Swift Essentials

Learn the Swift Programming Language
for iPhone, iPad, Apple Watch, macOS, and more

</b

codebaltimore

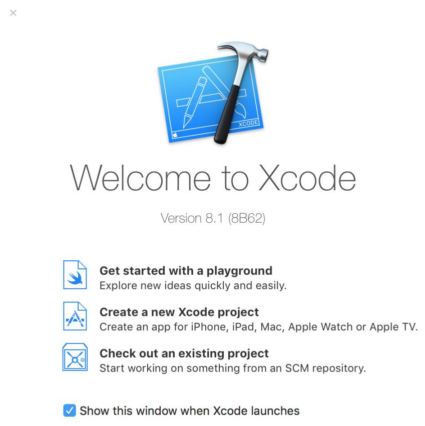
Welcome!

Welcome to Swift Essentials presented by Code Baltimore - a no-cost code bootcamp designed to teach real world skills through project based learning. We've designed the Swift Essentials book to teach you enough of the basics to get you through our project walkthroughs like the tip calculator or the shareable list app. Swift is an extremely powerful language with a simple syntax that is fairly easy to learn, especially for beginners, however this book is by no means an exhaustive look at the language. There will be a lot of concepts that we do not cover that can be helpful to you but if your goal is to build apps as quickly as possible and/or to follow along on our project walkthroughs, this guide will get you where you need to go!

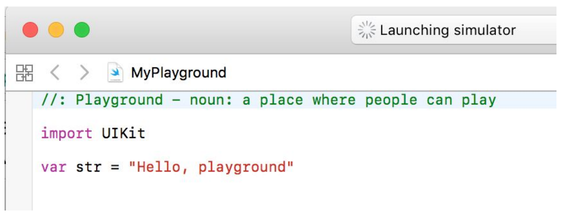
While you can just read this booklet to get a good idea of the language, we suggest you follow along and attempt the challenges at the end of each section. To properly follow along, you'll need a computer running macOS (most any Mac computer will do) and the free program Xcode. Xcode can be found in the App Store or at https://developer.apple.com/xcode/. Xcode is ONLY a mac app and you'll need a mac to develop any iPhone, iPad, Apple Watch, Apple TV, and macOS app.

Once you download Xcode, the easiest way to follow along is to open an Xcode Playground. Xcode Playgrounds allow you to write code that is automatically compiled, showing you errors right away and letting you easily create runnable functions. While you won't necessarily need playgrounds once you start building real apps, the playgrounds are an amazing way to play around with the language yourself.

Once you download Xcode, open it up and you'll be presented with a screen similar to this.


Welcome to Xcode
Version 8.1 (8B62)
Get started with a playground
Explore new ideas quickly and easily.
Create a new Xcode project
Create an app for iPhone, iPad, Mac, Apple Watch or Apple TV.
Check out an existing project
Start working on something from an SCM repository.
Show this window when Xcode launches

Tap "Get started with a playground", name it, and make sure the platform is "iOS". Save it wherever is convenient for you and press "Create".



```
//: Playground - noun: a place where people can play

import UIKit

var str = "Hello, playground"
```

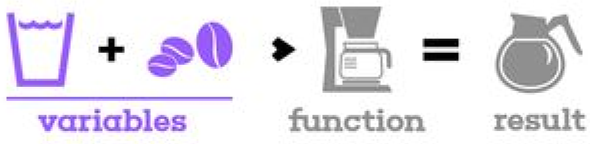
Follow along and write your code underneath the "import UIKit" statement. You may delete the var str = "Hello playground" line if you wish. It's not needed.

With any questions, direct them to hello@codebaltimore.org!

I. Variables and Constants

Variables and constants are at the heart of all programming and are the first thing we'll learn in Swift.

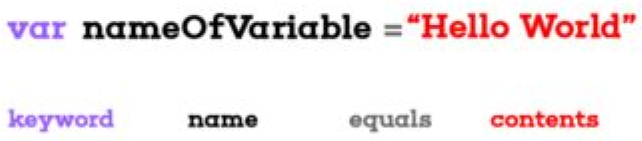
Swift is an 'Object Oriented Programming' language (OOP for short). Think of OOP as a coffee maker - we'll use this metaphor throughout the lessons. You put something or multiple things into the coffee maker to produce something else. In the typical use case, you would put in coffee grounds and water to the top of the maker and produce coffee. This is very similar to how all of object oriented programming works in general. You will take data and manipulate it (maybe combining certain things or otherwise altering it in some way) to produce a result. We'll get to this in much more detail later but let's focus on the first part of the coffee maker - the ingredients.



