CSCI S-65 Harvard Summer School Summer 2015

Assignment 1
Due: 6:30PM, 28 June 2015

READINGS

Swift Introduction, in order of depth:

One page promotional:
https://developer.apple.com/swift/

Unveiling at WWDC 2014:
https://developer.apple.com/videos/wwdc/2014/#402

20-30 minute "taste"
https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/GuidedTour.html

Extensive explanation of all facets of the language, to be used as needed by directed search for all assignments:
https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AboutTheLanguageReference.html

Xcode Introduction:
https://developer.apple.com/xcode/ide/

REFERENCES

Standard library functions (e.g., `map`, `println`, `reduce`)
https://developer.apple.com/library/prerelease/ios/documentation/Swift/Reference/Swift_StandardLibrary_Functions/index.html

REQUIREMENTS – PLEASE TAKE THE TIME TO READ OVER A FEW TIMES

A. All inputs and outputs must be rendered understandably and cleanly in the console output, clearly labeled by Exercise #. A penalty applies if "`Optional(…)`" appears in the output.
B. Because we have not yet covered reading information from the file system or other persistent stores, inputs may be hardcoded but the must be clearly labeled and static read-only members of a test-input `struct`.
C. There should be no magic constants, that is, arbitrarily chosen limits or values imposed from outside sources, left as literals. This means things like tax rates, maximum array lengths, and minimum acceptable values should be defined by name. See: http://stackoverflow.com/questions/47882/what-is-a-

magic-number-and-why-is-it-bad The recommended style in Swift is to group related magic numbers in a struct and name the struct helpfully, as a sort of "namespace" to give context to everything within. The constants are then declared as a series of **`static let`**s within. Suppose we were supporting more than one state:

```
struct MealTaxRates {
    static let MA = 0.07
    static let NH = 0.09
}

MealTaxRates.MA
// "Static" properties are members of the TYPE "MealTaxRates"
// rather than members of a particular INSTANCE.
// So, we need not declare an instance of the struct.
// Just using a nicely named constant.
```

Commentary: Interpret this with fluidity that adapts to code efficiency. For example there is some data that benefits from access by a data structure such as a Dictionary or Array. In this case the entire Dictionary/Array can be a "static let". Consider:

```
struct EnglishToFrench {
  static let Fruit = [ "Apple": "Pomme", "Pineapple": "Ananas" ]
  ...
}
```

D. Include test cases of out-of-range values that cannot be prevented at compile time. Include the obvious border cases of an empty array(/dictionary), a single element array, and a 2-element array for all problems involving arrays and dictionaries. Include the obvious border cases of the minimum and maximum input values when applicable.
E. Leverage optional types when appropriate, such as when an input is invalid and hence the output cannot be calculated.
F. Use safe unwrapping with **`if let`** when Optional types are being used and you cannot prove that the type is non-nil at that point.
G. When constructing string values, leverage the **`\(...)`** interpolation syntax for brevity & clarity.
H. Unlike the example outputs in the problems, console output should always be in clear, complete, helpful English sentences.

PROBLEMS. 270 pts total.

0. (6 pts) Get set up.
   a. Register as an Apple Developer to be able to join development teams (https://developer.apple.com/register/).

b. Download & install (or update to) the latest version of Xcode 6.3 (which includes Swift, the iOS libraries, the iOS simulator, the debugger, the Xcode UI, and related tools.) **Ignore Xcode 7 and Swift 2.0 and anything in beta.**

c. Update your computer to the latest release of Yosemite (10.10.x). Update all your iOS devices to the latest version of iOS 8.x. Keep up all with security & stability updates. We must all be working with identical platforms (read: bug-for-bug compatible. DON'T install beta releases on anything used for class, nor Mac OS 10.11 nor iOS 9.)

d. Make sure Xcode starts up and get help from us immediately if you have trouble.

e. Promise to make responsible hourly and daily backups to Time Machine, iTunes, and cloud storage. Absolutely no extensions will be granted for data loss regardless of the manner of loss.

