Swift
Accelerator
United Arab Emirates

Swift Language, Part 2

Unit 4: Arrays, Structs, and Closures

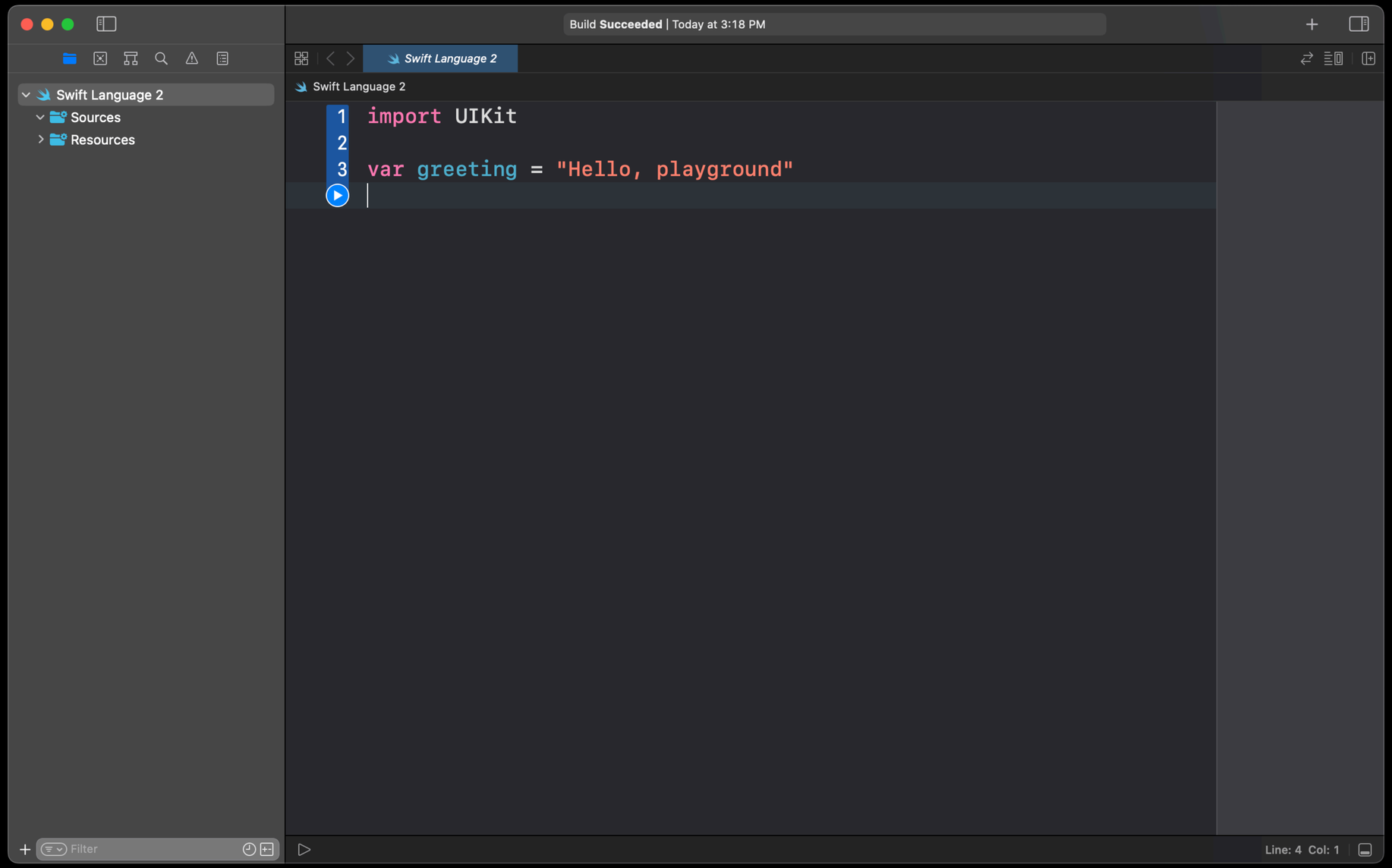
Unit overview

Back to Swift!

- Arrays
  - Loops, a brief introduction
- Structs
  - Classes (optional)
- Closures
- Collections and Collection Functions
  - Doing things imperatively vs. doing things declaratively

Getting started

- We're going back to Playgrounds — this time, we'll try **Xcode Playgrounds**, rather than Swift Playgrounds.
  - If you have an iPad, feel free to use Swift Playgrounds!
- To get started, start Xcode, then choose File → New → Playground, or (deep breath) Opt-Shift-Cmd-N, ⌥ ⇧ ⌘ N.
- Create a **Blank** playground, and save it anywhere. If you'd like, call it **Swift Language 2**.

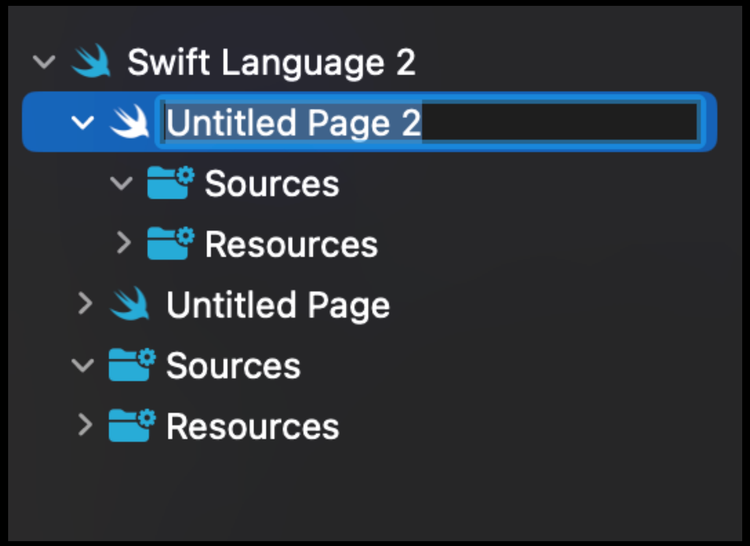

Build Succeeded | Today at 3:18 PM
Swift Language 2
Swift Language 2
Sources
Resources
Swift Language 2
1 import UIKit
2
3 var greeting = "Hello, playground"
Filter
Line: 4 Col: 1

Xcode Playgrounds

- The interface should be familiar!
  - Sidebar to show different Playground pages
  - Console at the bottom
  - Right section shows a Preview, like the 123 icon in Playgrounds would show you the value of any variables or calculations on that line
- In the middle code section, press the run ▶ button in the sidebar to *run all highlighted code up to that line*.

New Playground Page

- To make a new page, *make sure you have the sidebar open and the Playground title selected*, then use the menu: File → New → Playground Page, or use the shortcut **Opt-Cmd-N**, ⌥ ⌘ N.
- This will make an **Untitled Page 2**, which you can rename immediately, and make your previous page **Untitled Page** 🤔
  - Feel free to rename the previous one by right-/ctrl-/two-finger-clicking and selecting rename.