So what are variables and constants? In the simplest of forms they are boxes that hold your data so that you can use whatever is inside later in some form. In the case of the coffee maker, your variable would be the glass or pitcher the water is in and the jar you keep your coffee grounds in.

Declaration

Let's get to some coding. How do we declare a variable to be used in Swift? There are four parts to a declaration: the keyword, the name of the variable, the equal sign, and the contents.



```
var nameOfVariable = "Hello World"
```

Keyword
The keyword for variables is 'var' while the keyword for constants is 'let'. Variables and constants differ in their "mutability" - basically a fancy way of saying they can or cannot be changed. Do you want the box to be opened and new stuff replace the old stuff? If yes, you want a variable. If no, then you want a constant. To further illustrate this, take this example: you want to count how many times someone goes around on the merry-go-round.

```
var timesAroundMerryGoRound = 0
```

Every time that merry goes round, you want that number to change. Instead of creating a new variable for each time, you can just replace that number with the next.

```
timesAroundMerryGoRound = 1
```

Notice that I did not put the var keyword in the second declaration. When altering the contents of the variable, you don't want to create a new variable, you want to amend the old one. All you need to do is reference the name and set it equal to another value. If I call the variable (access what is inside of it, I can see that timesAroundMerryGoRound is equal to 1 now instead of what it initially was: 0.

```
timesAroundMerryGoRound //evaluates to 1
```

Name
Naming your variables may seem banal but it is quite important when building projects, especially large ones. Naming practices in Swift are generally the same as naming practices everywhere else in programming: be descriptive, be concise, and use Camel Case.
What is Camel Case? Since you are not allowed to put spaces in your words like typical english, we need some way of reading the variable name quickly without having to decipher words from one another. Camel case just means to capitalize every word except the first.

```
franklyMyDearIDontGiveADamn is Camel Case
franklymydearidontgiveadamn is not
```

You can see that Camel Case is MUCH easier to read than the latter.

You want to make sure your names are descriptive as well. While you can name your variables basically anything you want, remember you have to use them and remember what goes in them throughout your code. For example, if you wanted to put the first few digits of pi in a constant, a great variable name would be:

```
let firstFewDigitsOfPi = 3.1516
```

A bad variable name would be something that is too short to understand, too long, or nonsensical.

```
let a = 3.1516
let theseAreTheFirstFiveDigitsOfPiThatIWillUseToCalculateTheRadiusOfACircle =
3.1516
let antidisestablishmentarianism = 3.1516
```

Equals

In the Swift language, like many other programing languages, the equals sign, '=', is extremely important. The equals sign though DOES NOT LITERALLY MEAN equals. A better way of thinking about the equals sign is to think that something is being 'set' by the equals sign. When we write:

```
var one = 1
```

We are literally saying: create a variable named 'one' and "set" it to the value '1'. We then can set it to the value of 999 by saying:

```
one = 999
```

Here we are literally saying: take the variable 'one' and "set" it to the value of '999'.

Bonus: How do we actually check if something EQUALS something else?
You may be thinking, "Wait, so if '=' doesn't literally mean "equals" what does?". The answer is a double equals sign: '=='. This double equals sign checks for equality of two values. For example:

```
5 == 5 //is true!
6 == 5 //is false!
```

```
let theFirstNumber = 5
let theSecondNumber = 5
theFirstNumber == theSecondNumber //This is True!
```

Your turn! Try and complete these challenges. Answers are in the back.

1. Mike is turning 20 years old. Create a variable and set it equal to 20. Name it something descriptive about Mike and his age.
2. You have 12 eggs in a dozen. Create a constant, set it equal to the number of eggs, and name it something descriptive.

II. Basic Types

Computers read information in many types just like humans do. For example a 3 is a number, 'three' is a word, and 3.33 as a decimal point number. Computers have special types for each one of these as well. Let's take a look at a few...

