**Fall 2018**

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

**Solutions 1**

**Issued: 09/16/2018**

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

1. (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 Int (although this is arguably ambiguous; in the compound expression 0 / 4.0 the 0 will be treated as a Double)
   2. "0" String
   3. '0' (Hint: Individual symbols or characters are [nothing like as in C](https://developer.apple.com/documentation/swift/character) and related languages. Illegal expression; Unicode characters are much more complicated than ASCII so there are no literals for the general case. Use Character("0") instead.
   4. 0.0 Double
   5. "0.0" String (Always; there will never be automatic coercion like JavaScript or PHP!)
   6. "0.0".count Int: 3 (Might look odd, but this is just a computed property on a struct)
   7. 3 + 4 \* (7 - (6 \* (12 / 5))) Int: -17 (All integer arithmetic, truncating division.)
   8. 3 + 4 \* (7 - (6 \* (12 / 5.0))) Double: -26.6 (Approximately; rendering Double values in decimal is almost never exact)
   9. 3.0 + 4.0 \* (7.0 - (6.0 \* (12.0 / 5))) Double: -26.6 (No different; the promotion-to-Double logic makes this exactly equivalent)
   10. 1.3 Double
   11. "" String (Empty literals present no problem)
   12. "CSCI is my favorite department" String
   13. [ "Uh-huh", "No really, it is" ] Array<String> (or [String])
   14. .3 Illegal (Swift is picky, supposed to reduce errors). Corrected: Double: 0.3
   15. 4. Also illegal (Compare: most other languages). Corrected: Double: 4.0
   16. 1 / 3 Int: 0
   17. [ 0 ] Array<Int> or [Int]
   18. [ "0", "0.00" ] Array<String> or [String]
   19. [ 1 / 3, 4, 0 ] [Int]: [0, 4, 0]
   20. [1: 1 / 3, 4: 4 / 3] Dictionary<Int, Int> or [Int: Int]: [1: 0, 4: 1]
   21. ["A": 1, "B": 2, "D": 4] Dictionary<String, Int>
   22. [0, 1.0, 2, 3, 4, 5, 6] Array<Double> Note we have to find compatible type for ALL constants
   23. ["A", 1, "B", 2, "D", 4] Weird one. There is no common supertype so we must rever to the most generic catch-all type: Any. Technically, [Any] but sufficiently likely to be an error that you must make explicit annotation: ["A", 1, "B", 2, "D", 4] as Any
2. (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) No problem, the 1 is auto-promoted
   2. print(1.0 / 3) Also no problem
   3. var c = 1; print(c / 3.0) Now c cannot be promoted; it is glued down as an Int. Fix: print(Double(c) / 3.0)
   4. var d = 1.0; print(d / 3) Not a problem, as d requires no promotion.
   5. var d = 1.0; print(d / 3.0) Not a problem, obviously. The rule is that a *variable* once typed cannot change type in an expression, where as numerical constants are fluid.
3. (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 + "." The + operator is defined but only on (String, String).  
   Awkward way: "The answer is " + 4.description + "."  
   Correct way: "The answer is \(4)."

(Use your imagination; Rather than a constant, the numeric expression inside () is usually non-trivial using variables.)

Too-clever way: "The answer is 4." (OK, we have to give this credit.)

1. (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? [Int] Observe that they are all Int constants
  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. var i = 0
     3. while i < arrayOfInt.count {
     4. if arrayOfInt[i} > 50 {
     5. print(arrayOfInt[i])
     6. }
     7. i += 1

}

* + 1. Use an infinite loop also using while and use break to exit the loop. Probably the most awkward version.
    2. var i = 0
    3. while true {
    4. if arrayOfInt[i] > 50 {
    5. print(arrayOfInt[i])
    6. }
    7. i += 1
    8. if i == arrayOfInt.count { // or >=
    9. break
    10. }

}

* + 1. Use an explicit index variable and the repeat…while construct. Also quite awkward: forced repetition of test.
    2. var i = 0
    3. repeat {
    4. if i < arrayOfInt.count {
    5. if arrayOfInt[i] > 50 {
    6. print(arrayOfInt[i])
    7. }
    8. }

} while i < arrayOfInt.count

* + 1. 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)
    2. func tailRecurseLoop(array: [Int], atIndex index: Int) {
    3. // Base case
    4. if index >= array.count { // Should not be ==; we don't know how we will be called.
    5. return
    6. }
    7. // Recursive case: Do a small amount of work...
    8. if array[index] > 50 {
    9. print(array[index])
    10. }
    11. // Then recurse after breaking off that work
    12. tailRecurseLoop(array, index + 1) // No need for mutation!

