**Fall 2018**

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

**Assignment 1**

**Issued: 09/04/2018 Due: 09/11/2018**

**Purpose**: Learn the basics of the Swift (4.1) language in isolation, independent of iOS.

**Scoring**: Out of 175 points (both Undergraduate and Graduate); the points assigned to each problem reflect a combination of the difficulty and importance.

**Project Structure**: Create a **Terminal-based** App in Xcode (File Menu → New Project → OS Application → Command Line Tool). *This is the only time we will create a terminal-based application that runs on your Mac, not on the iPhone or iPhone simulator.* All code is dispatched from main.swift. Note that when this project is built, it can indeed run directly from Terminal, just like any Unix or Mac OS X text command.

**Output**: All output must be rendered understandably and cleanly in the console in complete sentences, clearly labeled by problem number – which takes a little bit more code than just writing the requested function. Do not let Optional(…) appear in printed output. Not all problems result in output.

**Inputs**: Because we have not yet covered reading information from the file system or other persistent stores, inputs should be hardcoded but labeled (see coding style).

**Coding Style** Study and absorb [my full list](https://harvard-ios.github.io/csci-e65g-2018/coding_style.html). Below is a condensed version specific to this assignment.

1. Xcode (the compiler) should generate no warnings or errors in the submitted work. Points will be deducted for each. This is not punitive; there is strong academic value in finding and fixing all warnings and errors from the beginning.
2. For problems that require test input, each test input value must be a thoughtfully named let constant. This is to practice avoiding Magic Constants right away.
3. Leverage optional types when appropriate, such as when an input is invalid and hence the output cannot be calculated.
4. Use safe unwrapping with if let … when handling possibly nil Optionals.
5. When constructing string values, leverage the \(…) interpolation syntax for brevity & clarity.
6. Modularity: Each problem should go in a separate file named Problem*N*.swift. All code relating to a problem should be isolated to that problem’s source file. If the problem contains code that runs, put it in a global function called runProblem*N* and invoke it from main.swift. If you find multiple problems would benefit from sharing code or input data, you may place those in Shared.swift.
7. If the problem simply asks you a question, put the answer in a multi-line comments labeled with question number and sub-part, e.g. /\* 1.a) ... \*/
8. Avoid repetition. Use functional abstraction (create reusable functions) and iteration instead. This becomes much more important later on.
9. Avoid unnecessary syntax where Swift infers things for you: e.g. type declarations, semicolons, and parentheses. Again, not punitive even though flowery/verbose code works; inference must be learned right away.
10. Use camelCase and fullWordVariableNames, not snake\_case or obscr\_abbrvns.

**Testing**: Include test cases of out-of-range values that cannot be prevented at compile time. For example, when given a problem involving an array, test with 0-length and 1-length arrays. Generally, whenever there’s a given bound, test with values or data structure sizes on the bound, just under it, and just over it. If the problem specification indicates some kind of error handling or special-casing, be sure to trigger those cases.

**Readings and Videos**

* [Lecture 1 playground](https://github.com/harvard-ios/csci-e65g-2018/tree/master/lectures/01)
* Apple’s Swift iBook: See embedded links below
* [Emacs tutorial (Optional, for working with Swift REPL)](https://www.dropbox.com/s/g4f9yxq97w3wxyv/emacs-tutorial.mov?dl=0)
* [Exceptions tutorial (includes forEach w/ closure, array-driven testing)](https://www.dropbox.com/s/y8s4hudjug60iex/exceptions-tutorial.mov?dl=0)

