**Fall 2018**

**CSCI E65g: Mobile Application Development Using Swift and iOS**

**Assignment 2**

**Issued: 09/11/2018 Finalized: 09/13/2018 Due: 09/18/2018**

**Purpose:** Design a small system with Object-Oriented Swift while continuing to gain fluency in lower level concepts.

**Scoring:** Out of 200 points (Undergraduate and Graduate)

**Project Structure:** Same as Assignment 1.

**Output:** Same as Assignment 1.

**Reading:** Lecture 1 & 2 playgrounds, Apple’s Swift iBook as reference (links embedded below)

**Problems**

1. (0 points) Create your [Assignment 2 repository](https://classroom.github.com/assignment-invitations/dffdee35e04e2ec3eed8b5ebacbe66f2)
2. (25 points) Use the map method of the Array class to write a function englishWordsOf(values: [Int]) -> [String?] that accepts an array of integers and produces a corresponding array of printed representations in English words. If the number falls outside of the range [0...99], compute nil instead. This is a great opportunity to practice with the Dictionary type to keep your code compact. Helper functions and Object-Oriented style (encapsulating in a struct) are permissible.

**Do not consult reference materials for generating this algorithm and don’t use built-in conversion functions that do this for you (specifically**[NumberFormatter](https://developer.apple.com/documentation/foundation/numberformatter?changes=_2)**). Part of the essence of this problem is to compose the underlying logical idea and express it in the Swift primitive operations. Try to make it efficient; if you have 99 separate cases that’s missing the intent.**

Example call: englishWordsOf(values: [7, 2, -13, 300, 6, 26])

Computed result (this is the actual value, not to be rendered on screen; you’ll need to format each item for output at the last step, otherwise Optional(...) will print):  
["seven", "two", nil, nil, "six", "twenty-six"]

Note here that this is an Array of Optional Strings. If we remove the syntactic sugar to reveal the explicit values, what Swift really makes of this is:

Array<Optional<String>>: [ Optional<String>.some("seven"), Optional<String>.some("two"),

Optional<String>.none, Optional<String>.none, Optional<String>.some("six"),

Optional<String>.some("twenty-six") ]

Formatted output:

\*Problem 1\*

7: seven

2: two

-13: no answer

300: no answer

6: six

26: twenty-six

1. (7 points) Write a function primesOf that uses the filter method of the Array class to produce a new array of exactly the primes contained in an input array, using your isPrime computed property from Assignment 1.

**Input**: An array of integers.

**Output**: An array of integers that contain exactly the positive primes of the input array, in order and without removing duplicates.

Remember that prime numbers, by definition, are greater than 1, and that 2 is prime. Ignore zero and negative inputs: you can skip right over them as your filtering progresses.

Example call:  
primesOf(values: [3, 1, -3, 27, 29, 3, 7, 2, 7, 83, 91])  
Computed Result:  
[3, 29, 3, 7, 2, 7, 83]  
Formatted Output:

\*Problem 2\*

The following values are positive prime: [3, 29, 3, 7, 2, 7, 83]

1. In the rest of the problems, you will gradually develop a system for taking orders at a restaurant with a traditional system of a menu and a bill per table.

(15 points)

Reading: [reduce for Dictionaries](https://developer.apple.com/documentation/swift/dictionary/2295354-reduce). Note this is an important variation on [reduce for Arrays](https://developer.apple.com/documentation/swift/array/2298686-reduce). Learn it for Arrays first and the numberSum example in particular.

Use the reduce method to write a function totalCostOf that takes a list of items ordered at a particular table and computes the total bill.

Inputs (function parameters): a dictionary consisting of items and their costs, with String keys and Int values.

Return value: an Int containing the bill total, in pennies, including a net total meal tax of 7%. It is easiest (but technically, not necessary) to dip into the world of Double and back out again to handle percentages (see [the Int conversion constructor](https://developer.apple.com/documentation/swift/int/1538869-init).) The default rounding behavior is fine.