Arrays

Unit 4.1: Storing multiple values in a single variable or constant

Arrays are ordered lists of data

- Imagine an announcement in school, asking for your class to report somewhere (maybe a dental appointment, fun!)
  - The announcer won't call you out by name, one by one. That would take forever!
  - They'll ask for your class name, e.g. Secondary 2C, since that *one name refers to all of you*.
  - If they want just the first few members of the class, that works, too — you probably have an *index number* in your class list, so they can call for "Secondary 2C, index numbers 1 to 10".
- That's a single variable (or constant) that has all of your names in an ordered list. An array!

Without Arrays

This is one way to store data. What could go wrong here?
(Don’t type this code out — it’s just an example!)

```
var student1 = "Alice"
var student2 = "Bob"
var student3 = "Charles"
var student4 = "Eunice"
// oh no! we forgot one
var student5 = "Deborah"
// wait, they're not in order, we need to swap s4 and s5
var tempstudent = student5
student5 = student4
student4 = tempstudent
print(student4)
print(student5)

// Wait. . . Alice got food poisoning and dropped out
// Now we need to move everyone again
student1 = "I give up"
```

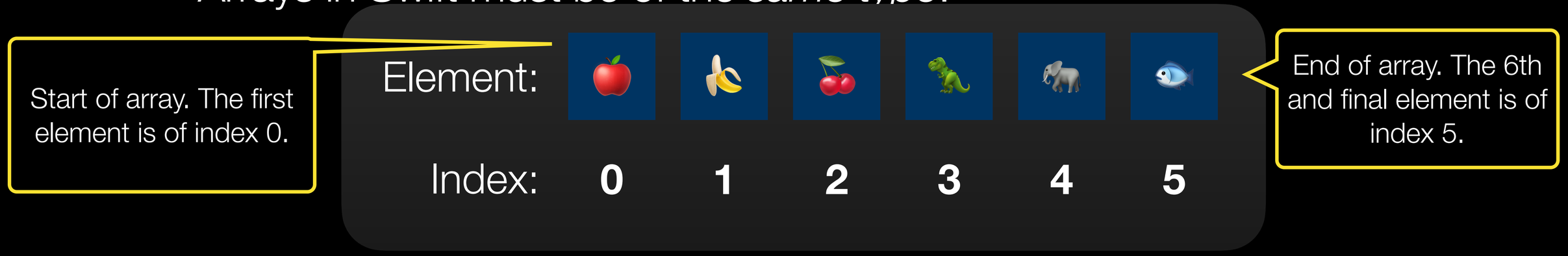
With Arrays

How about with arrays? Using arrays, we simply do the following: (Type *this* code out — feel free to skip the comments.)

```
var students = ["Alice", "Bob", "Charles", "Deborah"]
// Students are numbered from 0. So Alice is 0, Bob is 1, Charles 2, Deborah 3.

students.append("Eunice") // Add a student at the end.
print(students) // Print all the students to see what’s inside.

print(students[0]) // Who 's the first student?

students.remove(at: 0) // Remove a student! Everyone’s index updates automatically.
print(students)

students [0] = "Blob" // Oh we spelled Bob's name wrong
print(students)
```

Arrays

- Store multiple items, in an ordered list, into a single variable.
- This array, for instance, is of `Strings`:
  `var array = ["🍎", "🍌", "🍒", "🦖", "🐘", "🐟"]`
  - Arrays start counting from 0. This is the **index**.
  - Each item in the index is called an **element**.
  - Arrays in Swift must be of the *same type*.



Working with Arrays

These are functions and a property that apply to an array variable. To use them, put the name of the array, followed by a dot, and the function/property name.

- `.append(newElement)`: Add an element to the end
- `.insert(newElement, at:)`: Add an element at a certain index. Adding at 0 would add it to the front.
- `.remove(at:)`: Remove an element at a certain index
- `.count`: How many items in the array. This is not a function; there are no brackets!

Exercise 1

There are various ways to solve some of these!

- Create an array, `[1, 2, 3, 4, 3, 5, 6, 7, 8, 9, 10]`
- Delete the fifth element (not the element with index 5)
- Insert these elements at the end: `11, 12, 13, 14`
- Insert these elements at the front: `-3, -2, -1`
- Insert `0` at the right place
- Print everything out to make sure you have the right ordered list from `-3` to `14`

Unit 4.1

Arrays

What are Arrays?

1
2
3

Ordered Lists of Data

123

Indices & Elements

$f(x)$

Functions & Properties

Loops

Unit 4.2: Looping through data in your array

Looping through an array

- Now that we have arrays, we want to be able to work with the data within.
- To do that, you need to work with each individual element of the array — you might want to print each out, change each of them, sum them up, or find the largest in an array of numbers.
- Computers handle these operations sequentially, element by element. To do this, we use a **for loop**.

Looping through an array

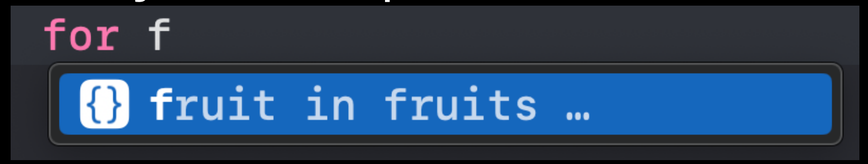
To go through an array and do something with each item, you can use a for-in loop. Here, we print out the name of each fruit.

```
let fruits = ["apple", "banana", "orange", "pineapple"]

for fruit in fruits {
    print(fruit)
}
```

Look! Fruit loops! (Sorry.)

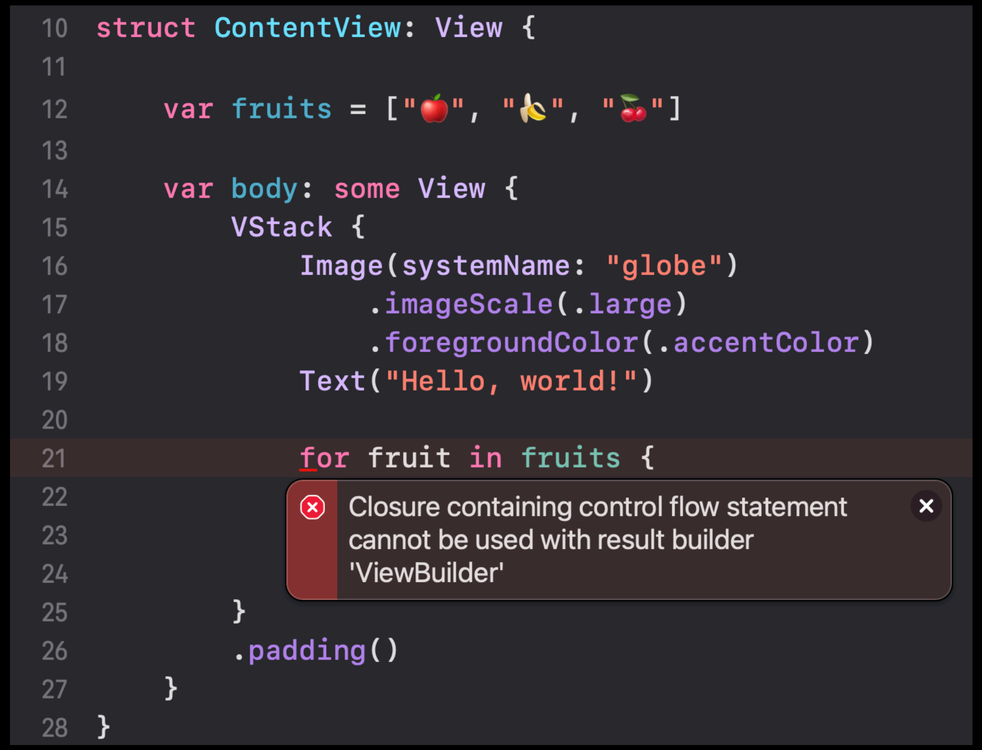
Loop Naming Convention

- By convention, we use a *plural noun* for an array's name:
  - E.g. `fruits`, `people`, `scores`, `dentalCheckups`
- Within a for-loop, use a *singular noun* for the iterator, the item that is being considered in each round of the loop:
  - `for singularNoun in pluralNouns`
  - `for person in people`
- Xcode's code completion will help you with this when you try typing out your loop!