Strings

Strings are characters that are "strung" together to make words and sentences. A string like "Hello" is actually 5 characters strung together "H", "e", "l", "l", "o". Let's set up a variable to hold our hello string. Strings are created with the double quotation marks.

```
var theStringHello: String = "Hello" //this is a proper string
var theStringHello: String = Hello //this is not proper and will result
in an error. Make sure you have the double quote marks!
```

You can see in the 'variable declaration' above that we added a colon and "typecasted" the variable. We're basically putting a label on the box that tells the computer "ONLY Strings can be in this box". We then set the variable equal to a string: "Hello". Swift has a nifty feature inside called 'type-inference' meaning that you don't HAVE to explicitly tell the computer what is going in the box - instead, it will INFER the contents from whatever you set it equal to. So to the computer, these are exactly the same thing:

```
let aStringVariable: String = "Hello"
let aStringVariable = "Hello"
```

Good practice though is to typecast as many variables as you can. It speeds up compile time and makes it explicitly clear to you, the programmer, what is supposed to go inside. Even though Swift has type inference, it doesn't mean, however, that you can change the type of the variable after it has been set. For example:

```
var aStringVariable = "Hello"
aStringVariable = 1 //Oh no an error! You can't do this!
```

You cannot set a variable of type String with another variable of some other type, like a number (an Int in this case).

Int

An Int stands for Integer. Just like in your basic math class, an Integer is any whole number. For example

```
var anExampleOfAnInt: Int = 0
var anotherExampleOfAnInt = 9
var lastExampleOfAnInt = 128723
```

Double, Float

Double's and Float's are decimal numbers with Double's being able to hold 'double' the amount of decimal points than floats allowing them to be much larger or more precise numbers.

```
var aDoubleExample: Double = 0.123456
var aFloatExample: Float = 0.123456
```

Even though the two numbers above are the SAME number, they are different types. If we tried to add them together we would have to convert one to another.

Booleans

A Boolean is just true or false. That's it! It's characterized by the 'Bool' keyword

```
var isTrue: Bool = true
var isFalse: Bool = false
```

Your turn!

1. Create a variable of type float with the value of pi to four decimal points. Strictly show the type in your declaration.
2. Create a Boolean variable called lightSwitchStatus and set it equal to false. In the next line, someone flipped the switch so set the lightSwitchStatus to true.
3. Create a constant Int with your age. Name the constant something descriptive.

III. Arrays and Dictionaries

So you now know how to instantiate a variable in Swift and put something, of some type, inside it. That's awesome for manipulating small amounts of data, but what about bigger data sets? What if, for example, you were trying to keep a list of students in a class. From what you know right now you could keep them all in their own separate variables like so:

```
var firstStudent = "Mary"
var secondStudent = "John"
var thirdStudent = "Israel"
var fourthStudent = "Pablo"
```

While that is alright for a few, what if you have hundreds or thousands? Are you going to be instantiating a new variable for each? Of course, not. That's why we have two collection types that store multiple values, Arrays and Dictionaries.

Array

An array is a collection of multiple values of the same type. Basically, it's a way to group data inside one variable. Let's see how to instantiate it and how to use it. Continuing on the class list example, I'm going to create a new array of strings called "classList" and fill it with the names of the students I had above.

```
var classList: [String] = ["Mary", "John", "Israel", "Pablo"]
```

Let's take a look at the structure above:

- ***Keyword:*** We've seen var before. The keyword instantiates the variable just like in a regular string variable. If we wanted to make it a constant (that is, if we know it will not be changed ever) we could have used let.
- ***variableName:*** Again, seen this before. Array names don't differ wildly from other variable names. The rules of concise, descriptive, etc. still apply - asDoesUsingCamelCase!
- ***Type Declaration:*** This is new. We've seen a regular variable declare its type like this var variable: String but now we have brackets around it. That signals to the computer that the variable is an array. You can alternatively use something like [Int] or [Bool] to declare an array of those types as well.
- ***Array Declaration:*** After the equals sign we see a whole bunch of new stuff. To add initial data to an array, you have to open a bracket to tell the computer the array is starting, fill it with the type you have previously declared, and separate each piece of data with a comma. Each piece of data is accessed with an "index value", basically just a number that points to a specific point in the array. Let's explore index values a bit...

