**Question 1.**

Write a function called siftBeans(fromGroceryList:) that takes a grocery list (as an array of strings) and “sifts out” the beans from the other groceries. The function should take one argument that has a parameter name called list, and it should return a named tuple of the type (beans: [String], otherGroceries: [String]).

**Hint:**

Here is an example of how you should be able to call your function and what the result should be:

let result = siftBeans(fromGroceryList: ["green beans",

"milk",

"black beans",

"pinto beans",

"apples"])

result.beans == ["green beans", "black beans", "pinto beans"] // true

result.otherGroceries == ["milk", "apples"] // true”

**func** siftBeans(fromGroceryList : [String]) -> (beans : [String], otherGroceries : [String]){

**var** beans = [String]()

**var** otherGroceries = [String]()

**for** elem **in** fromGroceryList{

**if** elem.hasSuffix("beans"){

beans.append(elem)

}

**else**{

otherGroceries.append(elem)

}

}

**return** (beans, otherGroceries)

}

**let** result = siftBeans(fromGroceryList: ["green beans",

"milk",

"black beans",

"pinto beans",

"apples"])

result.beans

result.otherGroceries

**Question 2 -**

Make a calculator class with a function name “equals” that take an enum case as value like multiply, subtraction, addition, square root, division.

**enum** Operations{

**case** add(Double,Double)

**case** subtract(Double,Double)

**case** multiply(Double,Double)

**case** divide(Double,Double)

**case** squareroot(Double)

}

**class** Calculator{

**func** equals(**\_** operation: Operations ) -> Double{

**switch** operation{

**case** .add(**let** num1, **let** num2):

**return** num1 + num2

**case** .subtract(**let** num1, **let** num2):

**return** num1 - num2

**case** .multiply(**let** num1, **let** num2):

**return** num1 \* num2

**case** .divide(**let** num1, **let** num2):

**return** num1/num2

**case** .squareroot(**let** num1):

**return** num1.squareRoot()

}

}

}

**var** c = Calculator()

c.equals(Operations.add(2, 3))

c.equals(Operations.subtract(7, 3))

c.equals(Operations.multiply(45, 3))

c.equals(Operations.divide(75, 3))

c.equals(Operations.squareroot(81))

**Question 3 -**

Make the same calculator using functions as an argument, define all type functions in a struct.

**var** addClosure : ((Double,Double?) -> Double) = {(**\_** num1 : Double, **\_** num2 : Double?) -> Double **in**

**return** num1 + num2!

}

**var** subtractClosure : ((Double,Double?) -> Double) = {(**\_** num1 : Double, **\_** num2 : Double?) -> Double **in**

**return** num1 - num2!

}

**var** multiplyClosure : ((Double,Double?) -> Double) = {(**\_** num1 : Double, **\_** num2 : Double?) -> Double **in**

**return** num1 \* num2!

}

**var** divideClosure : ((Double,Double?) -> Double) = {(**\_** num1 : Double, **\_** num2 : Double?) -> Double **in**

**return** num1 / num2!

}

**var** squareRootClosure : ((Double,Double?) -> Double) = {(**\_** num1 : Double,**\_** num2 : Double?) -> Double **in**

**return** num1.squareRoot()

}

**class** Calc{

**func** equal(**\_** number1 : Double, **\_** number2 : Double?, **\_** oper : (Double,Double?) -> Double) -> Double{

**if** number2 != **nil**{

**return** oper(number1,number2!)

}

**else**{

**return** oper(number1,**nil**)

}

}

}

**var** ca = Calc()

ca.equal(2, 6, addClosure)

ca.equal(14, 7, subtractClosure)

ca.equal(3, 50, multiplyClosure)

ca.equal(12, 3, divideClosure)

ca.equal(81, **nil**, squareRootClosure)

**Question 4 -**

Create a TraineesActivity Class which lazily initializes a data source of all the trainees in an array.

Define a closure to filter and find the trainee object based on the name passed.

Create an enum explained below which would also have a function returning a closure that takes the trainee object and return a string describing the skill for every enum case.

This TraineeActivity would provide three functions as below to perform, record, and rerun the activity. On calling perform passing the name of trainee make use of closure declared to find the trainee object, pass this object to activity closure defined in enum to execute the activity. Later record this activity in any data structure mapped to a trainee and use this data structure to rerun the activity performed.

on deinitialization, it should print - **Hey !!! Thanks, I am gone.**

***Note*** *- Make use of closures, lazy, type alias, optional binding & chaining,*

**Outline of class and data should be as follows -**

Class TraineesActivity

trainesData - load lazily

closure - chooseFilterObject

functions -

performActivity

recordActivity

rerunActivity

Struct Trainee

- name

- dance = 78

- run = 65

- Sing = 35

- fight = 2

- academic = 46

Enum {

case dance

case academic

case run

case sing

case Fights

a function returning activity closure that take trainee object and prints the activity skill

}

**Test Run -**

**var** trainee : Tainees? = Tainees()

trainee?.performActivity(trainee: “Waseem”, activity: .run)

trainee?.performActivity(trainee: “Anindiya”, activity: .academic)

trainee?.performActivity(trainee: “Rekha”, activity: .run)

trainee?.rerunActivity()

trainee = **nil**

*Prints log -*

Waseem **good run 70**

Anindiya **good academic 45**

**No trainee found**

Waseem **good run 70**

Anindiya **good academic 45**

**Hey !!! Thanks, I am gone.**