Where for-loops can go in SwiftUI

- Unlike if statements, for-loops can't just show up anywhere in SwiftUI!
- That's because for-loops are "do something" logical code, whereas most of SwiftUI (after `var body`) has "show something" view-builder code.
- If you need to, you can put for-loops in a `Button`'s action, or various other "do something" code places such as the `onAppear` modifier.
- If you want to create a loop to repeat various SwiftUI views, you can use a `ForEach` view, which we'll cover next.



Exercise 2A

- Create an array, `["Banana", "Potato", "Papaya"]`
- Loop through the array, and print out the following:

```
I like to eat Banana!

I like to eat Potato!

I like to eat Papaya!
```

Exercise 2B

- Create an array, `[2, 4, 6]`
- Loop through the list, and print out the square of each

```
4
16
36
```

Exercise 2C

- Create an array, `[1, 2, 3, 5, 6, 8, 100]`
- Loop through the list, and print out only the odd numbers.

  ```
  1
  3
  5
  ```

- You'll need an `if` statement before printing!
- E.g.: `if item % 2 != 0`

Exercise 2D

- Create an array, `[1, 2, 300, 6, 7, 100]`
- Loop through the list, and print out the sum of all the numbers.

  `416`

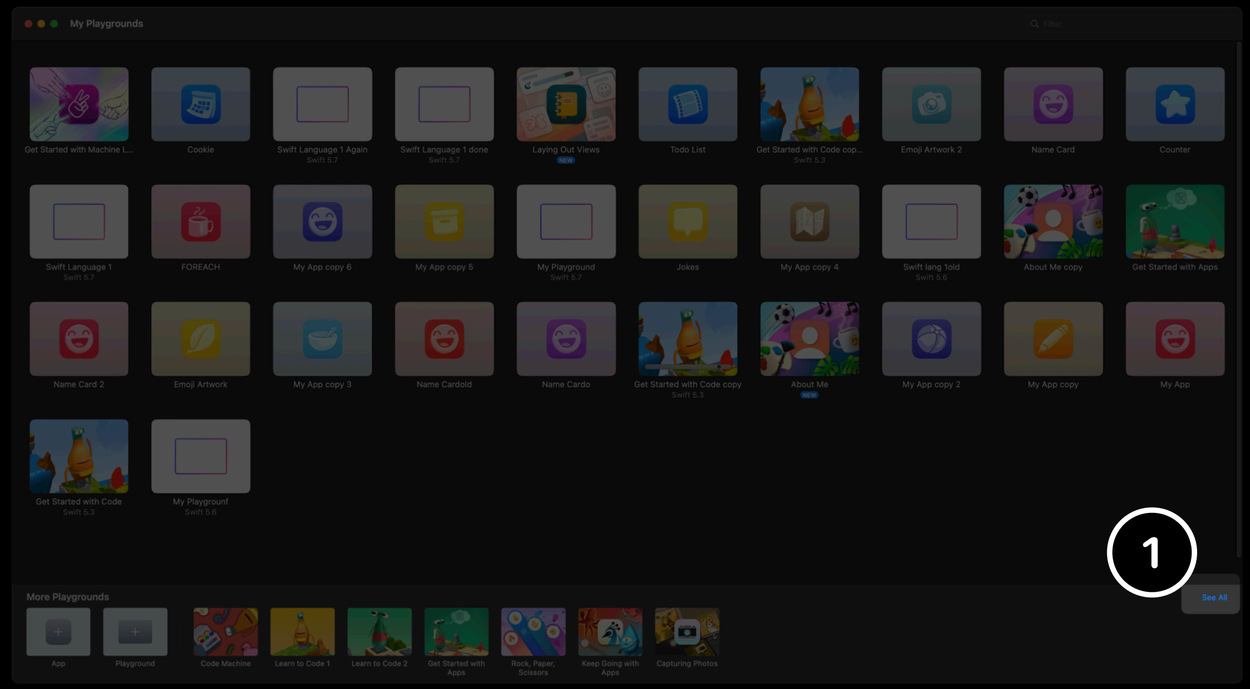
- You'll need:
  - A variable to store the total sum so far
  - This variable should start at 0
  - Keep adding the new number to this variable

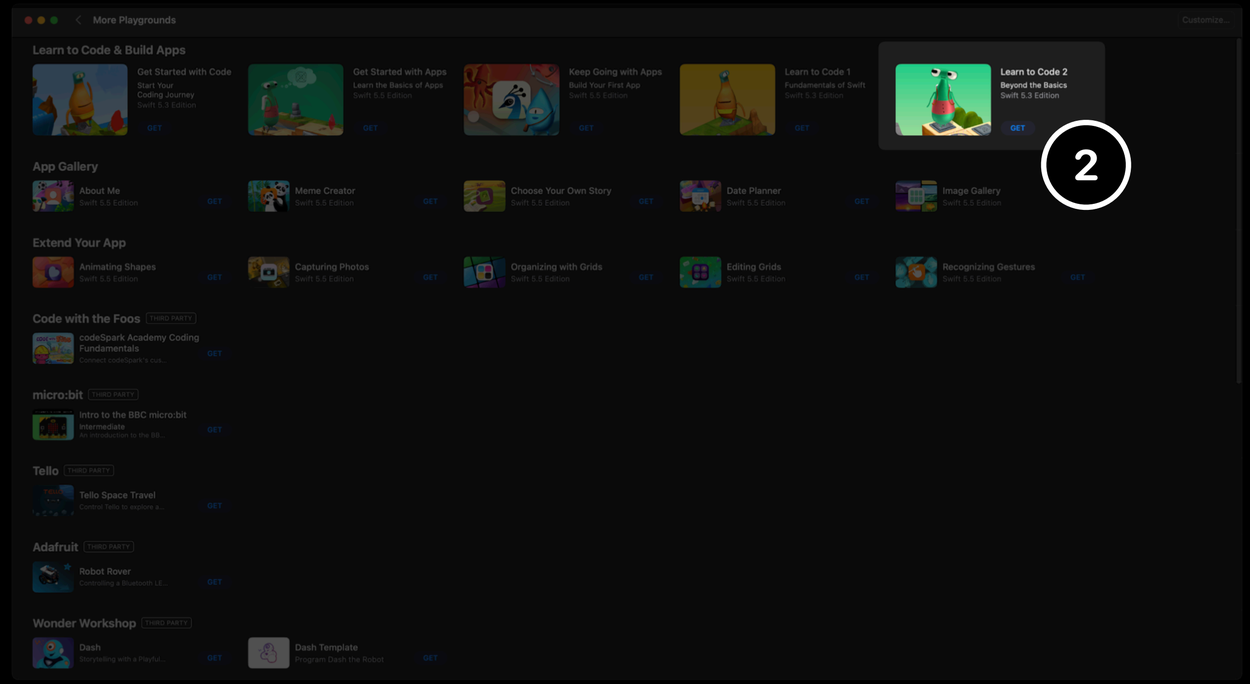
Learn To Code Arrays

Practise arrays with the **Learn To Code 2** series on Swift Playgrounds!