Index Values

Array's have "sizes". If I have 4 names inside my classList array above that means I have 4 indexes, one to hold each value. Now the tricky part comes from the numbering system. While we, as humans, count from one (1, 2, 3, 4), computers count starting at 0 (0, 1, 2, 3). When we look at an array, we access the values starting from 0 instead of 1. Let's take our classList array... How would I access each name?

```
var classList: [String] = ["Mary", "John", "Israel", "Pablo"]
classList[0] // This equals "Mary"
classList[1] // This equals "John"
```

And so on...

Dictionary

Another common collection type is the Dictionary. Dictionaries are similar to Arrays in that they hold multiple values of corresponding data types but unlike Arrays, dictionaries have two parts to them: the Key and the Value. Let's see how a dictionary is set up.

```
let swiftDictionaryExample: [String: Int] = ["String Value": 1, "Another
String Value": 198]
```

Let's take a look at the new stuff here.

- ***Type Declaration:*** In a dictionary, we open a bracket '[]' and add two types separated by a colon. Just like a regular english dictionary where we have a word then definition, in Swift dictionaries, we have a KEY then VALUE. The first type, in this case 'String', before the colon is the KEY while the second type, in this case Int, is the VALUE. What we are setting up here is a dictionary that accepts String type keys and Int type values. You can fill this dictionary with any String, Int pair you would need, however, just like Array's, you cannot fill them with conflicting types. For example, I could NOT fill the example above with a value like [0.11: false] since that would be a [Float: Bool] not a [String: Int].
- ***Data:*** Setting up the data in a dictionary is very similar to arrays as each index is separated by a comma. In the example above, there are TWO index values, "String Value": 1 and "Another String Value": 198. Make sure when you are filling dictionaries with data that you create key value pairs separated by commas like so [key:value, key:value, key:value]

Accessing Dictionary Values

Accessing dictionary values is different from what we have seen before. Just like looking up a specific word in an actual dictionary, we can look up a specific value by a key. Let's take for example a dictionary of names and corresponding ages.

```
var classNameAgeDictionary: [String: Int] = [
"Jon": 45,
"Ian": 24,
"Sumner": 87,
"Jim": 36,
"Anita": 45
]
```

Note: You can see above, you can set up your dictionary with a more human readable format and it will still be compiled by the computer. This formatting has no effect on the code but make sure you put commas after each key:value pair and close your bracket after!

Let's say we wanted to know Ian's age above. How would we do that. Easy.

```
classNameAgeDictionary["Ian"] //This will equal 24!
```

What if though, we got it wrong and Ian is actually 27? All we have to do is set that value equal to another and it will update our dictionary.

```
classNameAgeDictionary["Ian"] = 27 //Ian's age is now 27!
```

Your turn!

1. Instantiate a new mutable (var keyword) array of type String to hold the names of the colors of the US flag. You're going to have 3 index values in this string array, one for each color. Name the array something descriptive and fill it with the names of the colors of the US flag.
2. Instantiate a new array of Int's and name it whatever you want. Fill it with the numbers 1 to 5. Access the 4th index. Remember, computers start counting from 0!
3. There was just a big horse race at the racetrack. Create a dictionary of racehorse names and finishing places (Hint: the type should be [String:Int]). In the next two lines, switch places between two of the horses (ex. 1 to 4 and 4 to 1)

IV. Conditionals

Many many many times in software development you're going to want to check if something is true or false. Maybe it's to check if a user has filled in all the required information, maybe you want to run some code only for people over the age of 25. If you want to check for something, you'll need to use a 'conditional' or an if/else statement.

```
var superhero = "Batman"

if superhero == "Batman" {
 print("I am the Batman!")
} else {
 print("I am not the Batman")
}
```

In the above example, we set up a variable called superhero and set it equal to "Batman". We then ask the computer, “is the superhero variable equal to ‘Batman’”? If yes (true), then we execute the code print("I am the Batman!"). If no (false), then we execute the code print("I am not the Batman!"). We have a few new things here so let's go through them.