**Problems**

1. (0 points) Create and clone your personal assignment 1 repository with [this Assignment 1 invitation link](https://classroom.github.com/assignment-invitations/2a33481dd196dcc8414f1d6f409a9cf5) (careful not to nest your local copy within another repository! Use cd (change directory), pwd (print working directory), and ls -la (list detailed directory contents, including dot-prefixed files) to ensure your directory structure is clean and systematic.)
2. (23 points) (Reading: [The Basics](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html) sections **Integers** through **Numeric Type Conversion**). You may use the Playground or Swift REPL for the following two problems. What is the inferred type of the following expressions? For those that involve operators, what is the value? For those at are errors, fix them and then give the type.
   1. 0
   2. "0"
   3. '0' (Hint: Individual symbols or characters are [nothing like as in C](https://developer.apple.com/documentation/swift/character) and related languages.
   4. 0.0
   5. "0.0"
   6. "0.0".count
   7. 3 + 4 \* (7 - (6 \* (12 / 5)))
   8. 3 + 4 \* (7 - (6 \* (12 / 5.0)))
   9. 3.0 + 4.0 \* (7.0 - (6.0 \* (12.0 / 5)))
   10. 1.3
   11. ""
   12. "CSCI is my favorite department"
   13. [ "Uh-huh", "No really, it is" ]
   14. .3
   15. 4.
   16. 1 / 3
   17. [ 0 ]
   18. [ "0", "0.00" ]
   19. [ 1 / 3, 4, 0 ]
   20. [1: 1 / 3, 4: 4 / 3]
   21. ["A": 1, "B": 2, "D": 4]
   22. [0, 1.0, 2, 3, 4, 5, 6]
   23. ["A", 1, "B", 2, "D", 4]
3. (5 points) Which of the following fail? For those, fix so they are valid. What’s the rule that decides which ones are valid as-is?
   1. print(1 / 3.0)
   2. print(1.0 / 3)
   3. var c = 1; print(c / 3.0)
   4. var d = 1.0; print(d / 3)
   5. var d = 1.0; print(d / 3.0)
4. (2 points) (Reading: [The basics](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html) again, section **Printing Constants and Variables)**. The following works in languages like JavaScript and Python. What’s the Swift way?  
   "The answer is " + 4 + "."
5. (Reading: [Collection Types: Arrays](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html) and [Control Flow](https://docs.swift.org/swift-book/LanguageGuide/ControlFlow.html)) (36 points total)

Assume an array of integers:

var arrayOfInt: [Int] = [ 3, 5, -10, 102, 50, 4, 12, 49, 51, 300, 2 ]

* 1. (1 point) What part is redundant because of type inference?
  2. (27 points) Write several variations on a loop that walks through the array of integers, printing out only those above (not including) 50:
     1. Use an explicit index variable and a while construct.
     2. Use an infinite loop also using while and use break to exit the loop.
     3. Use an explicit index variable and the repeat…while construct.
     4. Use a tail-recursive function that takes the array as the first parameter and the index as the second. Invoke it by calling it once: tailRecurseLoop(array: arrayOfInt, atIndex: 0)
     5. Use a for loop and an explicit range that ranges over all the indices. Use the count property to compute the end index. Did you use ... or ..< and why?
     6. As above, except use the indices property of the Array type instead of an explicit range.
     7. Use a for loop using the in syntax but without mentioning the indices of the Array.
     8. Declare a helper function that takes a single integer parameter and performs the test. Use the forEach method on arrays and pass in this helper function by name.
     9. As above, except declare the helper function inline, anonymously, with no syntactical shortcuts.
  3. (7 points) Generate the following test cases to rigorously test your code. As specified in the style specs, define each one symbolically. (Note: Commerical test systems may generate large test inputs algorithmically, but don’t do that here.) Use this to help you think about how much testing is needed for a reliable full-scale App. Run them all on the tail-recursive version.
     1. The empty boundary condition: An empty array
     2. The smallest input boundary condition: An array with only one value, that produces no output
     3. The smallest input/output boundary condition: An array with only one value, that produces output
     4. The empty output boundary condition: An array with several values that produces no output
     5. The smallest output boundary condition: An array with several values that produces only one output
     6. The general case: An array with several values that produces several outputs
     7. The scaling case: An array with dozens of values.

1. (Reading: Same as previous) (25 points total) Practice with arrays.
   1. (6 points) Use the map method of the Array class to write a function that transforms an array such as:  
      [4, 5, -2]  
      into:  
      [ "Element 1: 4", "Element 2: 5", "Element 3: -2" ]  
      Use String interpolation to make the job easy. Use an anonymous function and trailing closure syntax.
   2. (8 points) Sort the previous array in descending order by passing in a brief anonymous custom comparator function. See **The Sorted Method** in the [closures section of the Swift book](https://docs.swift.org/swift-book/LanguageGuide/Closures.html).
   3. Assuming the above array of strings is called myStringArray, suppose we do:  
      var anotherArray = myStringArray; anotherArray.removeLast() will myStringArray be changed? Why or why not? Compare to another language you know.
   4. (1 point) Look up Array in the [official documentation](https://developer.apple.com/documentation/swift). How many methods and properties does it have?
   5. (10 points) Use the reduce method of the Array class to write a function that computes the sum of any array of integers. Test it on an empty array, an array with one element, and the above 3-element array. Again use an anonymous function. This time, use no explicit types unless required. Let inference take care of the rest.
2. (Reading: [NumberFormatter reference](https://developer.apple.com/documentation/foundation/numberformatter)) (15 points) Practice with number formatting.
   1. Write an extension on the Int struct named centsToUSDollars that transforms the value 492 into $4.92. Note the special cases (they can be logically consolidated; these are just illustrative): 0 → $0.00; 1 → $0.01; 10 → $0.10; 500; → $5.00; 510 → $5.10
   2. Test it on the above 6 cases and 3 more of your own.
   3. Obtain formatted output using the Swift standard library instgead. Use the built-in NumberFormatter class. (Reminder: Standard library functions will not be recognized until you import Foundation.) Use a format type of .currency, the correct locale property value using a Locale object, and the string(from: Number) method. Output assuming currency of Japanese Yen (you’ll want to lookup [ISO currency codes](https://en.wikipedia.org/wiki/ISO_4217)).
3. (Reading: ) (10 points) Write another extension on Int. It should be a computed property of type bool called isPrime. The following pseudo-code is adequate:
4. function is-prime(v):
5. for all integers in from 2 up through the (integer-truncated) square-root of v:
6. if in evenly divides v (no remainder):
7. v has a factor pair, so it is not prime. Return false immediately.
8. if we reach the end of the loop, no factors were found. Return true.

