# TY CSE AY-2023-24 Sem-II

# iOS Lab (6CS381)

## Assignment No 11

**Due date- 10/04/2024**

**PART A**

1. Create iOS application and test on MacBook, iPad and iPhone whose design made in assignment No. 1

**PART B**

**## Guard Statements**

1. Imagine you want to write a function to calculate the area of a rectangle. However, if you pass a negative number into the function, you don't want it to calculate a negative area. Create a function called `calculateArea` that takes two `Double` parameters, `x` and `y`, and returns an optional `Double`. Write a guard statement at the beginning of the function that verifies each of the parameters is greater than zero and returns `nil` if not. When the guard has succeeded, calculate the area by multiplying `x` and `y` together, then return the area. Call the function once with positive numbers and once with negative number.

2. Create a function called `add` that takes two optional integers as parameters and returns an optional integer. You should use one `guard` statement to unwrap both optional parameters, returning `nil` in the `guard` body if one or both of the parameters doesn't have a value. If both parameters can successfully be unwrapped, return their sum. Call the function once with non-`nil` numbers and once with at least one parameter being `nil`. When working with UIKit objects, you will occasionally need to unwrap optionals to handle user input. For example, the text fields initialized below have `text` properties that are of type `String?`.

3. Write a function below the given code called `createUser` that takes no parameters and returns an optional `User` object. Write a guard statement at the beginning of the function that unwraps the values of each text field's `text` property, and returns `nil` if not all values are successfully unwrapped. After the guard statement, use the unwrapped values to create and return and instance of `User`.

```
*/
struct User {
var firstName: String
var lastName: String
var age: String
}
let firstNameTextField = UITextField()

let lastNameTextField = UITextField()

let ageTextField = UITextField()
```

```
firstNameTextField.text = "Jonathan"

lastNameTextField.text = "Sanders"

ageTextField.text = "28"
```

4. Call the function you made above and capture the return value. Unwrap the `User` with standard optional binding and print a statement using each of its properties.

5. In the exercises on optionals, you created a failable initializer for a `Workout` struct that would only initialize a `Workout` object if the `startTime` and `endTime` were further apart than 10 seconds. You'll now create the same failable initializer, only using a guard statement to check that the start and end times aren't too close together.

6. Create a `Workout` struct that has properties `startTime` and `endTime` of type `Double`. Dates are difficult to work with, so you'll be using doubles to represent the number of seconds since midnight, i.e. 28800 would represent 28,800 seconds, which is exactly 8 hours, so the time would be 8am.

7. Write a failable initializer that takes parameters for your start and end times, and then checks to see if they are greater than 10 seconds apart using a guard statement. If they are, your initializer should fail. Otherwise, the initializer should set the properties accordingly.

8. Imagine a screen where a user inputs a meal that they've eaten. If the user taps a "save" button without adding any food, you might want to prompt the user that they haven't actually added anything. Using the `Food` struct and the text fields provided below, create a function called `logFood` that takes no parameters and returns an optional `Food` object. Inside the body of the function, use a guard statement to unwrap the `text` property of `foodTextField` and `caloriesTextField`. In addition to unwrapping `caloriesTextField`, you'll need to create and unwrap a new variable that initializes an `Int` from the text in `caloriesTextField`. If any of this fails, return `nil`. After the guard statement, create and return a `Food` object.

```
*/ struct Food {
var name: String
var calories: Int
}
let foodTextField = UITextField()
let caloriesTextField = UITextField()
foodTextField.text = "Banana"
caloriesTextField.text = "23"
```

9. Call the function you made above and capture the return value. Unwrap the `Food` object with standard optional binding and print a statement about the food using each of its properties. Go back and change the text in `caloriesTextField` to a string that cannot be converted into a number. What happens in that case?

## Scope

10. Using a comment or print statement, describe why the code below will generate a compiler error
if you uncomment //print statement.

```
*/
for _ in 0..<10 {
let foo = 55
print("The value of foo is \(foo)")
}
//print("The value of foo is \(foo)")
```

11. Using a comment or print statement, describe why both print statements below compile when similar-looking code did not compile above. In what scope is `x` defined, and in what scope is it modified? In contrast, in what scope is `foo` defined and used?

```
var x = 10
for _ in 0..<10 {
x += 1
print("The value of x is \(x)")
}
print("The final value of x is \(x)")
```

12. In the body of the function `greeting` below, use variable shadowing when unwrapping `greeting`. If `greeting` is successfully unwrapped, print a statement that uses the given greeting to greet the given name (i.e. if `greeting` successfully unwraps to have the value "Hi there" and `name` is `Sara`, print "Hi there, Sara."). Otherwise, use "Hello" to print a statement greeting the given name. Call the function twice, once passing in a value for greeting, and once passing in `nil`.

```
func greeting(greeting: String?, name: String) {
}
```

13. Create a class called `Car`. It should have properties for `make`, `model`, and `year` that are of type `String`, `String`, and `Int`, respectively. Since this is a class, you'll need to write your own memberwise initializer. Use shadowing when naming parameters in your initializer.

