# Diving in with Swift ( Part 1 ) Solutions

*Remember that everything can be solved in many ways; there is no “one right” way to do it.

Q. 1

A constant is useful for when you know the value is not going to change throughout the declaration of the program. In other words, because you don’t change it, it becomes *immutable.* A variable is the exact opposite; they are declared as mutable pieces of memory that can be freely changed throughout the duration of the program. If you try to mutate a constant, Xcode will issue you an error as part of its safety mechanisms.

Variable declarations and legal uses:

```
var a = 89
var b = “Hello, from Swift!”

a += 1
b += String(a)
```

Constant declarations (now immutable):

```
let a = 89
let b = “Hello, from Swift!”
```

Q. 2

As previously defined, Swift differentiates between mutability through the use of constants or variables. Simply put, constants are immutable (cannot be changed) and variables are mutable (can be changed):

```
var mutableString = “I can change!”
let immutableString = “I can’t change!”
```

Q. 3

The preferred practice is to allow the type-inference, unless the situation calls for it - such as declaring variables without an initial value, or perhaps creating an empty array for later use, etc. The types are implicit based on what you assign initially, so the compiler is aware of what type it should be.

```
var a = 3
var b = 9
var c = “Hello from Swift!”
var d = 45.0
var e = Int(“4”)
let pi = 3.141596
```

Q. 4

TRUE ( There is no “equals” method as there is in java, for instance. This might show how close Swift is to Python ).

Q. 5

We can use any strings we'd like, as long as the question criteria is fit, so here are my picks:

```
let greeting = "Hello"
let subject = "World"
let name = "Johnny Appleseed"

//String interpolation
var sentence = "\(greeting) \(subject)!"

//Shorthand, string interpolation, method use
sentence += " My name is"
sentence.append(" \(name)")

//Basic concatenation with addition
let finalSentence = sentence + " What is yours?"
```

Q. 6

You can choose whatever quote you want

```
let quote = "It does not do to dwell in dreams
and forget to live, Harry."

print("Then Dumbledore said, \"\(quote)\"")
```

Q. 7

No, both operations on the constants are invalid. Since they are constants, doing any type of mutability ( append or shorthand, etc), is an error. You can fix it by removing those mutating statements or changing the constants to variables.

Q. 8

```
let str = String(atHomeEP)
```

Q. 9

You can only index, or get ranges of, strings using String.Index's. So, I get those with a property and a method, then I create my string by converting the characters to strings and concatenating them.

```
let str = "Hallo!, Wie geht's dir?"
let ind1 = str.startIndex
let ind2 = str.index(ind1, offsetBy:8)
let newStr = String(str[ind1]) +
String(str[ind2]) //Prints "Hi"
```

Q. 10

There may be a few ways you can do this, but this is my solution. Gentle reminder to pay attention to what the methods you call return, as they may sometimes not be in-place.

```
var str = "Oh no, I can do this"
str.append(".")
str = str.replacingOccurences(of:",", with:"!") //
returns new string
```

Alternatively, you could have used the “replaceSubrange” method, but it’s slightly more involved.

Q. 11

```
var str = “Oh no, I can do this”
str.append(“.”)
str = str.replacingOccurences(of:”I can”, with:”I’m sure I can”)
```

For the removal, you can be as cheeky as you want or continue to use “replacingOccurences” or any you wish:

```
str = str.replacingOccurences(of:”I’m sure I can”, with:”I can”)

str.removeLast()

// one other option: str.removeAll(){ $0 == Character(“.”) }
```

Q. 12

When you first declare one, they share the same spot in memory as the string to conserve it. When you decide to mutate it, that is when the substring will occupy its own spot in memory. This effectively goes to most of the other Swift constructs. Of course, when you create a substring it’s still an instance of a substring. One minor issue is that, as long as substrings are used, the original string remains in memory.

Q. 13

```
let sent = “Mr. and Mrs. Dursley of Privet
Drive.”

let ind1 = sent.startIndex
let ind2 = sent.index(ind1, offsetBy: 19)

print(sent[ind1...ind2])
```

Q. 14

```
let str = “I would have stayed at home!”
```

a.

```
let ind1 = str.index(str.startIndex, offsetBy:2)
let ind2 = str.index(ind1, offsetBy: 19)

print(str[ind1...ind2])
```

b.

```
let ind1 = str.index(str.endIndex, offsetBy:-5)
let ind2 = str.endIndex

print(str[ind1..<ind2])
```

c.

```
let ind1 = str.startIndex
let ind2 = str.index(ind1, offsetBy:7)
let ind3 = str.index(ind2, offsetBy:6)
let ind4 = str.index(ind3, offsetBy:3)
let ind5 = str.index(str.endIndex, offsetBy:-6)
let ind6 = str.endIndex

print(str[ind1...ind2]+str[ind3...ind4]+str[ind5..<
ind6])
```

d.

```
let ind1 = str.startIndex
let ind2 = str.index(ind1, offsetBy:1)
let ind3 = str.index(ind2, offsetBy:7)
let ind4 = str.index(ind3, offsetBy:4)
let ind5 = str.index(str.endIndex, offsetBy:-8)
let ind6 = str.index(str.endIndex, offsetBy:-6)
let ind7 = str.endIndex

print(str[ind1...ind2]+str[ind3...ind4]+str[ind5...
ind5]+str[ind6..<ind7])
```

e.

```
let ind1 = (str.firstIndex(of:"s"))!
let ind2 = str.index(ind1, offsetBy:5)

print(str[ind1...ind2])
```

f.

```
let ind1 = str.index(str.endIndex, offsetBy: -8)
let ind2 = str.index(ind1, offsetBy:5)

print(str[ind1...ind2])
```

g.

```
let ind1 = str.startIndex
let ind2 = str.index(ind1, offsetBy:7)
let ind3 = (str.firstIndex(of: “s”))!
let ind4 = str.index(ind3, offsetBy:3)
let ind5 = str.index(str.endIndex, offsetBy:-1)

print(str[ind1...ind2]+str[ind3...ind4]+str[ind5...
ind5])
```

h.

```
var str2 =
str.replacingOccurences(of:”would”,with:”should”)

let ind1 = str2.startIndex
let ind2 = str2.index(ind1, offsetBy:13)
let ind3 = str2.index(str.endIndex, offsetBy:-8)
let ind4 = str2.index(str.endIndex, offsetBy:-6)
let ind5 = str2.endIndex

print(str[ind1...ind2]+str[ind3...ind3]+str[ind4..<
ind5])
```

Q. 15

These methods give you either a character array or a substring. Prefix gives you the first X characters, and suffix gives you the last X characters. I chose a very simple example:

```
let str = “Ich habe den Apfel gegessen!”
print(str.suffix(5)) // gives “essen!”
print(str.prefix(8)) // gives “Ich habe”
```

Q. 16

FALSE.

Swift only allows boolean-like conditional operations. Of course, there is still implicit behavior such as “ if thisIsTrue()” as long as it’s of a boolean type. Swift does NOT allow this behavior with nil-values. For instance, in C any non-zero value is considered “true” and NULL values can be implicitly evaluated. Here is a comparison of what you could ***roughly*** do in Swift vs C if you had a List data structure.
In fact, this is likely to be part of the next Suite when we deal with classes/structures in Swift.