1. (5 points) (Reading: [Enumerations](https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html) but skip **Recursive Enumerations**) Write an enumerated type TippingLevel with the following cases: OK, Good, and Excellent. Include 3 associated Double values: 0.15, 0.18, and 0.20. Use if let and the rawValue to construct an instance of the Excellent case out of a Double literal.
2. (10 points) (Reading: The struct portions of [Structures and Classes](https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html), and [Properties](https://docs.swift.org/swift-book/LanguageGuide/Properties.html), but you can skip the **Type Properties** and **Lazy Stored Properties** sections.) Write a struct named Square that includes a stored property sideLength and a computed property area that works correctly. The sideLength property should have a didSet property observer that rejects negative values by printing an error message and reverting the value. An init method is not necessary. Show it working on a square of size 0, 1, and 2. Change the square of size 2 to 3 and show the area being re-computed properly. Try to change it to -3 and show the error and that the area has not changed.
3. (12 points) (Reading: [Collections — Dictionary-related parts](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html)) Practice with dictionaries.
   1. Declare a Dictionary literal that maps the keys "First", "Second", and "Third" to the 3 square objects declared above.
   2. Use a for…in loop on it to sum the areas, then print the sum with user-friendly labeled output.
   3. Why does the following code fail to mutate the dictionary (that is, fail to actually change the entry stored at key "First")?  var firstSquare = squareDict["First"]  
      firstSquare!.sideLength += 1
   4. Fix the attempt to mutate the dictionary (it will change a lot). To do so, use if let for the dictionary lookup, and, considering that you get an immutable struct as a result, you will have to construct a new Square instance with a length one greater than side length of the looked-up Square. Use the built in member-wise initializer. See the **Memberwise Initializers for Structure Types** paragraph in [the Initialization section Swift book](https://docs.swift.org/swift-book/LanguageGuide/Initialization.html). As a result, the force-unwrap will also be gone.
4. (8 points) Write a new version of the Square struct, SmartSquare. Make it adopt the CustomStringConvertible protocol so it can be used in String interpolation. THe description computed property should describe its size and area.
5. (12 points total) Practice with optionals and function arguments. (Reading: [Functions](https://docs.swift.org/swift-book/LanguageGuide/Functions.html) but skip **In-Out Parameters** and **Variadic Parameters**; **Function Types** will be needed for later assignments only. )
   1. (5 points) Declare a function maybeAdd that takes two values. Each one should be an Optional Int. The return type should be the same. Use a single if let statement to unwrap safely both values and return their sum, or nil otherwise. Use external labels firstAddend and secondAddend, but use internal labels first and second. Give the second argument a default value of 0.
   2. (1 point) Write a version that uses the style: if foo != nil { …foo!… }. Now, delete this version, don't turn it in, and never use this style again.
   3. (3 points) Declare and initialize an empty array of tuples. Each tuple should contain a pair of Optional ints. Since there is no data from which to infer type, this should be explicit.
   4. (3 points) Using the append method 4 times, fill the array with tuples to create test cases for all 4 combinations of nil and non-nil. Use a for...in loop to test your maybeAdd function against these inputs.
      1. For the computable values, the output should be user-friendly as always, such as The sum of *x* and *y* is....
      2. For the un-computable values, a simple message like Could not add: one or more values was missing suffices.
6. (12 points total)

Declare a function strictAdd similar to maybeAdd except that instead of returning nil when the sum is not computable, have it throw an error object. Then demonstrate catching the error.

Example output in a non-exception case:  
The sum of 7 and 3 is 10.

Example output in an exception case:  
Error caught: firstArgumentInvalid

* 1. Use an enum as in the [Swift book example](https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html).
  2. Use separate guard let statements at the top of the function instead of if let.
  3. Use two cases (firstArgumentInvalid, secondArgumentInvalid) in the enum with associated human-friendly String values.
  4. Catch the error within the for loop and print it out with the human-friendly values.