Adding the Playground

Open Swift Playgrounds. On the home screen, click on **See All** ① in the bottom right, and download **Learn to Code 2** ②.



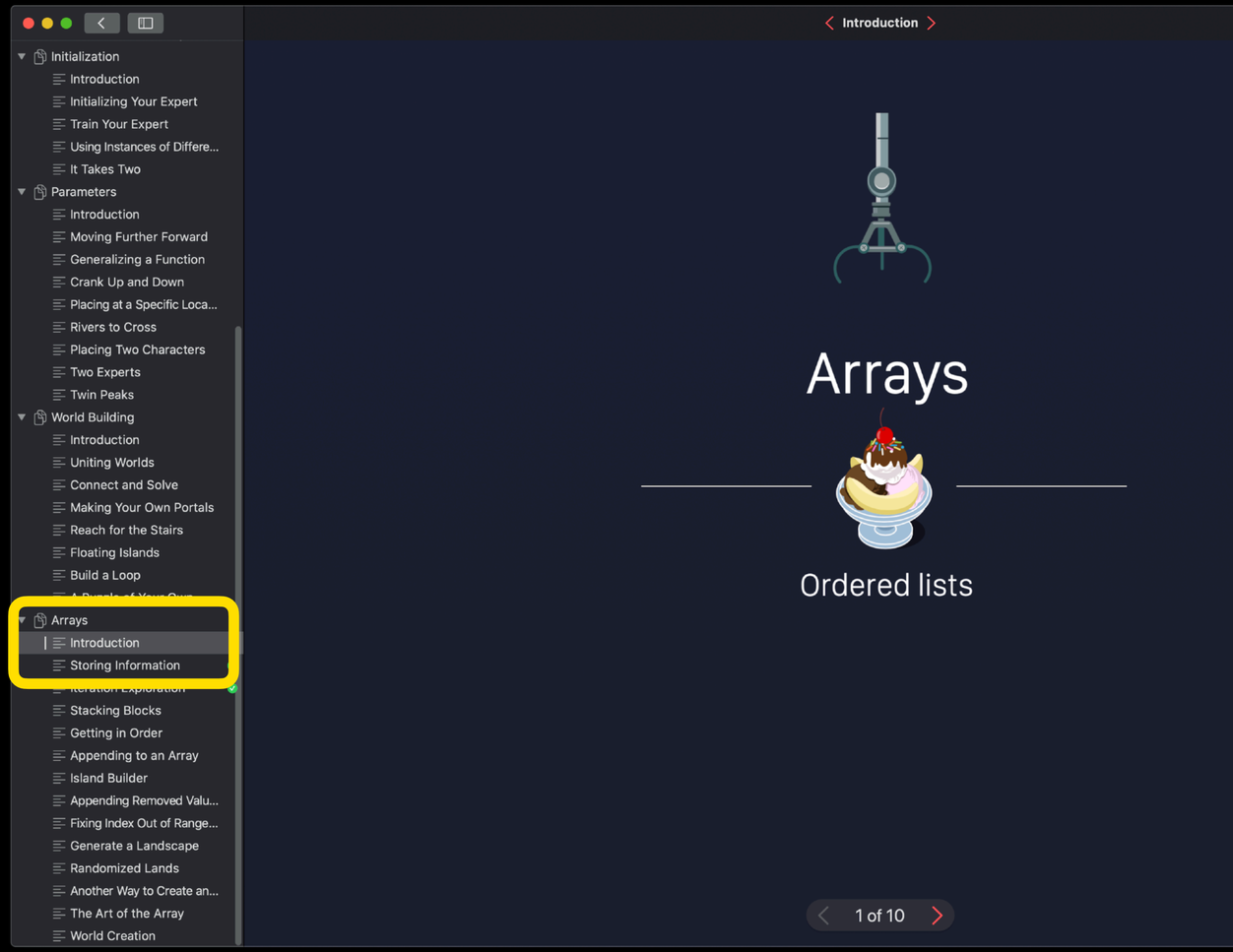


Recap: What is Learn to Code?

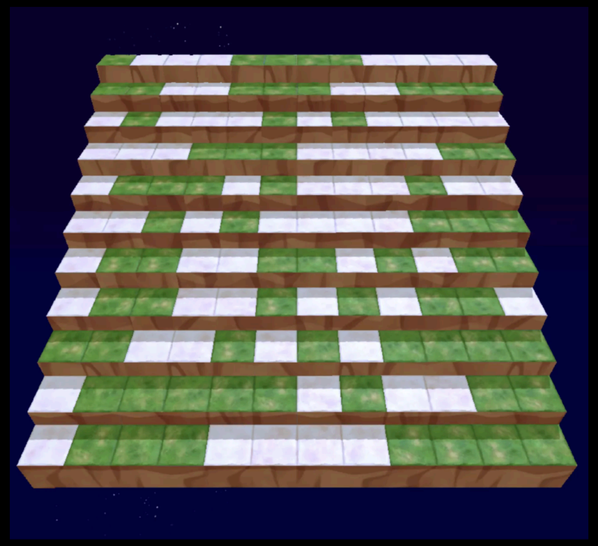
- A set of interactive coding puzzles
  - Type code on the left side, after reading the instructions
  - Use two fingers on your trackpad on the right graphical window to rotate and enlarge it
- Start with two basic commands: `moveForward()` and `collectGem()`
  - There are many more! Look at the auto-complete options.
- Complete each level by meeting the level objectives, or collecting all the gems.

Exercise 2E

- Open **Learn to Code 2**
- Open the navigation drawer, and scroll down to **Arrays**
- Go through the introduction, then start from **Storing Information**
- Share your final puzzle, **World Creation**, with your classmates!



Examples



Exercise 2F (optional)

- This one is for those who have taken, or are taking, computer science classes!
- Create an array, `[5, 8, 9, 6, 7, 10, -1, 39, 14, 11]`
- Find the largest number in the array, and print it out.

  `39`

- You'll need:
  - A variable to store the largest number found so far
  - Keep checking each number against this, and replace the variable when it's no longer the largest.
- This is an interesting demonstration of how computers work with arrays sequentially — it might be easy for us to see the largest number at a glance, but computers can't. That said, if there were 500 elements in the array, could you find the number easily?