Quick note: This is the first time we see the print() function. Print is a very important function in debugging as it will “print” objects you put into it onto the console (the white box at the bottom of the screen) so you can see what they evaluate to. In the example above, because the superhero was set to Batman, the first print statement will run, printing to the console, “I am Batman”. The second print statement will never run, unless you change the superhero to something else.

Operators
Operators are the signs we use to evaluate something. They are as follows:

```
< Less than
<= Less than or equal
> Greater than
>= Greater than or equal
== Equal
!= Not equal
|| or
&& and
```

The double equals sign for example, ==, asks "Is equal to?" while the exclamation point equal sign, != asks "Is not equal to?". In our Batman example above, because the superhero variable was "Batman", the if statement (if superhero == "Batman") evaluated to true, meaning that the code after the brace ({) was run and the else code was not run. Let's take another example.

```
let billsAge = 20
let larrysAge = 30

if billsAge > larrysAge {
    print("Bill is older than larry")
} else if billsAge == larrysAge {
    print("Bill is older than larry")
} else {
    print("Bill is younger than larry")
}
```

In this example we introduce the else if clause. Else if allows us to evaluate multiple conditions at once. We can check if Bill is older than Larry, if Bill is the same age, AND, if both those are untrue, be able to say "Oh ok, Bill is younger". Since Bill is 20 and Larry is 30, the computer will look at the those variables and say "is Bill's age greater than (>) Larry's?...no...ok so is Bill's age equal to Larry's?...no..ok so that means I have to run the else {} block!
We've also introduced in this section the print() statement. Print is really important not only while learning but while debugging an app. When running your code in Xcode, you'll use the print statement often to see what values evaluate to.

Your turn!
1. Jane and John both took a test. Jane scored a 92 on the test while John scored an 89. Create a conditional that takes both Jane's score and John's score and compares them then print out a sentence to show who scored higher.
2. You're making an app that looks for plagiarism in two sentences. Write a conditional that searches for whether or not two sentences are not equal to each other.

V. For Loop

Looping code multiple times is often an important part of software development and Swift helps do this seamlessly. There are multiple different types of loops but for now, let's just look at the most common one: the For Loop.

For Loops

A For loop is quite possibly the most common loop. In a for loop, all you need to do is specify a value and a range and the loop will execute that many times. For example, let's say we wanted to count from one to ten. We could say,

```
print(1)
print(2)
print(3)
//...and so on
```

but that is inefficient. How can we repeat that same code: print(aNumber) 10 times? With a for loop!

```
for someNumber in 1...10 {
  print(someNumber)
}
```

If you've programmed before, for loops aren't new but this syntax might be. For loops in swift are simplified, using the power of type inference to set up the temporary variable for you. This can get confusing so let's go through it piece by piece.

- ***for:*** this is the keyword to get things started just like if or var in their respective issuances.
- ***someNumber:*** this is a temporary variable that holds whatever the loop is currently evaluating. In the example above, the temporary variable will be the numbers from 1 to 10 because we explicitly stated the range (1...10). The first time the loop runs it prints someNumber which, because it's running for the first time in 1...10, will print 1. The second time, 2, third time, 3, and so on. Don't get this confused, however, with just a plain and simple number. This is the VALUE of the range. Let's take a look at another example to further explain this.

```
let bowlOfFruit: [String] = ["Apple", "Banana", "Pear", "Orange",
"Apricot"]
for someFruit in bowlOfFruit {
    print(someFruit)
}
```

This is a bit different. Instead of setting up an explicit range, we instead set up a loop OVER an array. This means that the VALUE temporary variable someFruit will be equal to the corresponding VALUE in the array. The first time the loop executes, it will print Apple, next time 'Banana', and so on instead of just the index number (1, 2, 3...)

- ***in:*** another keyword to break up the VALUE and RANGE that you fill in.
- ***range:*** As we just talked about the range can be over an array or an actual number range. Number ranges are set up with your first value, three dots (an ellipsis), then your last value. If you wanted to repeat things 20 times for instance you can write 1...20. If you wanted to repeat 100 times 1...100. This can also be the iteration over an array like shown above.

Your turn!