For all non-code problems (#1, #9, #11, #13, #16) number them clearly along with your name, and put them in a written document in any common format (e.g. PDF, Word, Pages, HTML) at the top level of the directory structure of your submission.

For all quantities representing money, use integer pennies for internal storage and only format them with a decimal point when time for output formatting. The following should help:

```
var nf = NSNumberFormatter()
// The default locale of the device is used
nf.numberStyle = .CurrencyStyle
println(nf.stringFromNumber(3.7)!)
```

1. (6 pts) What are your (min 3, max 5) favorite sources of technical information on the Internet, especially related to the course? What is a downside that you've experienced with each?

Problems 2-14 are entirely Console-based, or Terminal-bound, mini-programs. They do not involve iOS, nor do they have a graphical UI. They only use println() for output. The output will appear in the lower right area of Xcode, the console.

For management purposes, make a single Xcode Project / single App for exercises 2-8 and sequence all functions into a single main thread of control that executes them in the order laid out below.

However, each exercise & its related support code should go in a separate `.swift` file in the Project (but all in the same directory) to get practice navigating and pulling together multiple source files.

2. (3 pts) Write a one-line program than prints “Hello, World” to the console.

3. (10 pts) Use **`map`** to write a function that accepts an array of integers, each specified to be in the range 0...99, and produces a corresponding array of printed representations in English words for each, as strings. Remember the guidelines for invalid inputs.

   Example call: `intArrayToEnglishWords([7, 2, -13, 300, 6, 26])`
   Output: `["seven", "two", nil, nil, "six", "twenty-six"]`

4. (10 pts) Use **`reduce`** to write a function that takes a list of items ordered at a restaurant and computes the total bill.

   Inputs (function parameters): a dictionary consisting of items and their costs, with **`String`** keys and **`Int`** values. Don’t allow **`nil`** values (i.e. values should not be Optionals).
   Output (return value): the bill total in pennies, as an integer, including meal tax and 15% tip. You should round to the nearest integer, not truncate. (For this and all restaurant problems, the tip shall be calculated on the base total, not including tax, and the rounding convention is always to the nearest integer.)

   Example call: `sumOfItemCosts(["Ham": 345, "Cheese": 115])`
   Output: `561`

5. (10 pts) Write a function that uses **`filter`** to produce a new array of exactly the primes contained in an input array. (Primes must be greater than or equal to 2; ignore negative numbers). Use a helper function that takes an **`Int`** parameter and returns a **`Bool`** to determine if an input is prime. Provide 2 implementations, where the helper function is alternately:
   a. An extension of the **`Int`** class
   b. An inner function of your main function

   Example call: `primesOf([3, 1, -3, 27, 29, 3, 7, 2, 7, 83, 91])`
   Output: `[3, 29, 3, 7, 2, 7, 83]`

   Definition: An inner function means a function that is declared within another function. Example:

```
func outerFunction() {
      func innerFunction() {
            // innerFunc implementation
      }
      // innerFunction only has scope from here
      // to end of outerFunction,
      // but otherwise acts like any other function
}
```

6. (12 pts) Building on Exercise 4, add a tip calculating aspect. Define a global **`enum`** that encodes 3-5 discrete levels of service (e.g. poor, good, excellent). Let the **`rawValue`** of each service level be an appropriate tipping percentage of your choosing, expressed as a floating point value between 0 and 1. Create an extended version of the function that accepts a second parameter of that **`enum`** type. The total returned by this function should include the tip. You may reference the globally defined test cases from #2.

   Example call:
   ```
    totalInclTipWithItems(["Ham": 345, "Cheese": 115],
         serviceLevel: ServiceLevel.Good)
   ```
   Output, assuming Good means 18%: **`575`**

   (Self-check: why is "**`ServiceLevel.Good`**" unnecessarily verbose in the above context?)