Plus

Looping through an Array

For Loops

Unit 4.2

Loops

For loops in SwiftUI

Naming Conventions

Learn to Code 2

Structs

Unit 4.3: Custom types to let us make chonky variables to store more than one type of value (that can serve as SwiftUI `Views`)

Structs & Classes

- *Classes and structures are general-purpose, flexible constructs that become the building blocks of your program's code. You define properties and methods to add functionality to your classes and structures by using exactly the same syntax as for constants, variables, and functions.*
  ~ The official Apple Developer documentation

Let’s Consider Cars

Let’s say you want to make a new type, to collect information about a car. What variables might you use to describe *how a car looks and works*?

Car Properties

These are ***variables*** storing information about the car.

- Colour `colour: String`
- Number of wheels `numWheels: Int`
- Automatic or Manual? `isAuto: Bool`
- Air conditioned `hasAircon: Bool`
- Top Speed `topSpeed: Double`
- How many people can fit `numFittable: Int`

Keep Considering Cars

What *functions* might you use to control, or get information about, a car?

Car Methods

These are *functions* that the car can execute.

- Turn `func turn(to direction:String)`
- Accelerate `func accelerate(by axn: Double)`
- Honk `func honk(loudness: Int)`
- Stop `func stop()`
- Got fuel? `func runningOutOfFuel() -> Bool`
- Is broken? `func isBroken() -> Bool`

Recap:
Argument labels & param names

About `func turn(to direction:String)`:
We saw this briefly in our intro to Swift—this is an example of having a different argument label (`to`) and parameter name (`direction`). Documentation.

Recap:
Return values

About `func runningOutOfFuel() -> Bool`:
We saw this briefly in our first Swift language unit —this is an example of a function that returns a value (in this case, a `Bool`) to the caller. Documentation.

Creating our Car struct

We create a simple `Car` struct using the keyword `struct`, then add its properties by defining variables and their types.

```
struct Car {
    var colour: String
    var make: String
    var model: String
    var topSpeed: Int
    var maxFuel: Double
    var fuelLeft: Double
}
```

Instantiating our Car

We now need to make an actual *instance* of a `Car`. To do this, use the struct name, `Car`, and in parentheses, list out all the properties. Auto-complete will help.

```
var car = Car(colour:"red", make:"Tesla", model:"Model X", topSpeed:200, maxFuel:50.0, fuelLeft:40.0)

// Remember, car is just the name of our variable, and it
// can be called myCar, theCar, iCannotAffordThisCar, etc
```

Adding methods to a struct

Adding methods to a struct is as simple as defining functions inside the struct's code block definition.

```
struct Car {

    // … properties, etc here

    func honk() {
        print("Hjönk") // Goose horn?
    }

    func stillGotFuel() -> Bool {
        if fuelLeft >= 1.0 {
            return true
        } else {
            return false
        }
    }
}
```

Aside: Easy Bool returning

We can simplify that second method by removing 4 whole lines of code and about 6 words. What a life hack!

```
struct Car {

    // … properties, etc here

    func honk() {
        print("Hjönk") // Goose horn?
    }

    func stillGotFuel() -> Bool {
        fuelLeft >= 1.0 // So efficient!!!
    }
}
```

Default property values

When you give a property a value in the struct definition, you don’t have to set it when you instantiate the property (though you can, if you want to).

```
struct Cat {
    var name: String
    var breed: String

    var age: Int
    var willBiteYou = true // default value
}

// My cat bites. No need to set willBiteYou!
var myCat = Cat(name: "Pommy", breed: "Domestic Shorthair", age: 12)

// Presumably this one doesn't
var fictionalCat = Cat(name: "Garfield", breed: "Persian Tabby", age: 44, willBiteYou: false)
```

Exercise 3A: Add a Method

- Add a new **describe** method, that tells you about the instance in question, using its own properties.
- This will involve some string interpolation!

```
var car = Car(colour:"red", make:"Tesla", model:"Model X",
topSpeed:200, maxFuel:50.0, fuelLeft:40.0)

myCar.describe()

// should print out:
// This car is a red Tesla Model X with a top speed of
// 200 km/h.
```

Exercise 3B: Two Cars

- Make 2 `Car` instances, `car1` and `car2`, with your favourite makes and models.
- Write a **function** (*not* a method in `Car`!) to compare their top speeds.
- To test this `compareCar` function, you might write something like the below:

```
var car = Car(colour:"red", make:"Tesla", model:"Model X",
    topSpeed:200, maxFuel:50.0, fuelLeft:40.0)
var car2 = Car(colour:"red", make:"Mitsubishi", model:"Lancer",
    topSpeed:150, maxFuel:60, fuelLeft:20.0)
compareCars(car, car2)

// should print out:
// The Model X is faster than the Lancer
```

Exercise 3C: Make Your Own

- Design your own `Struct` for a pet dog or cat
- What properties would you need?
- What methods would be useful?
- Create an instance of it and call a method
- Get creative!

```
cat.hasPomeloOnHead = true
cat.isAnnoyed = true
```

(This is the slide author's cat. Sorry cat.)

Structs in SwiftUI

- At this point, you might recall that we've seen structs before in SwiftUI — near the beginning of your code, your **ContentView** is declared as a `struct`!
  - In fact, *just about every SwiftUI view is a struct.*
  - They're also named like structs, with each type starting with an uppercase letter, like **ContentView**, `Text`, `Image`.
- Structs are very lightweight data structures, which is great for SwiftUI, where re-renders happen often
  - This lets your device create, re-render, and discard views very quickly.

Structs

```
struct Car {
```

Unit 4.3

Structs

Instantiation

Methods

Classes

Plus

Properties and default values

Classes

Unit 4.3+: Another way of storing methods and properties (Optional)

Plus

Structs vs. Classes

We'll use structs for most of what we do, but it's good to know that classes *exist* — you will see them pop up in a later unit. The main differences between the two:

- Classes require explicit initialisers
- Classes pass by reference
- Structs can't inherit. (Like pets vs. children…?)
- Structs are smaller and faster. (Like pets vs. children…?)

We'll talk a bit about the first two points.

Initialisers

Creating a new class

Example Class

This `Cat` class won't work without the highlighted lines labelled with `init` — that's its *initialiser*.

```
class Cat {
    var name: String
    var breed: String

    var age: Int
    var willBiteYou = true // default value

    init(name: String, breed: String, age: Int) {
        self.name = name
        self.breed = breed
        self.age = age
    }
}
```

Classes need custom initialisers

- When we made structs (e.g. `struct Car`), Swift auto-generated the *initialiser* for us (e.g. `Car(make:, model:)`).
  - When we create a struct with default values, Swift auto-generated multiple initialisers for us!
- An initialiser is a function that runs in order to create your class or struct. The initialiser accepts parameters, which it stores to create your class or struct.
- Other languages may call this a *constructor*.

Setting parameters

- Within your initialiser, you must set every property that does not already have a value
  - This means, if you declare your variable like this, you do not need to set its value in your initialiser

```
var willBite = true
```

  - However, you will need to set an initialiser for this:

```
var name: String
```

- Think of your initialiser like running some code to set up your instance.
  - Structs can also have initialisers!

What's this `self.` thing about?

```
self.name = name
```

- This has to do with *variable scope*.
- In this case, the code is found within the initialiser, which takes in a `name: String` as a parameter. This means the `name` variable referenced here is going to refer to the incoming parameter.
- However, every property needs a value. Therefore, you need to set the value of the *actual* `name` in the class. In order to do this, you use `self.name` to refer to the `name` variable owned by the class.