1. You want to count to 100. Write a for loop that will print the numbers 1 to 100.
2. You're building a shopping list. Write a for loop that prints out the items of the shopping list.

VI. Functions

We've arrived. The basis of all object oriented programming: functions.

Functions do stuff. Everything you do in iOS programming is within some function. If you think way back to the coffee maker example, we had variables (water, coffee grounds) that we entered into the coffee maker to produce a result. In Swift functions we have variables (called parameters, this can be any type - String, Int, Double, Bool - whatever) that we enter into a function (the coffee maker) to produce a result (sometimes it can return a value or just mutate a previous value). Let's take a look at a very simple function to add two numbers together.

```
func addTwoNumbers(numberOne: Int, numberTwo: Int) -> Int {
    return numberOne + numberTwo
}

addTwoNumbers(numberOne: 10, numberTwo: 5) //evaluates to 15
```

Let's take a look at the syntax of a function. This function HAS a return statement. Functions do not necessarily need to, but if you want the function to evaluate to something, you can add a return statement.

- ***func:*** The keyword! This starts the function off and is required when building a function.
- ***addTwoNumbers:*** The name of the function. Again, follow classic naming conventions. Use camel case, be specific but concise.
- ***(numberOne: Int, numberTwo: Int):*** The parameters are variables you enter into the function to execute that code. Parameters have a name that you create and need to be assigned a type. In this case numberOne is of type Int. Functions are not required to have parameters and in that case, the parameters would just be empty like so: ()
- ***->:*** this arrow is a keyword as well. Just like func or if, the arrow signifies that the function returns something, that is it produces something you can do something with. In the coffee maker example, it would return coffee.
- ***Int:*** This is the return TYPE. Just like we put a type on our parameters, the value or values being returned must also have a type. This can be anything that you want to return, an Int, String, Array, whatever you do in the function and want to return.

We then can call the function by name and add our parameters:

```
addTwoNumbers(numberOne: 10, numberTwo: 5)
```

We can call this as many times with as many different numbers as we want

```
addTwoNumbers(numberOne: 20, numberTwo: 1) //21
addTwoNumbers(numberOne: 6, numberTwo: 2) //8
addTwoNumbers(numberOne: 12, numberTwo: 4) //16
```

For reference:

Let's put it all together is one big function.

```swift
func birthdayCakeValidator(arrayOfIngredients: [String]) -> String {
    var neededIngredients: [String] = []

    for i in arrayOfIngredients {
        if i == "sugar" || i == "flour" || i == "vanilla" {
            neededIngredients.append(i)
        }
    }

    if neededIngredients.contains("sugar") &&
neededIngredients.contains("flour") && neededIngredients.contains("vanilla") {
        return "Yes, we can make a birthday cake!"
    } else {
        return "No, we cannot make a birthday cake!"
    }

}

var ingredientListOne: [String] = ["fish", "flour", "water", "juice", "meat"]
var ingredientListTwo: [String] = ["flour", "sugar", "vanilla", "frosting",
"french fries"]

birthdayCakeValidator(arrayOfIngredients: ingredientListOne) // "No, we cannot
make a bithday cake!"
birthdayCakeValidator(arrayOfIngredients: ingredientListTwo) // "Yes, we can
make a birthday cake!"
```

Let's walk through this a bit. First we set up a new function called birthdayCakeValidator and we accept one parameter: an array of strings. Next, we set up our function to return a string. After the braces we set up an empty array that we will fill with the needed ingredients. To do so, we set up a for loop to loop through the parameter array and extract only the needed ingredients checking if the index value is sugar OR flour OR vanilla. After the for loop completes, we check if the needed ingredients array contains certain items - sugar AND flour AND vanilla. If they do, we return Yes! If not, we return a No string.

To test this function, we create two arrays of ingredients to enter as a parameter in the function. We then enter them and run the function!

Your turn!

1. Write a function that takes a name String as a parameter and prints it. There is no return statement in this function.
2. You want to know the largest number in a set of numbers. Write a function that accepts an array of Int’s as a parameter and returns the largest Int.
3. Write a function that accepts two numbers. If the first number is larger than the second number, subtract the two numbers. If not, add them. Return the result.