Example call:  
totalCostOf(items: ["Ham": 345, "Cheese": 115])

Computed result: 492

Correctly formatted output (using one of the formatters from Assignment 1):

\*Problem 3\*

Your bill including tax of 7.0% comes to $4.92.

1. (8 points) Building on Exercise 3, add a tip calculating aspect using TippingLevel from Assignment 1. Create an *overloaded* version of totalCostOf that accepts a second parameter withTipLevel of that enum type. The total returned by this function should include the tip. For the sake of consistency, all restaurant-related problems for the whole course will calculate the tip on the base total, not including tax.

Example call (where .OK = 15%):  
totalCostOf(items: ["Ham": 345, "Cheese": 115], withTipLevel: .OK)  
Return value: 561  
Correctly formatted output:

\*Problem 4\*

Your bill including tax of 7.0% and tip comes to $5.61.

1. (20 points) Building on Exercise 4, change the return value from Int to a custom struct BillSummary (which you must declare beforehand, globally) with 3 stored properties all denominated in pennies: base food cost, tax, and tip. Add a 4th *computed* property total (also pennies). Again assume meal tax of 7.0%. Remember from the Assignment 1 reading, the **Memberwise Initializers for Structure Types** makes construction easy without writing your own constructor.

Adopt the [CustomStringConvertible](https://developer.apple.com/documentation/swift/customstringconvertible) protocol and implement the required description property (computed), that outputs a printable breakdown and summary (all four other values). Use it for the final output. Remember that "\(x.description)" is redundant for adopters of this protocol; "\(x)" is equivalent (and that’s the point).

Example call (where .Good = 18%):  
totalCostOf(items: ["Ham": 345, "Cheese": 115], withTipLevel: .Good)

Return value: A struct BillSummary instance containing: { baseCost: 460, tax: 32, tip: 83 } where total would evaluate to: 575, and description would evaluate to: "base: $4.60\ntax: $0.32\ntip: $0.83\ntotal: $5.75\n"

Correctly formatted output:

\*Problem 5\*

Your bill summary:

base: $4.60

tax: $0.32

tip: $0.83

total: $5.75

1. (40 points) Reading: Classes portion of [Structures and classes](https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html); [Properties](https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html) with attention to stored vs. computed and didSet observers.

Object Orientation: The above problems use the old-fashioned “batch processing” style: a pre-defined input is fed into a function and output is created. Now, we will begin to create an object oriented system.

The first thing to note is that there are really two entities at play here: a *menu* (which changes very slowly, and just one per meal-time for all customers), and a *bill*, which is unique per customer table and changes as the order is augmented. We don’t want to repeat pricing data all over the place in many bills. (If you know UML, this is a rough start to an [entity relationship diagram](https://www.visual-paradigm.com/VPGallery/datamodeling/EntityRelationshipDiagram.html).) Also, don't confuse an internal *bill object* (which does not store pricing data) with its *printed representation*(which will show prices for human convenience only).

Implement a class RestaurantMenu which tracks items by name, description, and price. It should be possible to add and delete items, and ask a menu the price of any item given its name, or be informed there is no such item. You must use a helper struct MenuItem that aggregates the 3 essential properties of an item. Because the name is the unique identifier (and based on the API below), it must be a non-mutable property of the struct while the other 2 must be mutable.

LAST MINUTE CORRECTION/ADDITION 2018-09-17: Later, it will be important to compare them for equality, which is a very easily *synthesized* capability as long as we declare conformance to the Equatable marker protocol: See the **Conforming to the Equatable Protocol** paragraph in the [reference](https://developer.apple.com/documentation/swift/equatable).

The below implies at least a 2-case enumerated type for the Error object.

Implement the members:

addItem(name: String, description: String, price: Int) throws

Stores the price of name (in cents as usual), replacing the price and description if the item already exists. Throws an exception if the price is invalid (less than 25 cents or more than 1000 dollars).

removeItem(name: String) throws

Removes the entry associated with name. Throws an exception if the item does not exist.

lookupItem(name: String) -> (String, Int)?