Swift:

```
let list: List? = List()
var head = list
while head != nil
{
    print(head.value)
    head = head.next
}
```

C:

```
List* list = newList();
List* head = list;

// implicitly iterate as long as head is not null
while ( head ) {
    printf("%d\n", head -> value)
    head = head -> next;
}
```

Q. 17

Optionals are a safe construct in Swift which provide a way to declare variables and even constants as nil. The idea is that if, at any time in your code a value can be nil, then it is to be declared an optional. Unlike pointers in C or default object values in Java for instance, they are NOT required all the time. You only need to use optionals where it's necessary if at all. These instances may be a search error in a function, or an error elsewhere, etc.

Q. 18

FALSE. You're allowed to have variables without values as long as you declare which type they are so the compiler knows what it is when you later use it. That's why the best-practice (type-inference) is not applicable in this instance. So if you attempt to use a variable before it's given a value, then Xcode will give you an error.

Q. 19

You can declare a variable without initial values in two ways. You can either use an optional, or you can use regular explicit types. In one sense, the variable will automatically be nil if you declare one as an optional without a value. Below I have two variables that follow this idea

```
var str: String? // automatically nil
var num: Int // compiler knows type, but shouldn't use yet.
```

Q. 20

Because the variable 'num' is an optional, it is currently "wrapped", as it were. In both conditional branches, the variable does get a value, but if you try to use it as-is, you'll see it as: Optional(40). Thus, in order to get rid of the error we want the guaranteed numeric value:

```
print(num! + 300)
```

This allows us to use the value stored within the optional since we are sure that it has one. In any other case, it would crash if the value remained unassigned (nil).

Q. 21

We use optional unwrapping when we're sure that the value we want to use (that is stored within the optional), contains a legal value. As stated above, remember that without unwrapping we would have Optional(40). However, since we're positive it contains a value, we can unwrap it and thus use the literal 40. Since we're designating to the compiler that it has a guaranteed value, we can't unwrap a nil value. If we attempt to use the unwrap operator on a nil value, the program crashes. This is a safe mechanism in swift - Safe being defined as a well-defined behavior for all operations.

Q. 22

As an addendum, I mean “Not the safest way to use optionals” as it can be dangerous to your program if you attempt to unwrap a nil value. The crash is a safety mechanism in Swift, though.

Q. 22a

As stated twice over, you use unwrapping when you are positive that the variable you are attempting to unwrap contains a value. Otherwise, your program can crash if you unwrap a nil.

Q. 22b

Conditional guarding means to use an if-else statement to ensure a value is not nil. Particularly this requires unwrapping after the condition is passed. In this way, we ensure our unwrap operation is legal.

```
var multiplier: Int? = nil
let base = 2
multiplier = 50

if multiplier != nil
{
    var res = multiplier! * base
    print(“Multiplying \(base) by \(multiplier!)
    is \(res)”)
}
```

Q. 22c

Optional binding is an alternative way to conditional-guarding, except it avoids having to use force unwrapping. Instead, if the conditional can be successfully assigned a value, you can use this variable/constant locally as a temporary variable. Otherwise, you have the “else” condition (if you’ve provided one) to handle if the value is nil.

```
var multiplier: Int? = nil
let base = 2
multiplier = 50

if let multiplier = multiplier
{
    var res = multiplier * base
    print(“Multiplying \(base) by \(multiplier)
    is \(res)”)
}
```

In this instance I redeclared the variable locally, which shadows/hides the one declared earlier. I don’t have a separate branch to handle a nil value, just if the assignment is successful.

Q. 22d

Optional chaining is the idea that when you call a property/method/etc, it has the potential to return nil. If anything within the chain is nil, it’s nil overall. If a call is at the last part of the chain or a particular part is not an optional, you don’t need to use the “?” to denote that it’s an optional value.

Notice that the string constant here is an optional, so to call the count property, I need to note that it is one by “?”. I don’t unwrap it, however, I could if I wanted to. IF I unwrapped it, the resulting variable/constant would no longer be an optional, and if anything is nil, it would crash.

```
let myStr: String? = “Let’s try this out!”
let strSize = myStr?.count //Returns optional int.
print(strSize!) //print the strSize
```

Q. 22e

True! Since they return nil overall (possibly), that means that they actually return an optional. As you see above, unwrapping it gets the value if it’s not nil. Hence, I could use optional binding instead to print the value without hazard:

```
let myStr: String? = “Let’s try this out!”
if let strSize = myStr?.count {
    print(strSize) //print the strSize
}
```

Q. 23

We declare a variable as optional when we know at some point that particular variable will or can be nil. Since the compiler doesn’t know what the type is without an initial value, it has to be supplied as part of the declaration. The exception to this is when you call a variable/constant’s properties, methods, etc, when the return type allows type-inference. That’s why simply declaring a variable with an initial value of nil is invalid, since the type itself is not known either explicitly or implicitly.

Q. 24

There are three places that need correcting. Notice that the value of our optional is nil since the block constant is true. In order to change multi, it has to be a ‘var’. Another issue (if num where not nil) is when we mutate the “num” variable, we are saying that we want to multiply the OPTIONAL by whatever “multi” is. Remember we can’t do things like this to an optional unless it is unwrapped. The second issue is in the alternative branch where num is nil. We are at this point unwrapping an optional variable which is nil - so this particular program will crash. You can choose to fix it a few ways, so long as the solution involves not unwrapping the nil value. Here is mine:

```
var multi = 8
let block = true
var num: Int?

if block {
    num = nil
} else {
    num = 20
}

if num != nil {
    multi *= 9
    num! *= multi //unwrap is fine since not nil
} else {
    print(“NUM is nil”)
}
```

Q. 25

Using the fixed version of our code and changing it to optional binding is:

```
var multi = 8
let block = true
var num: Int?

if block {
    num = nil
} else {
    num = 20
}

if var num = num {
    multi *= 9
    num *= multi //unwrap no longer needed
} else {
    print("NUM is nil")
}
```

The benefit to using optional binding as opposed to conditional guarding is that unwrapping isn't necessary and we can be sure that utilizing the variable is not going to cause a crash. That is, it's implicitly unwrapped if the assignment is successful (the var/constant isn't nil) to the local variable within the condition. Notice here I used an if-var instead of if-let since I would later mutate the variable within the block of code. Reminder that this is a local variable and only scoped within the if condition block. Also, I can redeclare it since local variables shadow/hide previously declared ones.

Q. 26

It’s worth mentioning that you can compare any optional that is still wrapped with a literal of the same unwrapped type. The second one produces a warning because you’re attempting to compare an unwrapped non-optional value to a optional-only nil value. You can fix it by removing the “!” to make it an optional once again, or use optional binding, etc.

Q. 27

```
var num: Int? = 98
var s: String? = “s”

let stringCnt: Any? = s?.count

TASK 1

if let num = num {

   if let stringCnt = stringCnt {
      print(“Num \(num) with (s!) having (strCnt) chars”)
   }

}




TASK 2

if let num = num , let stringCnt = stringCnt {
      print(“Num \(num) with (s!) having (strCnt) chars”)
}
```

Q. 28

By succeed, it means which statements will execute their respective blocks.

Succeed:

2, 4

Don’t succeed:

1, 3, 5

If any of them contained the nil value, the block doesn’t execute.

Q. 29

The problem is that, although I have the idea correct, it’s not going to work the way it’s presented here. Since I’ve redeclared the variable, the second use of ‘num’ refers to the first assignment in the optional binding statement. Therefore, since it’s implicitly a value now and not an optional, the second binding will fail. You can fix this by changing the variable name to anything else, so that they both refer to the original.

Q. 30

This one is interesting since theoretically it would work. There’s two ways you can fix this. First, since the str variable will no longer be an optional, then remove the optional chaining syntax and simply call a property/method/etc that returns an optional; however, since we want to keep the general behavior of this statement the same, we can instead just do ’s?.count’, which is valid. Technically the second statement is excessive, so it’s not even needed. You can call this property within the block without optional chaining.