Overloading

- Let's say you want to create another initialiser for your class — you can do that, as long as it takes in different parameters.
- This is called overloading. It lets you create functions and initialisers with different implementations.
  - We saw this before, in Swift language 1, when we defined two functions with the same name, using argument labels for one of them.
- See: https://en.wikipedia.org/wiki/Function_overloading

Another initialiser

```
class Cat {
    var name: String
    var breed: String

    var age: Int
    var willBiteYou = true // default value

    init(name: String, breed: String, age: Int) {
        self.name = name
        self.breed = breed
        self.age = age
    }

    init() {
        // sorry for the awful cat names
        let catNames = ["Apple", "Banana", "Cherry", "Durian"]
        name = catNames[Int.random(in: 0..<catNames.count)]
        breed = "Random Cat"
        age = 0
    }
}
```

Notice how you don't need to use `self.name` here, because this time, it's very clear you can only be referring to the name from the class.

Pass by reference

Sending your variable around to everyone to change

Structs pass by value

When you assign a struct instance to another variable, or pass it as an argument, you make a copy of that struct (or rather, its values) that doesn't affect the original.

```
struct Rect {
    var width: Double
    var height: Double
}

var myRectangle = Rect(width: 10.0, height: 15.0)
var copyRectangle = myRectangle
myRectangle.width = 99
print(copyRectangle.width) // still 10.0
```

Classes pass by reference

If you change an instance of it, everything else “pointing to it” (when you do assignment, or passing it as an argument to a function) also sees the change.

```
class Rect {
    var width: Double
    var height: Double
    init(width: Double, height: Double) {
        self.width = width
        self.height = height
    }
}

let myRectangle = Rect(width: 10.0, height: 15.0)
let copyRectangle = myRectangle
myRectangle.width = 99
print(copyRectangle.width) // now 99.0!
```

Passing by reference, explained

- We only have one `Rect` instance, and when we assign `copyRectangle` to `myRectangle`, we're just asking the latter to *refer to* what the former is pointing to.
- This is what it means by *pass by reference*. You aren't actually initialising a new instance of the class every time you pass it around!

```
let myRectangle = Rect(width: 10.0, height: 15.0)
let copyRectangle = myRectangle
myRectangle.width = 99
print(copyRectangle.width) // now 99.0!
```

Passing by reference, explained

- One more thing — you may have noticed `myRectangle` and `copyRectangle` are declared as constants. But we're able to change their `width` property!
- As these two are just references, they aren't actually changed unless you actually initialise a brand new `Rect`.

```
let myRectangle = Rect(width: 10.0, height: 15.0)
let copyRectangle = myRectangle
myRectangle.width = 99
print(copyRectangle.width) // now 99.0!
```

Value Types vs. Reference Types

- Structs are value types
  - They actually *contain* the value
- Classes are reference types
  - They just say *“Hey, I’m pointing to this value over here!”*
- Read this blog post to learn more: *https://developer.apple.com/swift/blog/?id=10*

What’s more?

- There's an entire field of object oriented programming which… we won’t need to go into much with SwiftUI. These include:
  - Inheritance and subclassing
  - Overriding methods and properties
- With UIKit, you might see these a lot, as all your `ViewController`s in UIKit are classes.
  - We’ll also see classes when we work with ViewModels, in a later unit.
- Interested? Read more at https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html.

Closures

Unit 4.4: Functions as variables, and functions taking in functions as parameters.

Here are two squaring functions

```
// In function form
func squareFunc(_ num: Int) -> Int {
    num * num
}

// In closure form
let squareClosure = { (_ num: Int) -> Int in
    num * num
}

squareFunc(5) // 25! 🤯
squareClosure(5) // 25!! 🤯 🤯
```

From func to closure

```
func square(_ num: Int) -> Int {
    num * num
}
```

From func to closure

```
let square = { (_ num: Int) -> Int in
    num * num
}
```

Closure syntax

Parts in yellow are required. Sometimes, you can omit the return type, when the closure is used as a parameter. Don’t worry too much about this syntax — you won’t be *writing* closures much; you’ll be *using* them.

```
{ (parameters-if-any) -> ReturnType in
    // code to be run here
}
```

Wait… what does `in` mean??

Since Swift places such importance on naming… what does `in` represent then? Let's hear from the creator of Swift, Chris Lattner.

Link: https://www.youtube.com/watch?v=bMICCC-vvLE. Feel free to skip to 2:49.

Closure example: Sorting

One common closure you'll encounter is with *sorting*. Swift has a collection function called `.sorted()`, which will normally just sort numerically or alphabetically…

```
var scores = [90, 50, 80]
let sortedScores = scores.sorted()
print(sortedScores)
```

Using closures

What happens if you try and sort an array of `structs`, though? You'd need to use a custom sorting closure, to let `sorted` know what to do.

```
struct Student {
    var name: String
    var examScore: Int
    var height: Double
}

var students = [
    Student(name: "Alice", examScore: 90, height: 1.7),
    Student(name: "Bob", examScore: 50, height: 1.5),
    Student(name: "Charles", examScore: 60, height: 1.1)
]

let rankedStudents = students.sorted(by: { (s1: Student, s2: Student) -> Bool in
    return s1.examScore < s2.examScore
})

print(rankedStudents) // Bob, Charles, Alice
```

Simplifying closure syntax

“Syntactic sugar”

Version 0

Defining a closure as a separate variable, and using them in the sorted method.

```
let sortingClosure = { (s1: Student, s2: Student) -> Bool in
    return s1.examScore < s2.examScore
}

students.sorted(by: sortingClosure)
```

Version 1

“Standard” closure syntax! 😴

```
students.sorted(by: { (s1: Student, s2: Student) -> Bool in
    return s1.examScore < s2.examScore
})
```

Version 2

No need parameter types… we can infer that based on the type of the array 😄

```
students.sorted(by: { (s1, s2) -> Bool in
    return s1.examScore < s2.examScore
})
```

Version 3

No need return type, we can infer that from the “sorted” function, which expects a boolean return 🤷‍♂️

```
students.sorted(by: { (s1, s2) in
    return s1.examScore < s2.examScore
})
```

Version 4

Let's just assume that the last line of the closure is what's meant to be returned, so we remove the `return` keyword. This *implicit return* technique is used often in SwiftUI!

```
students.sorted(by: { (s1, s2) in
    s1.examScore < s2.examScore
})
```

Version 5

You know what, why bother naming the two variables? 🤣 Let's just give them indices. We'll also get rid of `in`. Bye!!!

```
students.sorted(by: {
    $0.examScore < $1.examScore
})
```

Version 6

If the last argument of a function is a closure, let's use a shorter syntax with one fewer pair of parentheses `()` and no `by`. This *trailing closure syntax* is used everywhere in SwiftUI!

```
students.sorted {
    $0.examScore < $1.examScore
}
```

What we've done

```
students.sorted(by: { (s1: Student, s2: Student) -> Bool in
    return s1.examScore < s2.examScore
})
```