}

* + 1. 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? Must be mindful of 0-indexing, so end of range is one LESS than count: ..<
    2. for index in 0..<arrayOfInt.count {
    3. if arrayOfInt[index] > 50 {
    4. print(array[index])
    5. }

}

* + 1. As above, except use the indices property of the Array type instead of an explicit range.
    2. for index in arrayOfInt.indices {
    3. if arrayOfInt[index] > 50 {
    4. print(array[index])
    5. }

}

* + 1. Use a for loop using the in syntax but without mentioning the indices of the Array. Now this is starting to look idiomatic and compact.
    2. for element in arrayOfInt {
    3. if element > 50 {
    4. print(element)
    5. }

}

* + 1. 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. Now we are back into wordy territory, but with a powerful abstraction and no index clutter.
    2. func printIfAbove50(val: Int) {
    3. if val > 50 {
    4. print(val)
    5. }
    6. }
    7. // Now we are in self-explaining one-liner territory!
    8. arrayOfInt.forEach(printIfAbove50)
    9. // But yuck, magic constant! We have to learn true closures now.
    10. func makeComparator(withThreshold threshold: Int) {
    11. return { (val: Int) -> Bool in
    12. print("\(val) is above \(threshold)") // And be more helpful in output right?
    13. }
    14. }
    15. // Learn and love the weirdness; capturing closures and UI programming go hand-in-hand.

arrayOfInt.forEach(makeComparator(50))

* + 1. As above, except declare the helper function inline, anonymously, with no syntactical shortcuts. No shortcuts, so include even Void.
    2. arrayOfInt.forEach { (val: Int) -> Void in
    3. if val > 50 {
    4. print(val)
    5. }

}

* 1. (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 let t1ebc: [Int] = [] or let t1ebc = Array<Int>()
     2. The smallest input boundary condition: An array with only one value, that produces no output let t2sibc = [50]
     3. The smallest input/output boundary condition: An array with only one value, that produces output let t3siobc = [51]
     4. The empty output boundary condition: An array with several values that produces no output let t4eobc = [-200, 2, 48, 49]
     5. The smallest output boundary condition: An array with several values that produces only one output let t5eobc = [-200, 2, 53, 49]
     6. The general case: An array with several values that produces several outputs let t6eobc = [-200, 2, 53, 49, 99, 5000, 23, 219]
     7. The scaling case: An array with dozens of values. Lots of possibilities. Repetition is OK but let’s make sure the code cannot fall back on patterns, and further that about half fall above and half fall below by selecting uniformly from -2450 through 2550. Note the use of \_ for an unused variable.
     8. let scalingSize = 300
     9. var t7sc = [Int]()
     10. for \_ in 0..<scalingSize {
     11. t7sc.append(arc4random\_uniform(5000) - 2450)

}

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. [4, 5, -2].enumerated().map { (index: Int, elem: Int) -> String in
   3. return "Element \(index): \(elem)"

}

* 1. (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).
  2. [4, 5, -2].sorted { (lhs: Int, rhs: Int) -> Bool in
  3. return lhs > rhs

}

Or with all the shortcuts:

[4, 5, -2].sorted { $0 > $1 }

* 1. 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. No, because Arrays are value types, so the = makes a complete, independent copy. This is not true in Java, C (depending on declaration), and JavaScript, among many others.
  2. (1 point) Look up Array in the [official documentation](https://developer.apple.com/documentation/swift). How many methods and properties does it have? There are 226 counting overloads separately. Cheat code for JavaScript debugger: document.getElementsByClassName("identifier").length. Did you know about += for example? On *Arrays*? Before writing custom boilerplate code, check the [Standard Library!](https://developer.apple.com/documentation/swift/swift_standard_library)
  3. (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.
  4. func sumOf(\_ a: [Int]) -> Int {
  5. return a.reduce(0) { (partialSum: Int, nextValue: Int) in
  6. partialSum + nextValue
  7. }
  8. }
  9. let emptyArr = [Int]()
  10. let oneElemArr = [ 5 ]
  11. let threeElemArr = [ -8, 12, 9 ]

print("Empty: \(sumOf(emptyArr)) One elem: \(sumOf(oneElemArr)) 3 elem: \(sumOf(threeElemArr))")