Returns a tuple containing the description and price of the desired item, or nil if not found. Does not throw any exceptions.

Write enough test cases to cover all the exceptions and the two normal cases (replacing vs. new). The normal cases should be proven correct by looking up and printing the result before, in-between, and after replacement. As usual your error cases should all be caught and should print human-readable errors.

1. (6 points) Reading: [Protocols](https://docs.swift.org/swift-book/LanguageGuide/Protocols.html)

Extract the above method signatures into a protocol named MenuProtocol and make your class conform. Keep this and all other protocols in your Shared.swiftfile.

1. (3 points) Declare and initialize a single, global instance of your menu in Shared.swift for use below.
2. (60 points)

Implement a class RestaurantBill which tracks the current order of a given table. Implement the members:

addItem(name: String, quantity: Int) throws

Adds *quantity* items matching *name* from the aforementioned RestaurantMenu instance. If some of that item are already in the bill, this quantity should be added in instead of overwriting. Throws an exception if *name* is not found in the menu, or if *quantity* is negative or excessive (more than 50).

removeAllItems(name: String) -> Int

Removes all of the items named by the parameter and returns the count of that item just before removal. Does not throw an exception if not present; merely returns 0.

serviceLevel: TippingLevel

A stored property of the TippingLevel enum type.

baseTotal: Int

A computed property representing total without tip or tax, in integer pennies. You must use the reducefunction on Array or Dictionary for this.

finalTotal: Int

A stored property representing total including tip and tax (why do we always represent dollar figures in pennies?). Since it is stored, you must observe changes to other stored properties that affect it and recompute it when they do. This may not feel natural, but it is included to drive home the idea of property observers.

description: String

Computed property that produces a line item invoice. A line item invoice merely aggregates multiple items of the same kind (name) on one line with detail: name, quantity, and subtotal for that line. The order of items here is not important. The bottom must include the base total, tax, tip, and final total as well on separate lines, in that order. Again leverage the property in String interpolation by adopting CustomStringConvertible.

Take the same approach to Exceptions as in the menu. Use a separate enum.

Example:

myMenu = RestaurantMenu()

myMenu.addItem(name: "Turkey sandwich", description: "Delicious oven roasted turkey in yesterday's bread", price: 926)

myMenu.addItem(name: "Tofu wrap", description: "Fermented soy goodness in a tasty wheat tortilla", price: 749)

myBill = RestaurantBill()

myBill.addItem(name:"Turkey sandwich", quantity: 2)

myBill.addItem(name: "Tofu wrap", quantity: 1)

myBill.description would evaluate to:

Tofu wrap (1 @$7.49): $7.49

Turkey sandwich (2 @$9.26): $18.52

Base Total: $26.01

Tax @7.0%: $1.82

Tip, OK @15%: $3.90

Final Total: $31.73

Have at least 6 test cases, where you probe edge conditions, such as an empty bill, or unreasonable values (-3 salads, 0 hamburgers at $4.99 each, a pickle that costs -0.25, an [apple from 2014](https://www.apple.com/mac-mini/))

Because the code is getting more complex, one test should be multi-step and make incremental changes (e.g. just the tipping level or adding one item or changing one aspect of one item) for a modular approach. Print out the full bill after each mutation.

1. The next 3 questions explore the subtleties of *identity* and *equality*.

(8 points) Change your global menu variable to a let. Can you still mutate the contents? Why is that? What can you not do now that it is a let?

1. (4 points) If you built two menus with identical contents, using separate calls to the constructor and to addItem for each, why can they not be compared with == ? Explain the compilation error: what’s missing? What about if you compare them with =​== ? (Experiment and) explain the second result.
2. (4 points) Same question, except apply to two identical MenuItem structs, where the situation is reversed: == is possible but === is not.