**Loops**

Computers are really good at two things: **storing information**, and **doing something over and over again**. If we want to execute a set of code repeatedly for a given number of iterations or until it reaches a specific condition, we will use loops. There are two types of loops: the for loop and the while loop. We **use the for loop when we know beforehand how many times we are going to have to repeat the code** while we **use the while loop when we don't know how many times we have to run the code, but we know we have to run the code until it reaches a particular condition.** However, at their core, they are both the same, we can do whatever you can do with a for loop instead of a while loop and vice versa.

**For-in Loops**

A **for-in loop performs a set of code in a specific range, sequence, or collection**. For now, we will focus on using the for-in loop with a range. Go ahead and run the following code below, in the playground, and see what the output is from the Assistant Editor.

// loop from 1 to 5 including 5

for i in 1...5 {

print(i)

}

// loop from 1 to 5 excluding 5

for i in 1..<5 {

print(i)

}

You can just as easily use variables to create a range as well!

var start = 0

var end = 5

// loop from start to end including end

for i in start...end {

print(i)

}

// loop from start to end excluding end

for i in start..<end {

print(i)

}

Note that when writing for loops in Swift, you will always use the "for-in" style with a range. The C-style for loops with the separate initialization, condition, and increment clauses was deprecated in Swift 2.x and taken out altogether in Swift 3.

**While Loops**

A **while loop is used to loop for an unknown number of times.** Unlike a For-in loop where you first specify a range to loop through, in a while loop, we simply specify a condition. Anything that we write with a for-in loop can be written with a while loop. Let's see how we can convert our for-in loops to while loops.

We can re-write this ***for-in*** loop...

for i in 1...5 {

print(i)

}

... as a ***while*** loop.

var i = 1

while i < 6 {

print(i)

i = i+1

}

copy

Note that the while loop syntax requires us to create our own bounds.

Documentation to for each

<https://developer.apple.com/documentation/swift/array/1689783-foreach>

# Arrays

We often have to combine related values into a collection. Swift provides us with two Collection types: Array and Dictionary. First, we will go over the Array.  **Arrays are an ordered collection of values**. On the other hand, **Dictionaries are an unordered collection of key-value pairs**. Let's first go over Arrays. Each position is defined with an index starting at 0. It's a lot easier to explain with examples.

We will declare a variable  toDoList to be of an **instance of the Array Type** which will hold onto an **ordered collection of instances of the String Type**. The [] brackets denote the Array type and inside of the brackets we place the type of the values that we'll be storing.

var toDoList: [String] = ["Learn iOS", "Build the next Flappy Bird", "Retire in Cancun"]

Here we are declaring the toDoList and giving it some values to begin with. What if we wanted to start with no tasks and then gradually add them on? We would have to initialize an empty array like so:

var toDoList: [String] = [String]() // Setting the array type and initializing the array

//alternative syntax that you may see:

var alternativeToDoList = Array<String>()

toDoList.append("Learn iOS")

toDoList.append("Build the next Flappy Bird")

toDoList.append("Retire in Cancun")

print(toDoList)

Every Instance of Array has the append() method that allows us to add elements to the Array collection.

Now if we are initializing an instance of an  Array Type that will hold onto instances of the String Type, we can get rid of the type annotation because Swift will be able to infer its type with [String]() just like value types.

var toDoList = [String]()

toDoList.append("Learn iOS")

toDoList.append("Build the next Flappy Bird")

toDoList.append("Retire in Cancun")

print(toDoList)

Just like how we could not assign an  Int to a variable or a constant that was declared to hold a different Type, if we try to append an instance of a Type that is not a String, Swift will throw us an error.



Accessing Arrays

Arrays are zero indexed. What this means is the first instance inside of an  Array lives at index 0. The second instance inside of an Array lives at index 1.

var arrayOfInts = [1, 2, 3, 4, 5] // Note that we let Swift infer the type here

// The first number lives at index 0.

print("\(arrayOfInts[0])")

// The second number lives at index 1.

print("\(arrayOfInts[1])")

// The third number lives at index 2.

print("\(arrayOfInts[2])")

// The fourth number lives at index 3.

print("\(arrayOfInts[3])")

// The fifth number lives at index 4.

print("\(arrayOfInts[4])")

We used ranges in our  for-in loops. Ranges can be used to index instances of the Array Type as well.

var arrayOfInts = [1, 2, 3, 4, 5]

// => "[1, 2]"

print("\(arrayOfInts[0...1])")

// => "[2, 3, 4]"

print("\(arrayOfInts[1..<4])")

// => "[3, 4]"

print("\(arrayOfInts[2...3])")

After we access an Array using its index, we can set the value at that particular index as well.

var arr = [1, 2, 3, 4]

arr[0] = 8

print(arr) // The array has now changed!

## Methods and Properties

Arrays have several methods and properties -- these are just a few that will help you use and modify your collections.