Appendix A: Answer Key

Section I: Variables

1. Mike is turning 20 years old. Create a variable and set it equal to 20. Name it something descriptive about Mike and his age.

```
var mikesAge = 20
```

2. You have 12 eggs in a dozen. Create a constant, set it equal to the number of eggs, and name it something descriptive.

```
let numberOfEggsInADozen = 12
```

Section II: Basic Types

1. Create a variable of type float with the value of pi to four decimal points. Strictly show the type in your declaration.

```
var pi: Float = 3.1415
```

2. Create a Boolean variable called lightSwitchStatus and set it equal to false. In the next line, someone flipped the switch so set the lightSwitchStatus to true.

```
var lightSwitchStatus: Bool = false
lightSwitchStatus = true
```

3. Create a constant Int with your age. Name the constant something descriptive.

```
let iansAge: Int = 24
```

Section III: Arrays and Dictionaries

1. Instantiate a new mutable (var keyword) array of type String to hold the names of the colors of the US flag. You're going to have 3 index values in this string array, one for each color. Name the array something descriptive and fill it with the names of the colors of the US flag.

```
var colorsOfTheUSFlag: [String] = ["Red", "White", "Blue"]
```

2. Instantiate a new array of Int's and name it whatever you want. Fill it with the numbers 1 to 5. Access the 4th index. Remember, computers start counting from 0!

```
var oneThroughFive: [Int] = [1,2,3,4,5]
oneThroughFive[3]
```

3. There was just a big horse race at the racetrack. Create a dictionary of racehorse names and finishing places (Hint: the type should be [String:Int]). In the next two lines, switch places between two of the horses (ex. 1 to 4 and 4 to 1)

```
var horsesInTheRace: [String: Int] = [
"Honey": 1,
"Larry": 2,
"NeighNeigh": 4,
"Carlos": 3
]

horsesInTheRace["Honey"] = 4
horsesInTheRace["NeighNeigh"] = 1
```

Section IV: Conditionals

1. Jane and John both took a test. Jane scored a 92 on the test while John scored an 89. Create a conditional that takes both Jane's score and John's score and compares them then print out a sentence to show who scored higher.

```
let janesScore = 92
let johnsScore = 89

if janesScore > johnsScore {
    print("Jane scored higher.")
} else if janesScore == johnsScore {
    print("They both scored the same.")
} else {
    print("John scored higher.")
}
```

2. You’re making an app that looks for plagiarism in two sentences. Write a conditional that searches for whether or not two sentences are not equal to each other.

```
var firstSentence = "This is a sentence."
var secondSentence = "This is another sentence."

if firstSentence == secondSentence {
    print("The senetences are the same")
} else {
    print("The sentences are different")
}
```

Section V: For Loops

1. You want to count to 100. Write a for loop that will print the numbers 1 to 100.

```
for number in 1...100 {
    print(number)
}
```

2. You’re building a shopping list. Write a for loop that prints out the items of the shopping list.

```
let shoppingList = ["Eggs", "Flour", "Meat", "Cheese"]

for item in shoppingList {
    print(item)
}
```

Section VI: Functions

1. Write a function that takes a name String as a parameter and prints it. There is no return statement in this function.

```
func printName(name: String) {
```

```swift
    print(name)
}

printName(name: "Ian")
```

2. You want to know the largest number in a set of numbers. Write a function that accepts an array of Int’s as a parameter and returns the largest Int.

```swift
func returnTheLargestNumberOfAnArray(setOfNumbers: [Int]) -> Int {
    var largestNumber: Int = 0
    for i in setOfNumbers {
        if i > largestNumber {
            largestNumber = i
        }
    }

    return largestNumber
}

returnTheLargestNumberOfAnArray(setOfNumbers: [5,6,7,8,9,10])
returnTheLargestNumberOfAnArray(setOfNumbers: [1,10,100,1005])
```

3. Write a function that accepts two numbers. If the first number is larger than the second number, subtract the second from the first. If not, add them. Return the result.

```swift
func compareTwoNumbers(firstNumber: Int, secondNumber: Int) -> Int {
    if firstNumber > secondNumber {
        return firstNumber - secondNumber
    } else {
        return firstNumber + secondNumber
    }
}
```