1. (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 → $5.10 This is a good answer for working on first principles and learning simple steps:
   2. extension Int {
   3. var centsToUSDollars: String {
   4. let cents = self % 100
   5. let dollars = self / 100
   6. let centsZeroFill = cents < 10 ? "0" : ""
   7. return "$\(dollars).\(centsZeroFill)\(cents)"
   8. }

}

This is a good answer for maximizing the Standard library and compactness:

extension Int {

var centsToUSDollars: String {

return String(format: "$%d.%02d", self / 100, self % 100)

}

}

* 1. Test it on the above 6 cases and 3 more of your own. Good would be some more boundary cases: $0.09, $0.11, $0.99, $1.01, and maybe a very large value. Did anyone try negative values? (Where should the minus sign go?)
  2. Obtain formatted output using the Swift standard library instead. 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)). Hundredths of Yen do not exist, so perhaps a different currency would be more practical. Apparently, the value "JPY", which comes directly from the static array Locale.isoCurrencyCodes, does not work and "ja-JP"must be used. Did anyone use a helper extension?
  3. import Foundation
  4. extension NumberFormatter {
  5. static var ToJapaneseYen: NumberFormatter {
  6. let fmt = NumberFormatter()
  7. fmt.numberStyle = .currency
  8. fmt.locale = Locale(identifier: "ja\_JP")
  9. return fmt
  10. }
  11. }
  12. extension Int {
  13. var hundredthsOfYenToYen: String {
  14. return NumberFormatter.ToJapaneseYen.string(from: self as NSNumber) ?? "Yen format error for \(self)!"
  15. }

}

1. (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:
2. function is-prime(v):
3. for all integers in from 2 up through the (integer-truncated) square-root of v:
4. if in evenly divides v (no remainder):
5. v has a factor pair, so it is not prime. Return false immediately.

if we reach the end of the loop, no factors were found. Return true.

We will take the liberty to write it as a computed property since it takes no arguments other than self. Did you experience the Range crashes with 2 before debugging?

extension Int {

var isPrime: Bool {

switch self {

case Int.min...1: return false

case 2: return true

default:

/\* For the mathematically inclined: A lot of this is wasted. It makes no sense to test division

by 4, because we have already tested divison by 2 and failed if we got that far. Every number

has a unique prime factorization if non-prime. We should only be testing against prime

divisors, which can be discovered recursively. \*/

for factor in 2...Int(sqrt(Double(self))) {

print("trying: \(factor)")

if self % factor == 0 {

return false

}

} /\* for \*/

return true

} /\* switch \*/

} /\* isPrime \*/

} /\* extension \*/

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.Reverting to the new convention that cases should be lowercase.
2. enum TippingLevel: Double {
3. case ok = 0.15
4. case good = 0.18
5. case excellent = .20
6. }
7. if let okVal = TippingLevel(rawValue: 0.15) {
8. print("constructed okVal: \(okVal)")
9. }
10. else {
11. print("Could not construct from 0.15!")

}

1. (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.
2. struct Square: CustomStringConvertible {
3. var sideLength: Double /\* or Int \*/ {
4. didSet {
5. if sideLength < 0 {
6. sideLength = oldValue
7. }
8. }
9. }
10. var area: Double {
11. return sideLength \* sideLength
12. }
13. var description: String {
14. return "side: \(sideLength) area: \(area)"
15. }
16. }
17. var firstSq = Square(sideLength: 0)
18. var secondSq = Square(sideLength: 1)
19. var thirdSq = Square(sideLength: 2)
20. print("first sq: \(firstSq) second sq: \(secondSq) third sq: \(thirdSq)")
21. thirdSq.sideLength = 3
22. print("mutated third sq: \(thirdSq)")
23. thirdSq.sideLength = -3

print("rejected mutation to -3 on third sq: \(thirdSq)")

1. (12 points) (Reading: [Collections — Dictionary-related parts](https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html)) Practice with dictionaries.
   1. Declare a Dictionary variable initialized to map the keys "First", "Second", and "Third" to the 3 square objects declared above.

var squareDict = [ "First": firstSq, "Second": secondSq, "Third": thirdSq ]