Q. 31

Tip: Nil coalescence is just a shorthand way of using optionals with conditional guarding.

Q. 31a

```
var num: Int? = 98
var res: Any
```

Ternary:

```
res = (num != nil ? num! : “No Dice! 😕”)
```

Optional Binding:

```
if let num = num {
    res = num
} else {
    res = “No Dice! 😕”
}
```

Conditional Guarding:

```
if num != nil {
 res = num!
} else {
 res = “No Dice! 😕”
}
```

Q. 31b

Before showing the answer, it's important to point out a few things. Notice that the only two variables which I have the type declared for are 'preferredStyle' and 'userStyle'. Since 'preferredStyle' gets assigned a value later, I need to designate what type it is for its subsequent use. Since 'userStyle' is an optional, I need to designate that in order to unwrap it or compare it against nil.

```
preferredStyle = userStyle ?? defaultStyle
```

Translated, this simply means "If userStyle is not nil, preferredStyle gets assigned its value. If it is nil, the preferredStyle gets the defaultStyle value". While I showed the equivalence in terms of condition, here is the equivalence in ternary:

```
preferredStyle = ( userStyle != nil ? userStyle!
: defaultStyle )
```

Q. 31c

You can see that by using this, or even the ternary operator, it's an incredibly compact way of writing this code. So we took it from 5 lines down to 1. In real situations, you may want to be more particular about your variable choice. The 'userStyle' variable was unwrapped and assigned to 'preferredStyle' because 'preferredStyle' is not an optional value.

Q. 31d

Yes. I could instead make ‘preferredStyle’ an optional and use it later by unwrapping or using another one of the optional constructs we’ve seen.

Q. 31e

```
var preferredStyle: Style?
var userStyle: Style? = Style.Lightning
var defaultStyle = Style.Water

if userStyle != nil {
    preferredStyle = userStyle
} else {
    preferredStyle = defaultStyle
}
```

Q. 32

The problem in particular is assigning the ‘specialStr’ variable the value of ‘s3’, or rather the lack thereof. Since ‘s3’ is an optional type and ‘specialStr’ is not, we need to unwrap ‘s3’ to obtain that value. Keep in mind that unwrapping is fine here since ‘s3’ has a value. But if it were nil, then it would crash.

Unwrap:

```
specialStr = s3!
```

Binding:

```
if let s3 = s3 { specialStr = s3 }
```

Q. 33

Optional chaining can be used when you're assigning the result to a variable, which can include using it within binding. You can only use it if one of the calls in your chain returns a nil value. And finally, you don't need the '?' or '!' operators on the last value, or the first value as long as that first one isn't an optional.

Q. 34

You can only use it if you're right hand value to assign is an optional. Otherwise, you'll get an error.

Q. 35

Optionals are useful when you know for a fact that a variable can/will be nil at some point. Furthermore, nil values can help indicate errors. A decent example is linear search - if a value is not found (and depending on what you return), you could return an optional int that is nil.

Q. 36

When two variables share a reference, that means that you have a variable set to another. At that point they share memory of course, but if you change either one, the changed variable becomes its own reference. So no the reference will no longer be shared, as they are separate components in memory. If you want to see this in action yourself, try the following code:

```
var str1 = “One str”
var str2 = str1 // share the reference to “One str”

print(str1.hashValue)
print(str2.hashValue) // They will be the same obj.

str2.append(“ now changed”)

print(str1.hashValue)
print(str2.hashValue) // They will be different
```

This is demonstrable in Python just the same, but now try it with arrays. You’ll see that for Python the same object is mutated, but in Swift it’s an entirely new one.

Q. 37

Enumerations are usually used to have some kind of symbolic variable representing a specific type and value. For example, I can have an enum that contains integer values represented by colors of the rainbow. The difference in C’s and Swifts enums is that C only supports enums with integral (numeric) values. Swift allows you to have numbers, classes, properties, etc. Essentially, they are able to contain behaviors similar to that of a class.

Q. 38

The single restriction is that if you decide to assign raw values to your enums, they must all be of the same type. Although not particularly a restriction, but something you'll notice later, is that all raw values must also be unique.

The general syntax of an enum is as follows

```
enum <Name> : <Type if any> {
  case <val 1> = <Raw value if any>
  ...
  case <val N> = <Raw value if any>
}
```

Q. 39

A member value is the name you are providing with the case statements. A raw value is the value you assign to them if you have a type specified. Sometimes you won't always have a raw value available, but depending on the type of enum it can be inferred (such as an int enum).

Q. 40

Member values: String, Wind, Percussion
Raw values:  Piano, Flute, Drums

Q. 41

Just certain data types as long as they conform to the RawRepresentable protocol, so Strings or Int for example.

Q. 42

This isn't like the question asked earlier. The specific similarity I describe here is that the integer raw values can be implied. So it can be automatically provided, or you can specify what the value is so it can auto-increment.

```
enum Ex: Int {

 case one = 2
 case two
 case four
 case five

}
```

In this instance, I specified the member value 'one' to be the literal 2. That means that 'two' is auto-incremented to 3, and so forth.

Q. 43

TRUE. The type once you have initialized it is inferred to be 'Instruments', so you don't need to constantly type the Enum.Member every time, although you're welcome to. Here is the equivalence:

```
var inst = Instruments.Strings
inst = Instruments.Wind
```

Q. 44

TRUE. The raw values are the literal types you designated the enum to hold. The member values are part of the Enum as a type.

Q. 45

First, it assigns the value of ‘Instuments.Strings' to the ‘someInst’ variable. Then the value is reassigned to the new ‘.Wind’ variable which is inferred to be from the enum. The next function call will display what type the variable is.

Next, a switch statement is used to evaluate what the value of the variable is. This is a great example of the type inference in action. You can see that in the switch statement, we can just specify the member values of the enum instead of the whole thing. So, we look to see if it has a value of ‘.Wind’ and filter by looking at whether or not the raw value assigned to Strings is currently Guitar. This where clause is kind of like a ‘&&’ within a conditional, since the clause also has to be true in order to execute the block.

The differences between the two statements are not very many. In fact, they are almost identical. The only difference is that we aren’t using a ‘where’ clause. This time it’s using the ‘&&’ conditional operator.

Q. 46

Yes, someInst has access to the ‘rawValue’ property. Note the line that says “assume we don’t reassign someInst”. So we could have written this:

```
where someInst.rawValue == “Guitar”
```

It’s important to point out that the type inference trick only works in places where a direct comparison is taking place. Since ‘where’ is a separate clause, it’s not directly comparing the original value, so we have to use it here.

Q. 47

Typically we don’t have access to this property if the enum we are using is not adherent to a particular data type. In other words, those enums which don’t have a type specified.