What we've done

```
students.sorted { $0.examScore < $1.examScore }
```

Trailing closure syntax in SwiftUI

- When the closure is a parameter to a view, you can “move it outside” for a cleaner look. The way it works:
  - If the view takes a single closure, close the brackets before the closure parameter, and switch to { }
  - Multiple closures? Close the bracket before the first; label subsequent ones.

```
Button(action: {
    print("Hello")
}, label: {
    Text("Hello")
})
```

```
Button {
    print("Hello")
} label: {
    Text("Hello also")
}
```

SwiftUI is full of closures

Take this code, for example. This is a standard `VStack`, which an onAppear modifier, containing code that runs when it appears.

```
VStack {
    Button {
        print("Hello")
    } label: {
        Text("Press me to say hello")
    }
}.onAppear {
    print("I APPEARED")
}
```

Crouching Tiger, Hidden Closure

The `VStack`'s contents are actually a closure! In its original form, it takes in a function under its `content` parameter.

```
VStack(content: {
    Button {
        print("Hello")
    } label: {
        Text("Press me to say hello")
    }
}).onAppear {
    print("I APPEARED")
}
```

Crouching Tiger, Hidden Closure 2

The `onAppear` modifier's contents are actually a closure, given to the `perform` parameter.

```
VStack(content: {
    Button {
        print("Hello")
    } label: {
        Text("Press me to say hello")
    }
}).onAppear(perform: {
    print("I APPEARED")
})
```

Crouching Tiger, Hidden Closure 3 & 4

The `Button` has *two closures*!!!!! One for the `action` to be run, and one for the `label` to be shown.

```
VStack(content: {
    Button(action: {
        print("Hello")
    }, label: {
        Text("Press me to say hello")
    })
}).onAppear(perform: {
    print("I APPEARED")
})
```

But they’re hidden for a reason

… because this code is much easier to read, write, and understand.

```
VStack {
    Button {
        print("Hello")
    } label: {
        Text("Press me to say hello")
    }
}.onAppear {
    print("I APPEARED")
}
```

Body?

Aahhh!! *Is everything a closure?* Is `var body` a closure?!?!?!? 🤯🤯
🤯 OK, no, `body` is not, it's a computed property. We'll see those later.

```
struct ContentView: View {
    var body: some View {
        VStack {
            Button {
                print("Hello")
            } label: {
                Text("Press me to say hello")
            }
        }.onAppear {
            print("I APPEARED")
        }
    }
}
```

ContentView.swift

Declaring closures for SwiftUI

You *could* declare a closure elsewhere in your code, to use in a SwiftUI view's content body. *There's no real need to do this!* But it's good to know.

```
struct ContentView: View {

    let myText = { () -> Text in
        Text("Hello world!")
    }

    var body: some View {
        VStack(content: myText)
    }
}
```

Exercise 4

We've written two SwiftUI elements below using closure argument syntax. Can you write them out using trailing closure syntax?

```
import SwiftUI

VStack(alignment: .trailing, spacing: 10, content: {
    Text("Hello! This is a Text in a VStack”)
})

Stepper("More Nutella", onIncrement: {
    print("Yay!")
}, onDecrement: {
    print("What! No.")
})
```

ContentView.swift

`in`

Func → Closure

Syntax

```
{ (parameters-if-any) -> ReturnType in
    // code to be run here
}
```

Sorting Closures

Unit 4.4

Closures

Trailing Closure Syntax

Closures in SwiftUI

Collection Functions

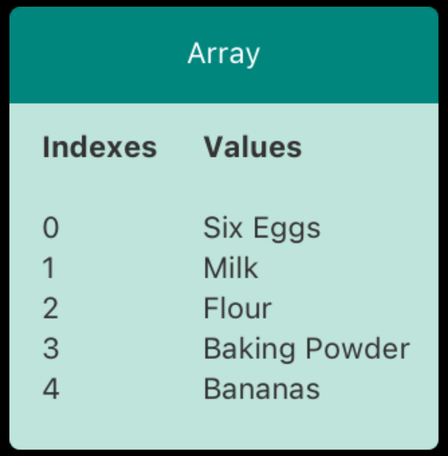
Unit 4.5: Working with groups of items, like arrays, using closures

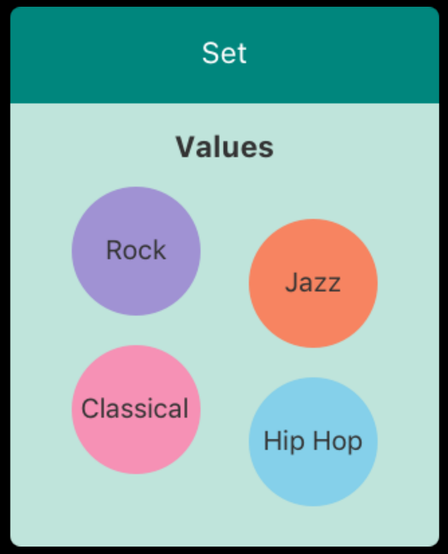
Collection Types

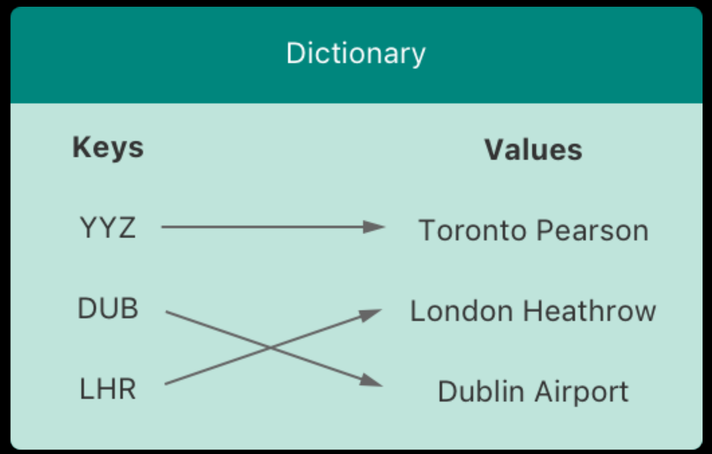
- There are several collection types; the one you just learned about is the array.
  - Think of these as different ways of storing multiple pieces of information
  - Arrays are an *ordered list* of items.
- Others you might see in Swift: Dictionaries, Sets, Ranges.
  - Dictionaries, like their real-world analogues, store items in key-value pairs, e.g. “p is for potato”, or more broadly, “the key potato has value 1”.
  - Sets are like a bag of unsorted items. Just throw things inside!
  - Ranges are a bunch of numbers, you may have seen them before in Learn To Code, such as `0..<10`.

More reading

- If you're super into collections and set theory, please take a look at the Swift documentation, and have tons of fun there
- *https://docs.swift.org/swift-book/LanguageGuide/CollectionTypes.html*







Collection functions

- Swift collections such as arrays have built-in collection functions: `.map`, `.filter`, and `.reduce`
- They each take in a *closure*, whose job is to *do something with each of the items in the collection*
  - `map` will create a new collection, with the closure mapped on each item
  - `filter` will create a new collection, based on a certain condition defined in the closure
  - `reduce` will create a single item, defined by the closure
- This is intermediate-level Computer Science, but we think you'll find these concepts handy when dealing with SwiftUI.
  - They're also called higher-order functions, so now you can feel clever about knowing them.