7. (10 pts) Building on Exercise 4, make the return value an instance of a globally defined **`struct`** with 3 components: base food cost, tax, and tip, and make the **`total`** be a *computed* property of that **`struct`**. Use local meal tax of 7.0% (for trivias' sake: This is Mass State Sales Tax of 6.25% plus 0.75% meal tax). Also set up a description property that outputs a printable breakdown and summary (all four values). You may reference the globally defined test cases from #2. Leverage the computed properties in your output.

   Example call: Same as in #5
   Output:
   ```
   { baseFoodCost: 460, tax: 32, tip: 83, total: 575 }
   ```

   Example of a struct declaration that contains a computed property:

   ```
   struct AbcStruct {
       let member1: Int
       let member2: String
       var combinedMembers: String {
           return "\(member1) - \(member2)"
       }
   }
   ```

   Example of how to instantiate such a struct:
   ```
   var abcInstance = AbcStruct(member1: 4, member2: "Hello")
   ```

8. **(40 pts) Object Orientation:** The above classes are of the old-fashioned "batch processing" style: a pre-defined input is fed into a function and output is created. Now, create a **`class RestaurantBill`** which tracks incremental changes to a restaurant bill. As items are added incrementally, the class should be capable of reporting the current total, current item, and other aspects of the check. All required details are below. Note that the final total is implemented much differently than in #6; it is a computed property there, while here it is a stored property that keeps up to date by a property observer on other stored properties that affect it.

   Implement the member functions and properties:

   `addLineItem()` – method that takes a description, quantity, and price per item and stores it in the check. If an item of exactly that description already exists in the check, the item count is instead incremented.

   `description` – computed property that produces a line item invoice, with each line showing: item description, quantity, and line item total, with a base total at the end (but no tax or tip included). Use the appropriate property from **totals** below. You can use the '\n' escape sequence for a newline just like many other languages. If desired, you can use the `spacePad` method from line 62 of **`TerminalIO.swift`**. Example usage is shown in the comments there. It is actually a custom 'extension' method called directly on an Int, even a literal! [As in: `56.spacePad(5)` returns `"   56"`]

   `serviceLevel` – stored property of enum type as in #4. Must be an observed property that, when set, forces **finalTotal** to recompute (see description below). It is up to you as to whether you prefer an intermediate stored **tip** property that recomputes itself by observing changes to **serviceLevel**, in which case the **finalTotal** would observe **tip** instead.

   `baseTotal` – stored property representing total without tip or tax. Because it is stored and not a computed property, you keep it up to date with respect to the mutating function **`addLineItem()`** above.

   `finalTotal` – Also a stored property representing total including tip and tax. Because it is stored, you must observe changes the properties that affect it and recompute it when they do.

The next set of exercises applies to the **`TerminalFib`** project shown in class. Make a copy of the project and turn in your updated project.

9. (5 pts) What goes wrong when the **`IntCacheable`** protocol declaration of **`LimitedIntCache`** is left out? How about **`Printable`**? How are these two types of mishaps fundamentally different?

10. (5 pts) Move as much as possible out of the **`init()`** method. Set default values for remaining parameter(s).

11. (5 pts) Why is **`maxSize`** declared as a **`let`** but the other class member declared as **`var`**?

12. (7 pts) Rewrite **`findIndexOfOldest()`** as a one-liner using as much syntactic sugar as possible.

13. (6 pts) The subscript function is defined to return an Optional **`Int`**. What specific fact(s) can you deduce about the current state of the cache if **`nil`** is returned for a particular key **`k`**?

14. (25 pts) Making a new class that also adopts the two protocols, implement a least-frequently-used replacement strategy instead of a least-recently-used. You'll need a simple frequency count, which will be used in a struct analogous to **`AgedEntry`** but instead called something like **`FrequencyEntry`**. Ties may be broken arbitrarily; that is, if two cached values are equally deserving of eviction because they are both the least frequently used, either one may be removed.