Q. 48

```
enum Ranks : Int {
 case Bronze = 1
 case Silver // 2
 case Gold // 3
 case Platinum // 4
}
```

Q. 49

```
var response: String?
```

Q. 50

```
var rank: Ranks?
```

Q. 51

```
var found = false
for rate in rates {
 if rate.contains(“amazing”) {
    found = true
    break
 }
}

if found == true {
 rank = .Platinum
}
```

Q. 52

```
var tally = 0
for rate in rates {
 if rate.contains(“ok”) || rate.contains(“good”) ||
    rate.contains(“poor”) {
    tally += 1
 }
}

switch tally {
 case 1:
    rank = .Bronze
 case 2:
    rank = .Silver
 case 3:
    rank = .Gold
 case 4:
    rank = .Platinum
 default:
    break // I just put something here to make it valid
}
```

Q. 53

```
if let rank = rank {
 print(“This restaurant received a score of \(rank).”)
}
```

Q. 54

```
enum Majors : String {
  case cs = “Computer Science"
  case mus = “Music Studies”
  case socs = “Social Sciences”
  case psy = “Psychology”
  case eco = “Economics and Finance”
  case bus = “Business Studies”
}

func majorOpinions(major: Majors) -> Void {
  switch major {
      case .cs:
           print(“Computer Science is the best!”)
      case .mus:
           print(“Social Sciences is hard!”)
      case .socs:
           print(“Music Studies is cool!”)
      case .psy:
           print(“Psychology is interesting!”)
      case .eco:
           print(“Economics and Finance is FANCY.”)
      case .bus:
           print(“Business Studies is popular…”)
      default:
           print(“I don’t know what major that is 😕")
  }
}

var major = Ranks.cs
majorOpinions(major: major)
```

Q. 55

FALSE. You can’t use associated values if you have an enum with a type specification.

Q. 56

Yes. The Korean member value has associated values. You can fix it by assigning it a String raw value instead.

Q. 57

It’s a good way to aggregate symbolic variables of an identical group of items. The associated values allow an easy way to give data to these variables without the need of a class or structure. That’s pretty much why I like to think of them as mini-classes ( but they aren’t ).

Q. 58

```
enum Inventory {

  //Associated vals: Name, price, item count
  case Book(String, Double, Int)
  case Computer(String, Double, Int)
 case Phone(String, Double, Int)
  case Television(String, Double, Int)
  case Headphones(String, Double, Int)

}

// example variable
var item = Inventory.Phone(“iPhone XS”, 1010.65, 1)

// You can separate these with commas instead too.
switch item {
  case let .Book(name, price, cnt)
     fallthrough
  case let .Computer(name, price, cnt)
     fallthrough
  case let .Phone(name, price, cnt)
     fallthrough
  case let .Television(name, price, cnt)
     print(“Name: \(name), Price: \(price), Cnt: \(cnt)”)
  default:
     print(“Out of stock of this item!”)

}
```

Q. 59

This one is simple with type inference. Note that in Swift, arrays must be homogenous data unless you explicitly state otherwise.

```
var myArr = [1, 3, 4, 99, 18]
```

Q. 60

The problem here is that the compiler knows what the type will eventually be, but you haven't provided any space in memory for this object. Specifically, you never actually allocated it, so using it will cause an error. There are two main ways in Swift that you can declare and allocate an array of your choosing:

```
var myArr = Array<Int>()
```

or

```
var myArr = [Int]()
```

Notice the generic syntax. So instead of Int, you can use any type you'd like.

Q. 61

There are a few ways you can do this. I'd say the best way to do this is with optional chaining. Optional binding wouldn't be a great course of action because if you remember, the variable shares a reference until you mutate it. You can of course do a force unwrapping of the array instead of '?', but in this case it wouldn't serve much of a purpose. Doing it with optional chaining allows us to do the append only if our value isn't nil. Here is the way to do it: myArr?.append(…). I would *likely* use this with binding depending on the context.

Q. 62

The glaring problem here is that this array does *not* contain homogenous data. While this is allowed, the way it's done here isn't correct.

Q. 63

The only time you truly have to explicitly declare the type you are using for an array is when you are initializing it with heterogenous data.  Any other time, it is inferred via the initializations or the syntax you use to create the object.

Fix:

```
var mixyArray:[Any] = [“Hello there”, 4, “Friend”, 8.19]
```

The reason you cannot use 'AnyObject' is because not all of the values within the array are 'Objects', per say. We refer to objects when the values are represented by classes. Some other values, like integers, are actually represented by structures, which means they aren't objects ( i.e. objects are instances of a CLASS, not a structure ). You can identify the difference usually given by the documentation or as you are typing in Xcode, it will give you context clues. You'll notice little squares with an 'S' or 'C', along with many others.

Q. 64

```
var a = [Int?]()
let b = Array<Int>()
let c = Array<Array<Array<Double>>>()
var d = [[Double]]()
let e = [[Int]]() or Array<Array<Int>>()
```

Q. 65

No, not necessarily. The type will be inferred in that case as well.

Q. 66

```
var a = [[Any]]() or Array<Array<Any>>()
```

Q. 67

```
var a = [[1, nil, 88, 9], [2, nil, 99, 0], [2, 2]]
```

Q. 68

```
[[Int?]]
```

Q. 69

a. `myArr[…3]`
b. `myArr[5…8]`
c. `myArr[8…]`
d. `myArr[…] // or myArr is fine too.`
e. `myArr[…8]`
f. `myArr[2…]`

Q. 70

TRUE.

Q. 71

FALSE. Not with range slicing you can't.

Q. 72

In this context, it's useful since any operation you perform is guaranteed to be valid for any element within the array. For instance, if I wanted to iterate through an array and get the count of every string, I could call the property without worry of hitting an integer which doesn't have this property.

Q. 73

I also have my own example array to demonstrate this. You can use your own.

```
var arr1 = [1, "Hello".hashValue, 3, 12, 22]
var arr2 = [99, 2, 98, 120, 21]

a. arr1 + arr2 // Give to new var
b. arr1 += arr2 // Mutate original
c. arr1.append(contentsOf: arr2)
```

Q. 74

FALSE. When I initially wrote the question, there would be an error with attempting to use an Any array after initializing. It seems to no longer be the case.

Q. 75

Since there is no apparent issue, there is no need to fix anything here.

Q. 76

TRUE. You'll notice that if you attempt to use an optional-like operator at the end of a call chain, it will give you an error saying it's not necessary.

Q. 77

Reminder that typically the modulus operator restricts maximum numbers to one less than the divisor.

```
var a = Array<Int>()
//Insert to back
for _ in 0..<10 {
 let ran = Int(arc4random() % 9)
 a.append(ran)
}

// Remove from front = FIFO
while !a.isEmpty {
 print(a.removeFirst())
}
```

Q. 78

The insert methods takes the element you want and inserts it literally in the index you specify, effectively replacing what's there. Whatever is there ends up being pushed backward.

```
var myArr = [ 0, 1, 8, 2, 3, 4, 5]
```

Q. 79

```
var a = Array<Int>()

//Insert to "top", where the back is the top.
for _ in 0..<10 {
 let ran = Int(arc4random() % 9)
 a.append(ran)
}

while !a.isEmpty {
 if let val = a.popLast() {
    print("Popped \(val)")
 }
}
```

Q. 80

Reverse performs the reverse operation in-place. This means that the original array is changed. Reversed performs the operation but returns a NEW version of this array that has been revered.

Q. 81

This is identical to the last question. Sort does this to the original array, and sorted returns a new version of this array which is sorted.

Q. 82

This is a peek into functional programming. If you're familiar with how lambdas work, then this is a similar construct. The first closure ( $0 < $1 ) indicates that we want the first value to be less than the second value. In total, this will end up having a strictly increasing array. The second ( $1 < $0 ) indicates that we want the second value to be less than the first value. This will result in a strictly decreasing array. Closures will be seen later.

Q. 83

The difference is that the filter method will only retrieve the values from the array that match your boolean within the closure. This will return a new filtered array. Map is similar in that it returns a new array, but it's based off of a transform statement you provide. That, or you return your values if it's not a one-liner.

```
var myArr = [1, 2, 99, 0, -3, 2, 100]

// new filtered array, evens less than 100.
myArr.filter(){ $0 % 2 == 0 &&  $0 < 100 }
// new array which modifies the values by * 56
myArr.map(){ $0 * 56 }
```

This also demonstrates why homogeneous arrays are important. Without the same type, we wouldn't be able to guarantee the comparison is valid for every value.