* 1. Use a for…in loop on it to sum the areas, then print the sum with user-friendly labeled output.
  2. var areaSum = 0.0 // Why not 0?
  3. for (key, val) in squareDict {
  4. areaSum += val.area
  5. }

print("The sum of the areas of the squares is \(areaSum)")

* 1. 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 firstSquare is a copy of the Dictionary entry, since the looked up entry (a struct) is a value type.
  2. (This has been re-written for clarity from the original assignment to better capture the intent.) Fix the attempt to mutate the dictionary. (This will be a complete re-write.) Rules:
     1. You must be able to detect lookup failure and print an error.
     2. You cannot use !.
     3. You may not write another initializer so 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. You may not compare to nil.

squareDict["First"]?.sideLength += 1 works except that failure cannot be detected to take an alternate action.

let key = "First" // Avoid repeating constant which could be mistyped

if let firstSquare = squareDict[key] {

squareDict[key] = Square(sideLength: firstSquare.sideLength + 1)

}

else {

print("Could not look up \(key) in square dictionary")

}

Alternate 1:

let key = "First" // Avoid repeating constant which could be mistyped

if let square = squareDict[key] {

var mutatedSquare = square

mutatedSquare.sideLength += 1

squareDict[key] = mutatedSquare

}

else {

print("Could not look up \(key) in square dictionary")

}

Alternate 2:

let key = "First" // Avoid repeating constant which could be mistyped

if let \_ = squareDict[key] {

squareDict[key]?.sideLength += 1

}

else {

print("Could not look up \(key) in square dictionary")

}

1. (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. Already done for convenience in previous version.
2. (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 letstatement to unwrap safely both values and return their sum, or nil otherwise. Use external labels firstAddend and secondAddend, but use internal labels firstand second. Give the second argument a default value of 0.
   2. func maybeAdd(firstAddend first: Int?, secondAddend second: Int? = 0) -> Int? {
   3. if let lhs = first, let rhs = second {
   4. return lhs + rhs
   5. }
   6. else {
   7. return nil
   8. }

}

* 1. (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.
  2. ~~func maybeAddBadStyle(firstAddend first: Int?, secondAddend second: Int? = 0) -> Int? {~~
  3. ~~if first != nil && second != nil {~~
  4. ~~return first! + second!~~
  5. ~~}~~
  6. ~~else {~~
  7. ~~return nil~~
  8. ~~}~~

~~}~~

* 1. (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. var tupleArr: [(first: Int?, second: Int?)] = []
  2. (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.
  3. tupleArr.append((nil, nil)) // did you find the double-paren tricky? Different meanings.
  4. tupleArr.append((nil, 1))
  5. tupleArr.append((2, nil))
  6. tupleArr.append((3, 4))
  7. for tuple in tupleArr {
  8. if let first = tuple.first,
  9. let second = tuple.second,
  10. let sum = maybeAdd(firstAddend: first, secondAddend: second) {
  11. print("Sum of \(first) and \(second) is \(sum)")
  12. }
  13. else {
  14. print("Could not add: one or more values was missing")
  15. }

}

1. (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.

enum AddException: Error {

case firstArgumentInvalid(String)

case secondArgumentInvalid(String)

}

/\* Did you remember to change the return type? \*/

func strictAdd(firstAddend first: Int?, secondAddend second: Int? = 0) throws -> Int {

guard let lhs = first else {

throw AddException.firstArgumentInvalid("nil value")

}

guard let rhs = second else {

throw AddException.secondArgumentInvalid("nil value")

}

return lhs + rhs

}

func doSomeExceptionalMath(withTuples tupleArr: [(first: Int?, second: Int?)] ) throws {

for tuple in tupleArr {

do {

let sum = try strictAdd(firstAddend: tuple.first, secondAddend: tuple.second)

// Somewhat annoyingly, we still have to use unwrapping. Shows that mixing exceptions with Optionals

// is a bit awkward.

let firstPrinted = tuple.first?.description ?? "missing"

let secondPrinted = tuple.second?.description ?? "missing"

print("Sum of \(firstPrinted) and \(secondPrinted) is \(sum)")

}

catch let exc {

guard let addExc = exc as? AddException else {

throw exc

}

print("Error: \(addExc)")

}

}

}

doSomeExceptionalMath(withTuples: tupleArr)