Seneca College of Applied Arts & Technology

PLAIN HTML VERSION: lecture5.161.txt.html
Xcode Project Archive: lecture5.161.zip

```
/*
file:    lecture5.161.swift
author:  Danny Abesdris
date:    February 10, 2016
purpose: MAP523AA/DPS923AA lecture #5

             Swift programming language topics:
             Dictionaries, Table Views, Navigation Controllers
             TableView/Navigation GUI example.
*/

/*
Swift Dictionary Type:
A Dictionary is an unordered collection of key/value pairs.
A Dictionary associates values with keys just as a real-world dictionary associates
definitions with words, and are often referred to as associative arrays.
Keys must be unique, but multiple keys can have the same value. The keys must all
have the same type and the values must all have the same type, however the key type
can be the same as or different from the value type.

General syntax:
struct Dictionary<Key : Hashable, Value>

A hash-based mapping from Key to Value instances.
Also a collection of key-value pairs with no defined ordering.

Just as an Array, a Dictionary is a generic type. When you declare a Dictionary you
specify its key and value types. For example, Dictionary<String, Int> is the generic
notation for the type of a Dictionary with String keys and Int values. However, the
Dictionary type also can be written with the preferred shorthand notation [String : Int
NOTE: Swift types Array, String and Dictionary are all value types, so their objects
      are passed and assigned by value and copies normally made.
      Also, in Swift Dictionary types are implemented as a hash table.

Dictionary instance variables:
var count: Int
The number of entries in the dictionary.

var description: String
A textual representation of self.

var endIndex: DictionaryIndex<Key, Value>
The collection's "past the end" position.

var first: (Key, Value)?
Returns the first element of self, or nil if self is empty.

var indices: Range<DictionaryIndex<Key, Value>>
Return the range of valid index values.

var isEmpty: Bool
```

```
true iff count == 0.

var keys: LazyMapCollection<[Key : Value], Key>
A collection containing just the keys of self.
Keys appear in the same order as they occur as the .0 member of
key-value pairs in self. Each key in the result has a unique value.

var startIndex: DictionaryIndex<Key, Value>
The position of the first element in a non-empty dictionary.
Identical to endIndex in an empty dictionary.

var values: LazyMapCollection<[Key : Value], Value>
A collection containing just the values of self.
Values appear in the same order as they occur as the .1 member of
key-value pairs in self.

Dictionary Instance Methods:
func contains(_:)
Return true iff an element in self satisfies predicate.

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

func dropFirst(_:)
Returns a subsequence containing all but the first n elements.

func dropFirst(n: Int) -> Slice<Dictionary<Key, Value>>

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

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

func dropLast(n: Int) -> Slice<Dictionary<Key, Value>>

func elementsEqual(_:isEquivalent:)
Return true iff self and other contain equivalent elements, using isEquivalent
as the equivalence test.

func enumerate()
Return a lazy SequenceType containing pairs (n, x), where ns are consecutive Ints
starting at zero, and xs are the elements of base:

for (n, c) in "Swift".characters.enumerate( ) {
   print("\(n): '\(c)'")
}
0: 'S'
1: 'w'
2: 'i'
3: 'f'
4: 't'

func filter(_:)
Return an Array containing the elements of self, in order, that satisfy
the predicate includeElement.

func indexForKey(_:)
Returns the Index for the given key, or nil if the key is not present
in the dictionary.
```

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

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 popFirst( )
If !self.isEmpty, return the first key-value pair in the sequence of elements,
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.

func reduce(_:combine:)
Return the result of repeatedly calling combine with an accumulated value initialized
to initial and each element of self, in turn,
i.e. return combine(combine(...combine(combine(initial, self[0]), self[1]),...self[cou

mutating func removeAll(keepCapacity:)
Remove all elements.

Postcondition: capacity == 0 if keepCapacity is false, otherwise the capacity will
not be decreased.
Invalidates all indices with respect to self.
keepCapacity: If true, the operation preserves the storage capacity that the
collection has, otherwise the underlying storage is released. The default is false.

mutating func removeAtIndex(_:)
Remove the key-value pair at index.

mutating func removeValueForKey(_:)
Remove a given key and the associated value from the dictionary. Returns the value
that was removed, or nil if the key was not present in the dictionary.

Declaration
mutating func removeValueForKey(key: Key) -> Value?

func reverse()
Return an Array containing the elements of self in reverse order.

func sort(_:)
Return an Array containing the sorted elements of source according to isOrderedBefore.

func split(_:allowEmptySlices:isSeparator:)
Returns the maximal SubSequences of self, in order, that don't contain elements
satisfying the predicate isSeparator.

mutating func updateValue(_:forKey:)
Update the value stored in the dictionary for the given key, or, if they key
does not exist, add a new key-value pair to the dictionary.
*/

// Example:
// creating a Dictionary of type String:String
var airportCodes : [String:String] = ["yyz":"Toronto", "cdg":"Paris", "jfk":"New York"
```

```
print(airportCodes["yyz"]!)
// adding a key value pair to a Dictionary
airportCodes["lax"] = "San Francisco"

// using the updateValue method to replace a value at a specific key
var oldVal = airportCodes.updateValue("Los Angeles", forKey: "lax")
print(airportCodes["lax"]!)