Q. 84

```
var myArr = [1, 2, 99, 0, -3, 2, 100]

myArr.forEach(){ print($0) }
```

Q. 85

```
var myArr = [“Scott”, “Doctor”, “Piano”,
“Guitar”, “Engineer”, “Control”, “Tempo”]

myArr.filter(){ $0.contains(“o”) }

myArr.forEach(){
 if $0.contains(“o”) {
    print($0)
 }
}
```

Q. 86

```
for str in myArr {
 if str.contains(“o”) {
    print(str)
 }
}

var i = 0
while i < myArr.count {
 if myArr[i].contains(“o”) {
    print(myArr[i])
 }
 i += 1
}
```

Q. 87

```
var myArr = [1, 2, 99, 0, -3, 2, 100]

for (x,y) in myArr.enumerated() {
 print("At index \(x), array has \(y) stored.")
}
```

Q. 88

This problem persists in a lot of languages that support it. The issue is that the variable you are using to iterate is simply a temporary shared reference to that location. When you attempt to mutate it, it's not a permanent change as you're only changing the temporary reference. In Swift, it doesn't let you do this anyway unless you do 'for var __ in collection'. Moreover, mutating a variable in Swift creates a new reference for the changed value.

Q. 89

This isn't a comprehensive list of equivalencies, by the way. You can write them in many other ways.

```
var dic = [String: Int]()
var dic2 = [String: [String: String]]()
var dic3 = Dictionary<Double,Array<Int>>()
var dic4 = [String:Double]()
```

Q. 90

```
var dic = [“one”:1,”two”:2,“three”:3]

//Accessing this one is weird! Feel free to try.
//It’s weirder as a one-liner.
var dic2 = [“dic”:[“one”:”eins”], “dic2”:
[“two”:”zwei”]]

var dic3 = [1.0:[1, 2, 3], 2.5:[3,4,5]]
var dic4 = [“version1”:1.1, “version2”:2.5,
“version3”:3.89]
```

Q. 91

```
//I like using the generic syntax but you can of
//course do [String:Int]()
var dic = Dictionary<String, Int>()
```

Way 1:

```
//Returns an optional val. Specifically if it had
//something before you may want it.
dic.updateValue(3, forKey: “Three”)
```

Way 2:

```
dic[“Three”] = 3
```

Q. 92

An optional is returned because the key/value pair you are attempting to access may not exist yet or at all. Naturally this behavior is different than arrays because you aren't indexing them the same way. ( But python does halt the program on invalid access so it depends on implementation ) Also on that note, assigning a nil value to a key removes it from the dictionary. Force unwrapping may be fine if you are positive that the value you have actually exists, but remember this might not be a good idea since it can crash your program. A better way of handling this is checking to see if what is returned is nil. This can be done in one of the many ways we've seen already. This is a good example where Swift allows you to handle this gracefully.

Q. 93

Way 1:

```
dic["Three"] = nil
```

Way 2:

```
dic.removeValue(forKey: "Three")
```

This may seem like the same question as earlier, but it's important to think about why this happens. Removing or updating the value returns an optional to you since the value for that key may or may not exist ( hence, the key itself might not even exist ). That's why it's usually good to think of nil values as indicators of something being non-existent. Arrays for instance, are well-defined ordered and contiguous portions of memory, so you're restrained to the memory you have allocated for your collection.

Q. 94

Before I show any kind of iteration, it's important to know that you can iterate through a dictionary in a lot of ways. This even includes the value/key collections if you decide to access those. Here I just do it in the basic way.

```
var dic = [“one”:1, “two”:2, “three”:3]

for (k, v) in dic {
 print(“For \(k) we have \(v)”)
}

dic.forEach() {
 print(“For \($0.key) we have \($0.value)”)
}
```

Q. 95

This is a for-in loop iterating over a given range. The '_' implies an invisible variable and we have an unbounded range. So in this case, it's an infinite loop. The loop doesn't know where to stop since there is no upper-bound.

Q. 96

```
var s = Set<Int>()
```

Q. 97

```
// You can do however many you want, however you want.

for _ in 1…100 {
 s.insert(Int(arc4random() % 10))
}
```

You'll probably notice that it's a lot smaller than you expect. This is because it's a SET. So if any duplicates are encountered, they aren't placed into the set. If not, then you got lucky with the random numbers.

Q. 98

```
var s1 = Set<Int>([1, 2, 9, 10, 13, 12])
var s2 = Set<Int>([0, 2, 88])

// S1 ∪ S2
s1.union(s2)

// S1 ∩ S2
s1.intersection(s2)

// S1 - S2
s1.subtract(s2)

// S2 - S1
s2.subtract(s1)
```

Q. 99

The as keyword is a way to cast values to another type as long as they share the same inheritance hierarchy. That is if they effectively share the same ‘is-a’ relationships somewhere.
No, it’s not always necessary since there are plenty of convenience conversion initializers; it just depends on the need.

Q. 100

Type-Casting; Up-casting; Down-casting;

Q. 101

Here we use ‘as?’ because we are attempting to cast an Objective-C NSArray to Swift’s array ( of ints ). In this case, NSArray ( or NSMutableArray for that matter ) can have heterogenous data. That being said, the cast can fail if any one of the elements can’t match what we want. So let’s take a look at our code here.

We have an array constant of type NSArray from Objective-C. Next we have a nil coalescence statement. Breaking it down, observe the casting with the ‘as’ operator. We want to cast our NSArray to Swift’s int array, but if any of the elements within the NSArray are not an integer, the cast can fail, which is why we use the ‘?’. A failure can return a nil, so I can use any optional construct I choose to handle this.

In this particular piece of code, the cast ends up failing because there are elements within my constant which are not integers. This returns a nil, which in turn triggers the nil-coalescence operator to assign the [1, 2, 3] array to ‘b’ instead.

Q. 102

```
var ns: NSString = “My String”
var s = ns as String
```

Q. 103

Before giving the example, it’s worth reiterating why we can use the ‘!’ here. We know that with optionals, unwrapping is fine when we are absolutely sure a value within the optional exists. As for type casting, we need to be sure that the cast is 100% safe. For instance, in Objective-C an NSNumber can either be a floating-point or a normal integer. If I do a ‘!’ on my cast, then the value to which I cast has to match.

```
var nsnum: NSNumber = 78.2
var n = nsnum as! Double
```

If I casted to an Int, then I would need to use ‘?’ otherwise it will fail.

Q. 104

The issue is that I’m forcefully casting the NSDictionary into the Dictionary<Int, String>. Notice that one of the keys within the ‘a’ constant is a string and NOT an integer, which would cause the cast overall to fail. This is another reason why type enforcement is important.

```
let a: NSDictionary = [1:”One”,2:”Two”,”Three”:3]
let b = a as? [Int:String]
```

Q. 105

```
var strs = [“Netflix”, “Doctor who”, “Love, Simon”, “Star
Wars”, “Harry Potter”, “iPhone”, “Pentatonix”]

var reps = Dictionary<Character, Int>()

reps[“A”] = 4
reps[“L”] = 1
reps[“E”] = 3
reps[“T”] = 7
reps[“G”] = 6
reps[“S”] = 5
reps[“B”] = 8
reps[“O”] = 0
reps[“P”] = 9

// Same for the lowercase letters.

// Here we map over the sequence elements and change them

strs = strs.map() {

  var str = $0 // Save a reference to the string

  // Iterate over the characters in each string.
  str.forEach(){

     // Remember the optional return from dicts.
     // Here I use optional binding and then access the char
     // if it exists.

     if let num = reps[$0] {
          // Replace the string with our new one.
          str = str.replacingOccurences(of: String($0),
          with: String(num))
     }

  }

  return str // Return the string into the new sequence.
}
```

A quick note on the replacingOccurences method: If you attempt to replace a string or anything which does not exist within the string, the same one will be returned. In other words, nothing happens.