14. Below is a `User` struct and three `User` instances. These will be used throughout the exercises below to simulate competition in the fitness tracking app.

```
struct User {
var name: String
var stepsToday: Int
}
let stepMaster = User(name: "StepMaster", stepsToday: 8394)
let activeSitter = User(name: "ActiveSitter", stepsToday: 9132)
let monsterWalker = User(name: "MonsterWalker", stepsToday: 7193)
let competitors = [stepMaster, activeSitter, monsterWalker]
```

The function below takes an array of `User` objects and returns the `User` object that has taken the most steps. The body of the function first declares a variable that is an optional

`User`, then loops through all of the users in the array. Inside each iteration of the loop, it will check if `topCompetitor` has a value or not by unwrapping it. If `topCompetitor` doesn't have a value, then the current user in the iteration is assumed to have the highest score and is assigned to `topCompetitor`. If `topCompetitor` has a value, there is code to check whether the current user in the iteration has taken more steps than the user that is assigned to `topCompetitor`.

15. At that point, the goal is to assign the user with the higher score to `topCompetitor`. However, the code generates a compiler error because, due to improper variable shadowing, `topCompetitor` has a narrower scope than it should if it is going to be reassigned. Fix the compiler error below and call `getWinner(competitors:)`, passing in the array `competitors`. Print the `name` property of the returned `User` object. You'll know that you fixed the function properly if the user returned is `activeSitter`.

```
func getWinner(competitors: [User]) -> User? {
   var topCompetitor: User?

   for competitor in competitors {
      if let topCompetitor = topCompetitor {
         if competitor.stepsToday > topCompetitor.stepsToday {
            topCompetitor = competitor
         }
      } else {
         topCompetitor = competitor
      }
   }
   return topCompetitor
}
```

16. Write a memberwise initializer inside the `User` struct above that uses variable shadowing for
naming the parameters of the initializer.

17. Now write a failable initializer inside the `User` struct above that takes parameters `name` and `stepsToday` as an optional `String` and `Int`, respectively. The initializer should return `nil` if either of the parameters are `nil`. Use variable shadowing when unwrapping the two parameters.

## Enumerations

18. Define a `Suit` enum with four possible cases: `clubs`, `spades`, `diamonds`, and `hearts`.
*/

19. Imagine you are being shown a card trick and have to draw a card and remember the suit. Create a variable instance of `Suit` called `cardInHand` and assign it to the `hearts` case. Print out the instance.

20. Now imagine you have to put back the card you drew and draw a different card. Update the variable to be a spade instead of a heart.

21. Imagine you are writing an app that will display a fun fortune (i.e. something like "You will soon find what you seek.") based on cards drawn. Write a function called `getFortune(cardSuit:)` that takes a parameter of type `Suit`. Inside the body of the function, write a switch statement based on the value of `cardSuit`. Print a different fortune for each `Suit` value. Call the function a few times, passing in different values for `cardSuit` each time.

22. Create a `Card` struct below. It should have two properties, one for `suit` of type `Suit` and another for `value` of type `Int`.

23. How many values can playing cards have? How many values can `Int` be? It would be safer to have an enum for the card's value as well. Inside the struct above, create an enum for `Value`. It should have cases for `ace`, `two`, `three`, `four`, `five`, `six`, `seven`, `eight`, `nine`, `ten`, `jack`, `queen`, `king`. Change the type of `value` from `Int` to `Value`. Initialize two `Card` objects and print a statement for each that details the card's value and suit.

**## App Exercise - Swimming Workouts**

24. Previous app exercises have introduced the idea that your fitness tracking app may allow users to track swimming workouts. Create a `SwimmingWorkout` struct below with properties for `distance`, `time`, and `stroke`. `distance` and `time` should be of type `Double` and will represent distance in meters and time in seconds, and `stroke` should be of type `String`.

25. Allowing `stroke` to be of type `String` isn't very type-safe. Inside the `SwimmingWorkout` struct, create an enum called `Stroke` that has cases for `freestyle`, `butterfly`, `backstroke`, and `breaststroke`. Change the type of `stroke` from `String` to `Stroke`. Create two instances of `SwimmingWorkout` objects.

26. Now imagine you want to log swimming workouts separately based on the swimming stroke. You might use arrays as static variables on `SwimmingWorkout` for this. Add four static variables, `freestyleWorkouts`,`butterflyWorkouts`,`backstrokeWorkouts`,and `breaststrokeWorkouts`, to `SwimmingWorkout` above. Each should be of type `[SwimmingWorkout]` and should default to empty arrays.

27. Now add an instance method to `SwimmingWorkout` called `save` that takes no parameters and has no return value. This method will add its instance to the static array on `SwimmingWorkout` that corresponds to its swimming stroke. Inside `save` write a switch statement that switches on the instance's `stroke` property, and appends `self` to the proper array. Call save on the two instances of `SwimmingWorkout` that you created above, and then print the array(s) to which they should have been added to see if your `save()` method works properly.

******************