StoryBoard Section

15. (90 pts) **YOUR FIRST ACTUAL APP. Up until now, we've only created Terminal-bound programs to help you learn Swift, where the output appears in Xcode's console window. Now, we're going to pull it all together into something that actually runs on the iPhone (and iPhone Simulator in your development tool suite.)** Create a simple iPhone app that implements a simplified interface to the **`RestaurantBill`** class. Start a new Single-View iOS App Xcode project. Design the following in StoryBoard.

    a. Add 2 food items and 1 drink item as **`UILabel`**s (descriptions should be short and hardcoded)
    b. Next to each label, add a **`UITextField`** that accepts the price in dollars and cents (as in 4.56 or 0.30 or .56)
    c. Next to each text field, add '+' and '-' buttons to adjust the quantity
    d. Next to the '+'/'-' buttons, show the current quantity
    e. EXTRA CREDIT (10 more pts): Below the 3 items, add a **`UIPicker`** or series of checkboxes to select the service level. (If you decline the

extra credit, leave the ServiceLevel as a hard-coded **`UILabel`** just to show what it's using.) Extra credit points count equally toward your overall omework point total.

f. Below the **`UIPicker/UILabel`**, show the summary, lay out however you see fit, but all clearly labeled, as a series of **`UILabel`**s:
   i. The tip
   ii. The tax
   iii. The food/drink total
   iv. The final total

Functionality: Everything in f.) should update instantly & correctly as the '+' and '-' buttons are pressed. The price entry fields should use the "Decimal Pad" keyboard type.

Design considerations:

The **`RestaurantBill`** class is the model and should be declared in a separate source file. When the UI changes and hence the **`UIViewController`** receives the event, it should update its contained **`RestaurantBill`** instance appropriately: by either calling **`addLineItem()`** or modifying **`serviceLevel`** when the buttons are tapped or the picker is rotated, respectively. The event handlers should then query the model instance for the updated totals (in part f. above) and update the UI appropriately.

The decimal keyboard will still allow invalid floating point numbers to be entered. If the number is unparseable, assume the value is $1.00. Other more sophisticated error checking and/or input constraining is important and will be covered later.

DISCUSSION:
The functionality paragraph represents the cornerstone of Graphical User Interface design – and is often the biggest stumbling block. User input must be captured, reflected in the underlying data model, and pushed as fast as possible to the display. For now you may put all of this "round-trip" within the single event handler method. Note that it makes for a fragile architecture that won't withstand the complexity of large real-world applications. However, all solutions are "hard" and involve other trade-offs, especially up-front complexity. There will be several design patterns to apply to this idiomatic flow. For now just learn (memorize!) the flow:

A) user manipulates UI (that you laid out in Storyboard, then all user events captured by iOS at runtime) →
B) iOS interprets the user events and calls the relevant view controller event handler that you registered in Storyboard and that you implemented) →

C) Your event handler updates the internal state (the model) →
D) Your updates to the model propagate to visually update any UI elements that reflect the model (e.g. the total bill)

16. (20 pts) There are three ways of managing memory for objects that are allocated and then no longer needed:

- Manual free & release
- Garbage collection
- Automatic Reference Counting (ARC)

Swift uses the 3rd. Do a little research and contrast each of the 3. **CITATIONS ABSOLUTELY REQUIRED EVEN IF YOU ALREADY KNOW BY PRACTICE.** Write 4 sentences for each:

- A major language that uses it as its principal or only method, besides Swift
- How essentially each one works, especially the division of responsibility; between the language runtime and the programmer;
- A major disadvantage, relative to the others;
- A major advantage.

The advantages and disadvantages must be relative to the others. Google-level scholarship is fine but do not copy directly: write on a blank sheet with no sources visible to ensure you use your own words.

To understand this, you may need to do some review or background reading on the difference between the *stack* (where function calls are piled onto each other, along with local function parameter values, and is much simpler to understand and requires to management technique), and the *heap* (which is the persistent object area that is managed by the above techniques and often used to store objects that exist over a significant portion of the application's lifetime). Every structured programming language from Pascal to Ruby to C++ to Lisp revolves around these concepts, whether they are made explicit or not. They bear review!