My solution is really funky, but you could have done it with while loops just the same.

```
var i = 0
while i < strs.count {
  var j = 0
  while j < strs[i].count {
      let in = strs[i].index(strs[i].startIndex, offsetBy: j)
      if let num = reps[strs[i][in]] {
           strs[i].replacingOccurences(of: String(strs[i]
           [in]), with: String(num))
      }
      j += 1
  }
  i += 1
}
```

Q. 106

Tuples are essentially immutable collections. That is, you can't remove or add elements, but you CAN change them. So, TRUE they allow heterogeneous data but can't be mutated like arrays would for example.

Q. 107

```
var t = ("Donut", 12, "Glazed", 99.0, "Choco")
t.0 = "Doughnut"
t.1
t.2 …
```

Q. 108

We have the ability to name the elements within a tuple. You can name some or all of them. This allows us to, as the question states, replace a class/structure when we don't need methods to act on our data. Here is an example:

```
var t = (Food: “Donut”, Cals: 12, Type: “Glazed”,
Price: 99.0, Flavor: “Choco”)
```

Q. 109

TRUE

Q. 110

```
for (k, v) in d {
 print(d[k]!)
}

for (k, v) in d {
 if let val = d[k] {
    print(val)
 }
}

for (k, _) in d {
 print(d[k]!)
}

for (k, _) in d {
 if let val = d[k] {
    print(val)
 }
}
```

Q. 111

```
var (n, n2, s, s2) = t
```

Q. 112

Switch statements in C are limited to just integral values. Also, there is an implicit fallthrough, whereas with Swift there is none. Swift also allows you to perform much more complex conditions for your case statements, and even more. At times it can be more simplistic to use than if-else branches.

Q. 113

This is necessary in those languages because otherwise they will fall through to the next case. In Swift, you have to explicitly declare ‘fallthrough’ to get this behavior, so a break statement isn’t needed.

Q. 114

FALSE. You can either make it exhaustive ( outlining every possible case ) or utilize the default case to catch everything else. If you do make it exhaustive, the default case isn’t necessary.

```
enum Exhaust {
  case Blue
}

var a = Exhaust.Blue

switch a {
  case .Blue:
     print(“BLUE!”)
}
```

Q. 115

‘where’. This allows you to add an additional clause to your case statements. This acts similar to an ‘&&’ conditional operator, since it needs to be true as well in order for the block to be executed.

Q. 116

Enum member values must all have UNIQUE raw values. The issue is that ‘a’ and ‘b’ have identical strings, which is an error.

## Q. 117

```
func swimmingInCode(tup: (name: String, Int), lock: Bool?) -> Void {
    var lock = lock ?? true
    let ns = [“Scott”, “Mitch”, “Matt”, “Kirstin”, “Kevin”]
    switch tup {
      case let (str, age) where (str == “Scott”) && (lock != true):
           print(“Got \(str) who is \(age) years old.”)
      case let (str, age) where (str == “Mitch”) && (lock != true):
           print(“Got \(str) who is \(age) years old.”)
      case let (str, age) where (str == “Matt”) && (lock != true):
           print(“Got \(str) who is \(age) years old.”)
      case let (str, age) where (str == “Kirstin”) && (lock != true):
           print(“Got \(str) who is \(age) years old.”)
      case let (str, age) where (str == “Kevin”) && (lock != true):
           print(“Got \(str) who is \(age) years old.”)
      default:
       if !ns.contains(tup.name) {
           print(“Unknown name or locked!”)
       }
     }

}
```

## Q. 118

The issue is that I'm repeating the code way too much. They are all identical except for the string checking.

```
func swimmingInCode(tup: (name: String, Int), lock: Bool?) -> Void {
    var lock = lock ?? true
    let ns = [“Scott”, “Mitch”, “Matt”, “Kirstin”, “Kevin”]
    switch tup {
      case let (str, age) where (str == “Scott”) && (lock != true):
           fallthrough
      case let (str, age) where (str == “Mitch”) && (lock != true):
           fallthrough
      case let (str, age) where (str == “Matt”) && (lock != true):
           fallthrough
      case let (str, age) where (str == “Kirstin”) && (lock != true):
           fallthrough
      case let (str, age) where (str == “Kevin”) && (lock != true):
           print(“Got \(str) who is \(age) years old.”)
      default:
       if !ns.contains(tup.name) {
           print(“Unknown name or locked!”)
       }
      }

}
```

## Q. 119

No, the switch statement isn't really necessary. Here I have two ways of doing it.

Way 1:

```
func swimmingInCode(tup: (name: String, Int), lock: Bool?) -> Void {
      var lock = lock ?? true
      var found = false
      let ns = [“Scott”, “Mitch”, “Matt”, “Kirstin”, “Kevin”]
      ns.forEach(){
            if tup.name == $0 && lock != true {
                  print(“Got \(tup.name) who is \(tup.1) years old.”)
                  return
            }
      }
      if !found {
            print(“Unknown name or locked!”)
      }

}
```

Way 2:

```
func swimmingInCode(tup: (name: String, Int), lock: Bool?) -> Void {
      var lock = lock ?? true
      let ns = [“Scott”, “Mitch”, “Matt”, “Kirstin”, “Kevin”]
      for name in ns {
            if tup.name == name && lock != true {
                  print(“Got \(tup.name) who is \(tup.1) years old.”)
                  return
            }
      }
      print(“Unknown name or locked!”)
}
```

Q. 120

You can choose whichever variable you want to unpack on, or do all of them. I show just one here. You can use the where clause or just inner if statements if you want. Notice that using the ‘where’ clause will limit the exhaustive cases, so you need the ‘default’ case.

```
switch phone {

  case let .iPhone(name, model, price) where
              name.contains(“o”) || name.contains(“a"):
     fallthrough
  case let .iPad(name, model, price) where
              name.contains(“o”) || name.contains(“a"):
     fallthrough
  case let .Mac(name, model, price) where
              name.contains(“o”) || name.contains(“a"):
     print(“Name: \(name) Model: \(model) Price: \(price)”)
  default:
     print(“We don’t have a device like that”)

}
```

Q. 121

There’s not a particular difference aside from the syntax of it. The similarity is that the loop guarantees at least one execution every time. Another thing is that you don’t need to prime the loop as you would with a normal ‘while’ loop. That is, the update variable doesn’t need to be initialized beforehand.

```
var i: Int
repeat {
    i = 0
    print(“ONE EXECUTION!!”)
} while (i != 0) // finishes after one iteration
```

Q. 122

```
var someNum = 90
switch someNum {
    case 0..<10, 10..<100 where someNum % 2 == 0,
    200...500:
      print(“This is a nice number!”)
    default:
      print(“This is an uncool number!”)
}
```

Q. 123

We have a tuple which we use a switch on with multiple bindings. In addition, we want to filter the gpa stored within our tuple after having it bound to the gpa constant. In this case, the second one gets matched because gpa is < 3.2.