Map

- Goes through every item in an array, and changes it, returning a new array.
- For example, you might want to go through an array of `String`s and add more text to each of them:
  - Before: `["Alice", "Bob", "Charles"]`
  - After: `["Evil Alice", "Evil Bob", "Evil Charles"]`
- To do this with a for loop, you would need to loop through each, modify it, and add it back to a new array.

Map Example 1

```
let people = ["Alice", "Bob", "Charles"]

// For loop
var evilPeople: [String] = [] // empty String array
for person in people {
    evilPeople.append("Evil \(person)")
}

// Mapped
let evilPeopleMapped = people.map { "Evil \($0)" }
```

Map Example 2

```
let nums = [1, 2, 3, 4, 5]

// For loop
var triplesLoop: [Int] = [] // empty Int array
for num in nums {
    loopedTriples.append(num * 3)
}

// Mapped
let triplesMapped = nums.map { $0 * 3 }
```

Filter

- Goes through every item and figure out if it should belong in a brand new filtered array
- For example, you may have an array of integers, and you want a new array which only has the integers that are greater than 3.
  - Before: `[1, 2, 3, 4, 5]`
  - After: `[4, 5]`
- To do this with a for-loop, you'd have to create a new empty array, loop through the existing array, check with an if-statement if each element fits, and add it to the new array.

Filter Example 1

```
let numbers = [1, 2, 3, 4, 5]

// For loop
var numbersGreaterThan3Array: [Int] = [] // empty Int array
for number in numbers {
    if number > 3 {
        numbersGreaterThan3Array.append(number)
    }
}

// Filtered
let numbersGreaterThan3Closure = numbers.filter { $0 > 3 }
```

Filter Example 2

```
let names = ["Alvin", "Beatrice", "Chimpanzee"]

// For loop
var longNamesLoop: [String] = [] // empty String array
for name in names {
    if name.count > 6 {
        longNamesLoop.append(name)
    }
}

// Filtered
let longNamesFilter = names.filter { $0.count > 6 }
```

Reduce

- Combine (reduce) a collection into a single value.
- Perhaps you want to add every element of an integer array into a total.
  - Before: `[1, 3, 5, 7, 9]`
  - After: `25`
- To do this with a for loop, you'd have to initialise a value outside the loop, and keep adding to it inside the loop.

Reduce Example 1

- In the reduced version, the first parameter is the initial value.
- `$0` is the partial result so far, from the previous iterations.
- `$1` is the value *at* that iteration.

```
let numbers = [1, 3, 5, 7, 9]

// Looped method
var loopedSum = 0
for num in numbers {
    loopedSum = loopedSum + num
}

// Reduce method
let reducedSum = numbers.reduce(0) { $0 + $1 }
```

Reduce Example 2

- Here, we set the initial value to an empty string
- When we go through the reduce function, it adds the empty string to a comma, followed by the current element. Try this out — it won't quite come out as expected!

```
let fruits = ["apples", "bananas", "pineapples", "tomatoes"]

fruits.reduce("") {
    $0 + "," + $1
}
```

Remember these exercises?

- From an earlier sub-unit:
  - Given an array of food items, print out `I like to eat food item`!
  - Given an array of integers, print out each number's square
  - Given an array of integers, print out only the odd numbers
  - Given an array of integers, add them up
- With what you know about `map`, `filter`, and `reduce`, can you solve these problems again in one line each?

Exercise 5A

- Create an array, `["Banana", "Potato", "Papaya"]`
- Using a collection function, print out the following:

```
I like to eat Banana!

I like to eat Potato!

I like to eat Papaya!
```

Exercise 5B

- Create an array, `[2, 4, 6]`
- Using a collection function, print out the square of each

```
4

16

36
```

Exercise 5C

- Create an array, `[1, 2, 3, 5, 6, 8, 100]`
- Using a collection function, print out only the odd numbers.

```
1

3

5
```

Exercise 5D

- Create an array, `[1, 2, 300, 6, 7, 100]`
- Using a collection function, print out the sum of all the numbers.

```
416
```

Why collection functions?

- In SwiftUI, you often want to try and express code briefly
- For instance, this is code you might see later, when displaying the number of todos that aren't done in an array:

```
Text("\(todos.filter { !$0.isCompleted }.count ) todos undone”)
```

- This means:
  - Go through the array of todos
  - Filter out those that are not done (assuming the `Todo` struct has a boolean property called `isDone`)
  - Get the count of those that are left

$f(x)$

Functions

Collection Types

Unit 4.5

Collection Functions

Map

Filter

Reduce

Imperative vs. Declarative

Unit 4.6: Different ways of working with arrays, and how this relates to SwiftUI

Imperative vs. Declarative

- When we looped through the arrays, we were doing things in an *imperative* way — “here’s how exactly we want to solve this problem”.
- When working with the collection functions, we worked *declaratively* — “I know you can do things in a certain way, I just want you to apply this code to each item”.
- These are two different and distinct styles of coding, and we thought it’d be interesting to point them out!

Exact Instructions Challenge

- The video, by YouTube family Josh Darnit, has the dad challenging his kids to give him exact instructions to make a peanut butter and jelly sandwich.
- We'd do this the declarative way:
  - Make a sandwich!
- Computers, however, expect imperative instructions:
  - Get two slices of bread
  - Get a butter knife
  - … and more

Example: Younger sibling

- Imagine you have a younger sibling, age 2, who's only just learning to talk. How would you ask him to get you a packet of juice from the kitchen?
  - Go to the kitchen, that's the one with the glass door
  - Open the door (push, not pull)
  - Find the second cabinet (no, not that one, the other, yes)
  - Look for the juice packet (the one in white, with an apple on it)
  - … I'll just get it myself
- Imagine your sibling at age 8.
  - Hey! Get me an apple juice from the kitchen OK?
  - What do you mean, "no"?!?!?

SwiftUI is declarative

- When you work in SwiftUI, say creating a `Button`, you don't specify:
  - The co-ordinates of the button on the screen
  - The memory address for your code to be able to update the button, and the memory address of where your action code is
  - The font size, colour, etc. — these are set later using modifiers, not when you create it
- … you just say, give me a `Button`, and iOS puts it on screen for you!
- These are things you would have had to do in UIKit, the original iOS software development kit.
  - It's not that UIKit is worse, though; just older and different.

Exercise 6+ (optional)

Here's an array:

```
let numbers = [14,20,-3,42,6,8,9,-10,-99,14,6]
```

Using the collection functions, ***using one line of code each***:

- Sort the square of all numbers in descending order
- Get all the numbers that are multiples of 3, sorted in ascending order
- Find the product of all the negative numbers

Plus

Declarative SwiftUI

```
Button {
    action:
} label: {
    label:
}
```

Loops vs. Collection Functions

Unit 4.6

Imperative vs. Declarative

Imperative UIKit

```
let button = UIButton(
```

Unit 4.1

Arrays

Unit 4.2

Loops

Unit 4.3

Structs

Unit 4.6

Imperative vs. Declarative

Unit 4

Overview

Unit 4.5

Collection Functions

Unit 4.4

Closures

Unit overview

Back to Swift!

- Arrays
  - Loops, a brief introduction
- Structs
  - Classes (optional)
- Closures
- Collections and collection functions
  - Doing things imperatively vs. doing things declaratively