterns for case:
A cases patterns may include:
A single value (or object) of any type.
A comma-separated list of values.
A closed range (using ...) or half-open range (using ..<).
Tuples (arbitrary lists).
Various combinations of these patterns in a comma-separated list.
```

```
With Swift's switch/case control, cases do not automatically pass through previous
conditions unlike other C-Based Languages.
In most C-based languages, without break statements at the end of each case, each time
a match occurs, the statements for that case and subsequent cases execute until a breal
statement is found. This is referred to as "falling through to subsequent cases.
Swift does not allow fall through unless you explicitly use the statement:

fallthrough

to explicitly allow control logic to pass through the specific case being evaluated.
*/

var temperature = 60
switch (temperature) {
   case 0...49 where temperature % 2 == 0:
      print("Cold and even")                 // NOTE: No explicit break is required, St
                                             //       leaves the case once examined. Ho
   case 50...79 where temperature % 2 == 0:  //       directive can be used to simulate
      print("Warm and even")
      fallthrough
   case 80...110 where temperature % 2 == 0:
      print("Hot and even")
   default:
      print("Temperature out of range or odd")
}

/*
Swift Tuples and Enums:
A tuple is an aggregate list of zero or more values represented within single
variable. The closest analog would be the C struct.
A tuple literal is a list of values separated by commas between a pair of parentheses.

For example, the tuple below:
*/
var address = (70, "Seneca College", "Toronto", "M3J 3M6", "The Pond Road", "Ontario")

// contains a list of Int and String types stored in a variable named "address".
// To access the contents of a tuple, the dot . operator is used with the elements
// referred to by numeric position starting at index 0.

let college : String = address.1
let city : String = address.2

print(college)
print(city)

// Using the dot operator, it is possible to change the values within a tuple if its
// declared as a variable.

address.2 = "North York"

// Tuples with named elements:
// Swift permits the elements within a tuple to be named. An element name is an identi
// followed by a colon : in place of having to refer to the elements by numeric positi

// For example, to name the elements of the address tuple above, you could write:

var address2 = (number:70, school:"Seneca College", city:"Toronto", postal:"M3J 3M6",
street:"The Pond Road", province:"Ontario")
let college2 = address2.school
let city2 = address2.city
```

```swift
let (myStreet, myPostal) = (address2.number, address2.postal)

/*
A tuple's type is determined by the values it contains. So, for example,
("tuple", 1, true) will be of type (String, Int, Bool).


An enum is an enumeration that declares a set of constants represented by identifiers.
An enumeration uses the keyword enum and a type name. As with a class, braces { and }
delimit the enums body. Inside the braces is a case containing a comma-separated list
enumeration constants. The enum constant names must be unique. However, unlike enums i
other C-based programming languages, a Swift enums constants do not have values by
default. The constants themselves are the values. It is possible for each constant to
a raw value specified explicitly.
If a variable has an enum type, you can assign enum constants to the variable using th
shorthand notation:

variableName = enum.ConstantName

It is an accepted guidline that enum constant names should begin with a capital letter
and use camel-case naming.
If an enum types constants represent sequential integer values, they can be defined as
a comma-separated list in one case, as in:
*/
enum Months: Int {
case January = 1, February, March, April, May, June, July,
   August, September, October, November, December
}
let myMonth = Months.February
let monthNum = Months.February.rawValue
print(myMonth)
print(monthNum)


// The raw values of an enums constants must be unique. In an enum with one of the int
// numeric types, if the first constant is unassigned, the compiler gives it a value o

// String enum example:
enum Provinces : String {
   case BC = "British Columbia", AB = "Alberta", SK = "Saskatchewan", MB = "Manitoba",
        ON = "Ontario", PQ = "Quebec", NS = "Nova Scotia", NB = "New Brunswick",
        NL = "Newfoundland and Labrador", PE = "Prince Edward Island"
}


// Loops:
let count : Int = 10
for i in 0...count {            // for in loop
   print("i is: \(i)")
}
print("============================")

for(var j=0; j < count; j++) {  // traditional C-style for loop (with parentheses)
   print("j is: \(j)")
}
print("============================")

for var k=0; k < count; k++ {   // traditional C-style for loop (without parentheses)
   print("k is: \(k)")
}
```

```
print("============================")

for i in 0..<5 {                    // for in loop with ranges a..<b (runs from a to b-1 i
   print(i)
}
print("============================")
for i in 0...4 {                    // for in loop with ranges a...b (runs from a to b inc
   print(i)
}
print("============================")

for var i in 0.stride(through: 10, by: 2) {
   // to: (up to, but not including) through: (up to AND including)
   print("in stride i is: \(i)")
}
print("============================")

let PI = 3.14159
var i2 : Int = 0
while i2 < 8 {                      // while loop
   print("\(Double(i2) * PI)")
   i2++
}
print("============================")

i2 = 0
repeat {                            // repeat/while loop (do/while in C/C++)
   print("\(i2)")
   i2++
} while(i2 < 8)


// capturing user input from standard input (similar to scanf( ) or gets( ) in C.)
print("Enter some data and press the enter key", terminator:"")
response = readLine(stripNewline: true)!
print("response is: \(response)")

                                    // enum representing game status constants (no ra
enum GameState {
   case KeepPlaying, Won, Lost
}

                                    // enum with Int constants representing common di
enum DiceRollNames: Int {
   case SnakeEyes = 2
   case Trey = 3
   case Seven = 7
   case YoLeven = 11
   case BoxCars = 12
}

                                    // function that rolls two dice and returns them
func rollDice( ) -> (die1: Int, die2: Int, sum: 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)
}

                                    // function to display a roll of the dice
func displayRoll(roll: (Int, Int, Int)) {
   print("Player rolled \(roll.0) + \(roll.1) = \(roll.2)")
```

```
}

                                    // play one game of craps
var myPoint = 0                     // points awarded if no win or loss on first roll
var status = GameState.KeepPlaying  // can contain KeepPlaying, Won or Lost

var roll = rollDice( )              // first roll of the dice
displayRoll(roll)                   // display the two dice and the sum

                                    // determine game status and point based on first
switch roll.sum {
                                    // win on first roll
   case DiceRollNames.Seven.rawValue, DiceRollNames.YoLeven.rawValue:
      status = GameState.Won
                                    // lose on first roll
   case DiceRollNames.SnakeEyes.rawValue, DiceRollNames.Trey.rawValue,
      DiceRollNames.BoxCars.rawValue:
      status = GameState.Lost
                                    // did not win or lose, so remember point
   default:
      status = GameState.KeepPlaying // game is not over
      myPoint = roll.sum            // remember the point
      print("Point is \(myPoint)")
}

                                    // while game is not complete
while status == GameState.KeepPlaying {
   roll = rollDice( )               // first roll of the dice
   displayRoll(roll)                // display the two dice and the sum

                                    // determine game status
   if roll.sum == myPoint {         // won by making point
      status = GameState.Won
   }
   else {
      if(roll.sum == DiceRollNames.Seven.rawValue) {
                                    // lost by rolling 7
         status = GameState.Lost
      }
   }
}
                                    // display won or lost message
if status == GameState.Won {
   print("Player wins")
}
else {
   print("Player loses")
}
```