Q. 124

```
let ts = [(“One”,1),(“Two”,2),(“Three”,3),(“Four”,4)]
for t in ts where t.1 % 2 == 0 {

     print(“\(t.1) is worded as \(t.0)”)
}
```

Q. 125

In the last question, it reduces the amount of code we have. In other situations, it may help filter what we want and provide additional conditions where normal if-statements may not be applicable.

Q. 126

For-case allows us to filter a collection based off of the structure of the ‘case’. So, it filters the collection first essentially and then we iterate through what has been filtered. In our example, all of the elements that adhere to this case ( i.e. the ones with “x” as the first value in the tuple ), are filtered through and we iterate those. The ‘let y’ constant is found and we print it.

Q. 127

```
let names = [“Matt Smith”, “Peter Capaldi”, “David
Tennant”, “Tony Stark”, “Jeff Bezos”]

let tups = Array<(name: String, num: Int)>()

for _ in 1...101 {

  let name_ind = Int(arc4random()) % names.count
  let num = Int(arc4random()) % 101

  tups.append((names[name_ind], num))

}

for case let (name, num) in tups where num % 2 == 0
&& num > 0 {

  print(“Name \(name) has num \(num)”)

}
```

Q. 128

```
for case let (name, num) in tups where name.contains(“o")
  print(“Name \(name) has num \(num)”)
}
```

Q. 129

```
for case let (name, num) in tups where num == 2
  print(“Name \(name) has num \(num)”)
}
```

Q. 130

This is similar to optional binding in the way it works. Specifically, I’m seeing if the value I’m attempting to assign matches the pattern given in the case. If it does, then it prints the number. Here, it does execute successfully.

Q. 131

This is a shorthand notation of finding which values within the array/collection have a non-nil value. The equivalent way of writing this is choice A.

Q. 132

```
for n in myArr {
    if let n = n {
        print(n)
    }
}
```

Q. 133

Again you can choose whichever one you want to unpack.

```
if case let .Accounting(emps) = b1 {
 ...
}
```

The downside to using a switch statement is having to exhaustively outline the cases for the enum, especially if you just need one of them as shown here. Below is the exhaustive switch statement version ( that is if you don't utilize the default statement to catch all other cases )

```
switch b1 {
 case let .Accounting(emps):
    …
 case let .MusicSchool(emps):
    …
 case let .RecordingLabel(emps):
    …
 case let .StockBrokers(emps):
    …
}
```

Q. 134

TRUE

Q. 135

TRUE

Q. 136

They are passed in as an array of this type. They are so easy to use because you just have to access the array.

Q. 137

By default, all parameters are passed in as constants to the function, so they cannot be changed. One way to get around this is by use of a local variable to share the reference to this parameter. Furthermore, if you modify it the parameter variable does NOT get changed. A new copy is created for the local variable.

Q. 138

Yes this is correct.

Q. 139

Since the parameter labels are not specified to be omitted, you have to provide them within the call.

Q. 140

```
func addEm(addNum a: Int, withNum b: Int) -> Int {
    return a + b
}

addEm(addNum: 78, withNum: 2)
```

The explicit argument labels are ‘addNum’ and ‘withNum’. They are only used for the programmer and not within the function.

Q. 141

Argument labels serve as a way to make the function calls more sentence-like. If you decide to use explicit labels as above, they are only ever used when you call the function. If you don't use them, then the parameters themselves will be the argument labels by default. Of course, you can omit these altogether if you want to.

Q. 142

I have these ordered in the way the question has them.

```
func add(_ a: Int, _ b: Int) -> Int {
  return a + b
}

func add(firstVal a: Int, secondVal b: Int) -> Int
{
  return a + b
}

func add(_ a: Int, withNum b: Int) -> Int {
  return a + b
}

func add(a: Int, b: Int) -> Int {
  return a + b
}
```

Q. 143

TRUE. Arrays, Tuples, Really any collections, Structures, Classes, etc.

Q. 144

Guard statements are very similar to if and if-let statements. They allow you to handle failure statements earlier in your code. For instance, if you have an if-else, the guard statement will allow you to deal with the ‘else’ block first if the condition fails. Otherwise, the rest of the code executes if the condition succeeds. In addition, they can only be used within functions and provide a change-of-control statement.

Q. 145

```
guard (a >= 100) || (a == b) else {
    print(“My condition failed! 😭”)
    return
}

// if a >= 100 or a == b, this will execute.
print(“What a success!”)
```

Q. 146

```
guard (str != nil) && str!.contains(“hello”) else {
    print(“OH NO! STR IS NIL”)
    return
}

print(“This is the string you’re looking for!”)
```

Q. 147

```
guard let _ = str, let _ = str?.contains(“Hello”) {
    print(“OH NO! STR IS NIL”)
    return
}

print(“This is the string you’re looking for!”)
```

Q. 148

TRUE. Because both statements rely on the optional binding being successful. Whether or not the string contains “hello” is irrelevant. So if it’s nil regardless, it will fail.

```
guard let _ = str?.contains(“Hello”) {
    print(“OH NO! STR IS NIL”)
    return
}

print(“This is the string you’re looking for!”)
```

Q. 149

For space sake, I only provide the conditional guarding version for the first function type. Afterward I just use the guard statements.

```
// Explicit argument labels

func varAdd(numOne a: Int?, numTwo b: Int?, nums c: Int?…)
{

  if a == nil {
     print(“First param is nil”)
     return
  }

  if b == nil {
     print(“Second param is nil”)
     return
  }

  for case let n? in c {
     print(“\(a!) + \(b!) * (n) = \(a! + b! * n)”)
  }


}
```

```
// Implicit argument labels

func varAdd(a: Int?, b: Int?, c: Int?…) {

 guard a != nil else {
    print(“First param is nil”)
    return
 }

 guard b != nil else {
    print(“Second param is nil”)
    return
 }

 for case let n? in c {
    print(“\(a!) + \(b!) * (n) = \(a! + b! * n)”)
 }


}

// Omitted argument labels

func varAdd(_ a: Int?,_ b: Int?,_ c: Int?…) {

 guard a != nil else {
    print(“First param is nil”)
    return
 }

 guard b != nil else {
    print(“Second param is nil”)
    return
 }

 for case let n? in c {
    print(“\(a!) + \(b!) * (n) = \(a! + b! * n)”)
 }

}
```

Q. 150

I again am only showing one version of both. I commented the proposed alternative.

```
func fib(of a: Int) -> Int {
 // If a <= 35 execute stuff after this block.
 // otherwise, execute error within block.
 guard a <= 35 else {
    print(“ERROR! a > 35”)
    return a
 }

 // If a is 0, return 0
 guard a != 0 else {
    return 0
 }

 // If a is 1, return 1
 guard a != 1 else {
    return 1
 }

 return fib(of: a - 1) + fib(of: a - 2)
}
```

```
func fib(a: Int) -> Int {
 // If a <= 35 execute stuff after this block.
 // otherwise, execute error within block.
 guard a <= 35 else {
    print(“ERROR! a > 35”)
    return a
 }

 // If a is 0, return 0
 guard a != 0 else {
    return 0
 }

 // If a is 1, return 1
 guard a != 1 else {
    return 1
 }

 return fib(of: a - 1) + fib(of: a - 2)
}
```

```
func fib(_ a: Int) -> Int {
 // If a <= 35 execute stuff after this block.
 // otherwise, execute error within block.
 guard a <= 35 else {
    print(“ERROR! a > 35”)
    return a
 }

 // If a is 0, return 0
 guard a != 0 else {
    return 0
 }

 // If a is 1, return 1
 guard a != 1 else {
    return 1
 }

 return fib(of: a - 1) + fib(of: a - 2)
}
```