### append()

We already saw how we could append instances to an  Array. We use the append() method to add a given element to the array. This method is very similar to push() methods in other languages:

var nums = [1, 2, 3, 4]

nums.append(5)

print(nums)

### remove()

We can run the method remove() on an Array and provide the index of the element we want to delete. This method removes and returns that particular element at the provided index.

var arrayOfInts = [1, 2, 3, 4, 5]

let popped = arrayOfInts.remove(at: 0)

print(popped)

print(arrayOfInts)

### removeLast()

**If you're missing the ease of popping the last element from the array, never fear! Swift uses removeLast() to remove and return the final element in an array.**

**var arrayOfInts = [1,2,3,4,5]**

**let popped = arrayOfInts.removeLast()**

**print(popped)**

**print(arrayOfInts)**

### insert()

We run the insert() method with two arguments. The first argument is the value that we would like to add to the array, and the second argument is the index where we would like to insert it. When we pass in the second argument, we have to give it a name at. We will be going over these named parameters more in the Functions tab.

var arrayOfInts = [1, 2, 3, 4, 5]

arrayOfInts.insert(6, at: 5)

### count

The  count is a property of the array types. It holds the total number of elements in an array. This property is very useful when we are looping through an array. We can write the previous code using the count method.

var arrayOfInts = [1, 2, 3, 4, 5]

arrayOfInts.insert(6, at: arrayOfInts.count)

## Looping Through Arrays

The for-in loop makes looping through all instances within an array a breeze.

var starters = ["Fisher", "Kobe", "Gasol", "Bynum", "World Peace"]

for starter in starters {

print(starter)

}

Or we can use a range as well along with the count property:

for i in 0..<starters.count {

print(starters[i])

}

# Dictionaries

An  Array Type is useful for a collection of ordered instances, but a **Dictionary Type is more suitable for unordered instances**. When we want to look for a definition, or value of a particular word in a dictionary, we look up the word in the dictionary and read its definition. A **Dictionary Type is a collection type that organizes its content by key-value pairs**. A *key* maps to a value just like a word in a dictionary maps to a definition.

## Initializing a Dictionary

// Here we are declaring myDict variable to be of Dictionary Type that will use instances of String as

// its keys and instances of Int as its values

var myDict2: [String: Int]

Once again, if we just declare a variable or a constant to be of the  Dictionary Type, **we will get an error when we try to do something with it because we have not initialized it yet.**

var myDict2: [String: Int] = [String: Int]()

Of course, we can use Swift's type inference:

var myDict2 = [String: Int]()

We can declare, initialize and set the values of the Dictionary in one line:

// here we are declaring myDict variable to be of Dictionary Type that will use instances of String as

// its keys and instances of Int as its values

var dictionary = [

"Kobe": 24,

"Lebron": 23,

"Rondo": 9

]

// Note how we are leveraging Swift's type inference here

## Accessing a Dictionary

We can access  Dictionaries just like how we accessed Arrays. **We just have to provide the key in-between brackets**. However unlike Arrays where the keys are instances of Ints and are zero-index based, Dictionary**keys can really be anything** and are not particularly ordered. We can access Kobe's number this way from the previous dictionary that we created:

dictionary["Kobe"]

On the Playground's right pane, it will say that the value is 24 but don't be fooled.  **When we access a Dictionary we get back an Optional Type**. This makes sense because there is a chance that a key doesn't exist in the dictionary. For example, the word 'gullible' might not exist in the Dictionary, therefore, it makes sense that when you access a Dictionary, **Swift will warn us that it might be nil by returning an Optional**. We can see this safety net in action when we try to store the result of the access to a separate variable:

var jerseyNumber = dictionary["Kobe"]

print(jerseyNumber) // Optional is not unwrapped

We can be very bold and just unwrap it:

var jerseyNumber = dictionary["Kobe"]!

print(jerseyNumber)

But it is better practice to use  **Optional Binding**:

if let jerseyNumber = dictionary["Kobe"] {

print(jerseyNumber)

}

## Modifying a Dictionary

We can modify a Dictionary in a similar way we did with Arrays.

var dictionary = [

"Kobe": 8,

"Lebron": 23,

"Rondo": 9

]

print(dictionary)

dictionary["Fisher"] = 2

print(dictionary)

dictionary["Kobe"] = 24

print(dictionary)

If the  key exists, we update the value of what is on the right-hand side of the equals sign. If the key doesn't exist, we add a new key to the dictionary along with the value we provided on the right of the equals sign.

## Removing

We can remove a  key-value pair by accessing the key and giving it the value of nil:

dictionary["Lebron"] = nil

Or we can send the message  removeValue.

var lebronsNumber = dictionary.removeValue(forKey: "Lebron")

print(lebronsNumber)

It is important to note that  **this message returns an Optional Type that might contain the value of the key-value pair that was deleted**. Once again it makes sense that this method returns an Optional Type because the key specified by the user might not be present in the dictionary. We can unwrap it with Optional Binding:

if let lebronsNumber = dictionary.removeValue(forKey: "Lebron") {

print(lebronsNumber)

}

## Looping

We can use the  for-in loop to loop through the keys and the values of Dictionaries.

for (key, value) in dictionary {

print("The key is \(key) and the value is \(value)")

}

The ( key, value) is of the Tuple type. Read more about **Tuples** in the Tuples section in  [The Basics](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID329).

for x in dictionary {

print(x)

}

If we don't provide the ( key, value) then Swift will print out a tuple for each key-value pair. Tuples ARE (key, value) pairs.