Arrays

Many Variables / Constants

Often we need to have a lot of variables / constants for the same task. Creating all of these variables / constants can clutter up your code and make it very hard to program. Also, the number of variables / constants that you created for that program is a static amount, meaning it's not going to change when the program runs. That's problematic when it comes to creating good adaptable and reusable code! All of that is why Swift has **arrays**.

Arrays are an ordered collection of values that can be accessed through one variable / constant and can have any number of values in them. For example,

```
let old_robot_programmers: [String] = [“Lucas”, “Wesley”, “Finn”, “Kelly”,
    “Kyle”]
```

> **Constant Arrays**
> Just like single constants, constant arrays *cannot* have items added to them or have their items' values change.

Array Types

For *any* type, you can make an array of that type. Just like variables and constants, arrays have **static types**, meaning that Swift either guesses the array's type or you write it explicitly and that the array's type can never change after it is initialized. It also means that you cannot add a String to an array of Bools.

Creating Arrays

That's all great stuff, but how do you actually create arrays? What if you have a list of values you already know?

```
let x = true
let y = true
let array0 = [x, y, true, false]
// array0 is an array of Bools
```

What if you don't know how many or what values you'll have, but you know you need to store them all in the same place?

```
var array1: [Int] = []
var array2 = [Int]()
// These two statements both create empty Int arrays.
// The type annotation is necessary for an empty array
```

What if you need to have a whole array of repeating values?

```
let array3 = [Double](repeating: 0, count: 100)
let array4 = Array(repeating: 0.0, count: 100)
// These two statements both create Double arrays with 100 0's in
     // them. The decimal point in the second statement tells Swift
     // that it is a Double
```

What if you want an array that is made up of two different arrays?

```
let array5 = array3 + array4
// array5 is inferred as [Double] and has 200 0.0 elements
// When arrays (or Strings) are involved, + is not commutative
```

Accessing Arrays

To access one element in an array, you would use an index, which is *always* an Int:

```
let some_primes = [2, 3, 5, 7, 11, 13, 17, 19]
print(some_primes[0])
// prints 2
print(some_primes[4])
// prints 11
print(some_primes[7])
// prints 19
```

Notice how instead of using `some_primes[1]` to access 2, we use `some_primes[0]`. That's because Swift is **0-indexed**. This means that the *initial* element in a sequence, like an array, is given the index 0. The next index would be 1, then 2, then 3, and so on until the last element.

How Long is an Array?

Knowing how long an array is extremely helpful. Luckily, Swift has the ability to know how long an array is built right into those types. All you have to type is `some_array.count` and that gives the value of how many elements are in `some_array`. For example,

```
print("# of elements in some_primes: \(some_primes.count)")
// Output: # of elements in some_primes: 8
print("Last element: \(some_primes[some_primes.count - 1])")
// Output: Last element: 19
```

Notice how, to access the last element in `some_primes`, we had to write `some_primes[some_primes.count - 1]`. That's because, like we said before, arrays in Swift are 0-indexed. When we normally count things in real life, we use **1-indexed counting**, so when Swift *counts* the elements in an array, it uses 1-indexed counting. That causes an off-by-one error if we tried to use `some_primes.count` as index. So to convert the 1-indexed count to 0-indexed, we simply subtract 1.

Modifying Arrays

As said before, to modify an array at all, you must declare it as a variable when you create the array. Just like any way of storing data, you cannot change an array from a constant to a variable array after declaring it as a constant array.

Say you have the array

```
var shopping_list = ["Eggs", "Milk", "Flour", "Sugar", "Butter", "Vanilla"]
                  // looks like someone is making crepes!
```

What if you had a guest coming over that was vegan? Well, now you have to change some items, or elements, in your shopping list. How would you do that?

```
shopping_list[0] = "Maple Syrup"
shopping_list[1] = "Soy Milk"
shopping_list[4] = "Soy Margarine"
```

Great! No more eggs, milk, and butter! Just sweet vegan goodness.

But what if Wesley was coming over? You need cinnamon or he is going to diss your crepes! Also, strawberries make the perfect dessert crepe even more perfect. How do you add cinnamon and strawberries to your shopping list?

```
shopping_list.append("Cinnamon")
shopping_list.append("Strawberries")
```

That's a lot of typing though, especially when you could be writing

```
shopping_list += ["Cinnamon", "Strawberries"]
```

Those two code blocks do the same thing! They add `"Cinnamon"` and `"Strawberries"` as separate elements to the end of the `shopping_list` array.

Note that you cannot use an index to create a *new* element. You can only use an index to access and change an element. Swift will give an error if you try to access any element outside of the array's indices. Just remember, an array's indices go from 0 to the array's count - 1.

Iterating Over Arrays

What if your guests didn't really like strawberries? Well, you don't really want to remove strawberries from your list because you personally like them a lot! But you do want, perhaps, blueberries to come before strawberries on your list as they will be preferred by more people. So how do you insert an element into your array?

First, you have to know where the "Strawberries" element is. To find that out, we can use **iteration**. Iterating over arrays means to consecutively access each element's value in an array. Basically, you are looking at each value in an array one after the other (in order!). Iteration is *super* common so pay attention, because you will be doing this *a lot* with arrays.

To iterate, you need two basic things: a loop to repeat code for every valid index and a increasing index so you don't get stuck in the loop. For example,

```
var index = 0
while index < shopping_list.count {
      // Your code you want to do for every index
      index += 1
}
```

Some notes about the variable index:

- It has to be an Int because it is a counter (it does not make sense to have 0.5 of a loop) *and* will be used to access elements from the array.
- It can never equal the array's count if you are going to use index to access elements from the array. Reread the past pages for explanation.
  - It also does not make sense for index to equal the array's count if you start at 0 because then you will have one more loop than elements in the array.
- Many programmers use the name i to stand for index. We use index here for clarity.
- Do not forget to have some way for the loop to end or else you will be stuck in a forever looping while loop.
  - To make sure the above loop ends, we have a condition that can be satisfied if index is large enough and we make sure to increase index every loop so that the condition will be satisfied eventually.

Now that we know how to iterate through our array, how do we insert an element at a specific index in the array? With `array.insert(element_value, at: index)`! How do we use that in our code? Well, we don't care where the blueberries go on our list as long as they come before the strawberries so let's put them at the index the strawberries are currently at. `insert()` will move all elements at and after the index you enter into `insert()`. So our code would like this:

```
var i = 0
while i < shopping_list.count {
      if shopping_list[i] == "Strawberries" {
            shopping_list.insert("Blueberries", at: i)
            break
      }
      i += 1
}
```

Now what does our final shopping list look like? Simply `print()` to find out.

```
print(shopping_list)
// Output: ["Maple Syrup", "Soy Milk", "Flour",
      // "Sugar", "Soy Margarine", "Vanilla",
      // "Cinnamon", "Blueberries", "Strawberries"]
```

And there you have it! The perfect ingredients for perfect guest-friendly crepes.

Take a Break!

It's always important to take breaks once in awhile, and that's true for while loops too.

The whole purpose of doing the iteration of the array was to simply find the index of "Strawberries", so why should we keep looping when our task is finished?

What break does is literally what it says. It breaks Swift out of the loop so it can continue with the rest of the code. No more looping through unnecessary code!

Conceptual challenge: what would happen if there was no break statement there?