Q. 151

```
func fib(of a: Int) -> Int {

 // here we can actually increase the limit
 guard a <= 35 else {
    return a
 }

 guard a != 0 else {
    return 0
 }

 guard a != 1 else {
    return 1
 }

 var vals = [0, 1]

 for i in 2...a {
    vals.append(vals[i-1]+vals[i-2])
 }

 return vals[a]

}
```

```swift
func fib(a: Int) -> Int {

  // here we can actually increase the limit
  guard a <= 35 else {
     return a
  }

  guard a != 0 else {
     return 0
  }

  guard a != 1 else {
     return 1
  }

  var vals = [0, 1]

  for i in 2...a {
     vals.append(vals[i-1]+vals[i-2])
  }

  return vals[a]

}
```

```swift
func fib(_ a: Int) -> Int {

  // here we can actually increase the limit
  guard a <= 35 else {
     return a
  }

  guard a != 0 else {
     return 0
  }

  guard a != 1 else {
     return 1
  }

  var vals = [0, 1]

  for i in 2...a {
     vals.append(vals[i-1]+vals[i-2])
  }

  return vals[a]

}
```

Q. 152

```
var vals = [0:0, 1:1]

func fib(of a: Int) -> Int {

 guard a <= 35 else {
    return a
 }

 guard let val = vals[a] else {
    // if the assignment was nil here, then we calc.
    vals[a] = fib(of: a - 1) + fib(of: a - 2)
    return vals[a]!
 }

 // If the assignment was not nil, then we can return
 // what was found.

 return val
}
```

```swift
var vals = [0:0, 1:1]

func fib(a: Int) -> Int {

 guard a <= 35 else {
    return a
 }

 guard let val = vals[a] else {
    // if the assignment was nil here, then we calc.
    vals[a] = fib(of: a - 1) + fib(of: a - 2)
    return vals[a]!
 }

 // If the assignment was not nil, then we can return
 // what was found.

 return val

}
```

```swift
var vals = [0:0, 1:1]

func fib(_ a: Int) -> Int {

 guard a <= 35 else {
    return a
 }

 guard let val = vals[a] else {
    // if the assignment was nil here, then we calc.
    vals[a] = fib(of: a - 1) + fib(of: a - 2)
    return vals[a]!
 }

 // If the assignment was not nil, then we can return
 // what was found.

 return val
}
```

Feel free to change the limit of the passed in variable. For me it was 90 before the playground couldn’t take it anymore.

Q. 153

First, It's worth tracing the function as to what it does. We have a guard statement which protects against nil using a form of optional binding. If the assignment is successful ( aka if the value assigned from a is NOT nil ), then the print statement at the end of the function is executed. If the assignment fails ( aka if the value assigned from a IS nil ), then the print statement within the guard is executed. Below I will point out which function calls are valid by stating their implicit number.

1, 2, 3, 4 ( So all of them are valid ).

Some of them look like they wouldn't be valid, but notice how the 'b' variable has a default value, so using it within the function call isn't necessary.

Q. 154

For this one I won't provide function alternatives. I did, however, provide a sample function call afterward.

```
func binarySearch(_ arr: [Int]?, _ val: Int) -> Int? {

 guard let arr = arr else {
    return nil
 }

 func isSorted(arr: [Int]) -> Bool {

    if arr.count == 1 {
        return true
    }
```

```
    if arr.count == 2 && arr[0] < arr[1] {
        return true
    } else if arr.count == 2 && arr[0] > arr[1] {
        return false
    }

    if arr[0] > arr[1] {
        return false
    }

    if arr[arr.count - 1] < arr[arr.count - 2] {
        return false
    }

    var sorted = true
    var i = 1
    while i < arr.count - 1 {
        if arr[i-1] > arr[i] || arr[i] > arr[i+1] {
            sorted = false
            break
        }
        i += 1
    }

    return sorted

}


if !isSorted(arr: arr) {

    arr.sort()

}

var low = 0
var high = arr.count
var mid = (low + high) / 2
```

```
  while low <= high {

     if val < arr[mid] {
         high = mid - 1
     } else if val > arr[mid] {
         high = low + 1
     } else {
         break
     }
     mid = (low + high) / 2

  }

  return mid

}

var myarr: [Int]? = [1, 33, 2, 0, 13, 3]
let val = 1
var ind = binarySearch(myarr, val)
if let ind = ind {
  print("Index is \(ind)")
}
```

Q. 155

For this one assume that the nested ‘isSorted()’ function still exists. For space I’ve condensed this function. Here may be a great place to use argument labels, too. In fact, that’s why they are great, especially when you have a large parameter list. It disambiguates the function call.

```
func binarySearch(_ arr: [Int]?, _ val: Int) -> Int? {

  guard let arr = arr else {
     return nil
  }

  // isSorted() function still goes here.

  func binSearchRec(_ arr: [Int], _ val: Int, _ high:
                        Int, _ low: Int) -> Int {

     let mid = (low + high) / 2

     if arr[mid] == val {
          return mid
     }

     if arr[mid] < val {
          return binarySearchRec(arr, val, low, mid - 1)
     }

     return binarySearchRec(arr, val, mid + 1, high)

  }

  return binSearchRec(arr, val, 0, arr.count)

}
```

Q. 156

Here I don't provide function alternatives ( and I won't for the rest of them since they are identical in functionality ), but the argument labels are great here when I call my function.

```
func findStudent(inDatabase dic: (name: String, gpa:
Double, major: String), withId name: String) -> Bool {

  guard let value = dic[name] else {
     // If assignment fails, we do this stuff
     print("Student \(name) is not enrolled.")
     print("Enrolled students: ")
     dic.forEach(){
          print($0.value.name)
     }
     return false
  }

  // If assignment is NOT nil, we can do this stuff.

  print(value)
  return true

}

findStudent(inDatabase: dic, withId: "tik781")
```

Q. 157

```
func findStudent(inDatabase dic: (name: String, gpa:
Double, major: String), withMajor name: String? = nil,
andGpa gpa: Double? = nil) -> Bool {

  guard let _ = major else {
     // If assignment fails, we do this stuff
     print("Major not provided")
     return false
  }


  guard let _ = gpa else {
     // If assignment fails, we do this stuff
     print("gpa not provided")
     return false
  }

  guard !dic.isEmpty else {
     // If assignment fails, we do this stuff
     print("Database is empty")
     return false
  }


  // If assignment is NOT nil, we can do this stuff.
  for case (let name, gpa, major) in dic.values {
     print("Found \(name)")
  }

  return true

}
```

Q. 158

The modulus operator wraps values around by restricting it to the max of the divisor. For instance, if the index were 4 and the array size were 5, then the result is 4 % 5 = 4. If it's 5, then it becomes 0. So the max index is, of course, 4.

```
func searchAndReplace(array arr: inout [Int], atIndex i:
                        Int, withValue val: Int) {

 arr[i % arr.count] = val

}
```

Q. 159

```
func stepper(_ arr: [Int], step: Int = 1) {

 var step = step
 var i = 0
 while i < arr.count {
    if arr[i] % 2 == 0 {
        step += 1
    }
    print(arr[i])
    i += step
 }

}
```

Q. 160

Swift provides a way for you to do File IO in a C-like manner. This question is kind of tricky because it deals with error handling (which hasn't been dealt with yet).

```
var lines: [String]
do {
     let file = try String(contentsOfFile: “<file path>”)
     lines = file.split(separator: “\n”).map(){String($0)}
} catch {
     print(“File not found”)
}
```