// removing key/value pair "cdg":"Paris"
var removedValue = airportCodes.removeValueForKey("cdg")
print(airportCodes)

// looping through a dictionary
for (key, value) in airportCodes {
   print("Dictionary key \(key) -  Dictionary value \(value)")
}

// looping through a dictionary using the enumerate( ) method
for (key, value) in airportCodes.enumerate( ) {
   print("Dictionary key \(key) -  Dictionary value \(value)")
}

for key in airportCodes.keys {
   print("key: \(key)")
}

for value in airportCodes.values {
   print("value: \(value)")
}

// using the filter and reduce methods
var evens = [Int]()
for i in 1...10 {
  if i % 2 == 0 {
    evens.append(i)
  }
}

var evenSum = 0
for i in evens {
  evenSum += i
}
print(evenSum)

evenSum = Array(1...10)
    .filter { (number) in number % 2 == 0 }
    .reduce(0) { (total, number) in total + number }
print(evenSum)


// Dictionary with an array as value
let gradeBook = [
    "Susan" : [92, 85, 100], // name : Array of grades
    "Tom" : [83, 95, 79],
    "Natasha" : [91, 89, 82],
    "Robert" : [97, 91, 92]
]
var allGradesTotal = 0.0
var allGradesCount = 0

// using the reduce( ) method and a closure to compute
```

```
for (student, grades) in gradeBook {
    let total = Double(grades.reduce(0, combine: {$0 + $1}))
    print("AVERAGE GRADE FOR \(student): " + String(format: "%.2f", total / Double(grade
    allGradesTotal += total
    allGradesCount += grades.count
}
print("AVERAGE GRADE FOR ALL STUDENTS: " + String(format: "%.2f", allGradesTotal / Doul


/*
Table Views:
Table views are among the most used components of the UIKit framework and are an integ
part of the user experience on the iOS platform. Table views do one thing very well,
presenting an ordered list of items. The UITableView class combines several key concep
of Cocoa Touch and UIKit, including views, protocols, and reusability.

Data Source and Delegate:
As one of the key components of the UIKit framework, the UITableView class is optimize
for displaying an ordered list of items.
The UITableView class is only responsible for presenting data as a list of rows. The da
being displayed is managed by the table view's data source object, accessible through t
table view's dataSource property. The data source can be any object that conforms to th
UITableViewDataSource protocol.
The table view is only responsible for detecting touches in the table view and is not
responsible for responding to the touch events. To handle events, the table view must a
contain a delegate property that must notify its delegate whenever a touch event occurs
The table view's delegate is responsible for handling the touch event.

By having a data source object managing its data and a delegate object handling user
interaction, the table view can focus on data presentation. This organization results
in a reusable UIKit component that is in perfect alignment with the MVC
(Model-View-Controller) pattern used in all iOS applications.
The UITableView class inherits from UIView, which means that it's only responsible for
displaying application data.
A delegate object is delegated control of the user interface by the delegating object,
whereas a data source object is delegated control of the data.
The table view asks the data source object for the data it should display and is
therefore responsible for managing the data it sends to the table view.

Table View Components:
A UITableView instance is composed of rows with each row containing one cell, that bei
an instance of UITableViewCell.
Table views are only in charge of displaying data that is delivered by the data source
object, and detecting touch events, which are routed to the delegate object.
A table view is therefore nothing more than a view managing a number of subviews,
the table view cells.


Navigation Controllers:
iOS navigation controllers are one of the primary tools for presenting multiple screens
of content.
For displaying content across multiple screens a navigation controller is required, and
the UINavigationController class implements this type of functionality.

Like any UIViewController subclass, a navigation controller manages a view, an instanc
of the UIView class. The navigation controller's view manages several subviews,
including a navigation bar at the top, a view containing custom content, and an optiona
toolbar at the bottom. What makes a navigation controller special is that it creates a
manages a hierarchy of view controllers, often referred to as a navigation stack.

A navigation controller and a stack of (table) view controllers present an elegant way
```

```
to display nested sets of data across different scenes in an iOS app.

In addition to UINavigationController, a UITableViewController can be used to manage a
UITableView instance instead of a UIView instance. Unlike a UIViewController, Table vi
controllers automatically adopt the UITableViewDataSource and UITableViewDelegate prot
*/
```