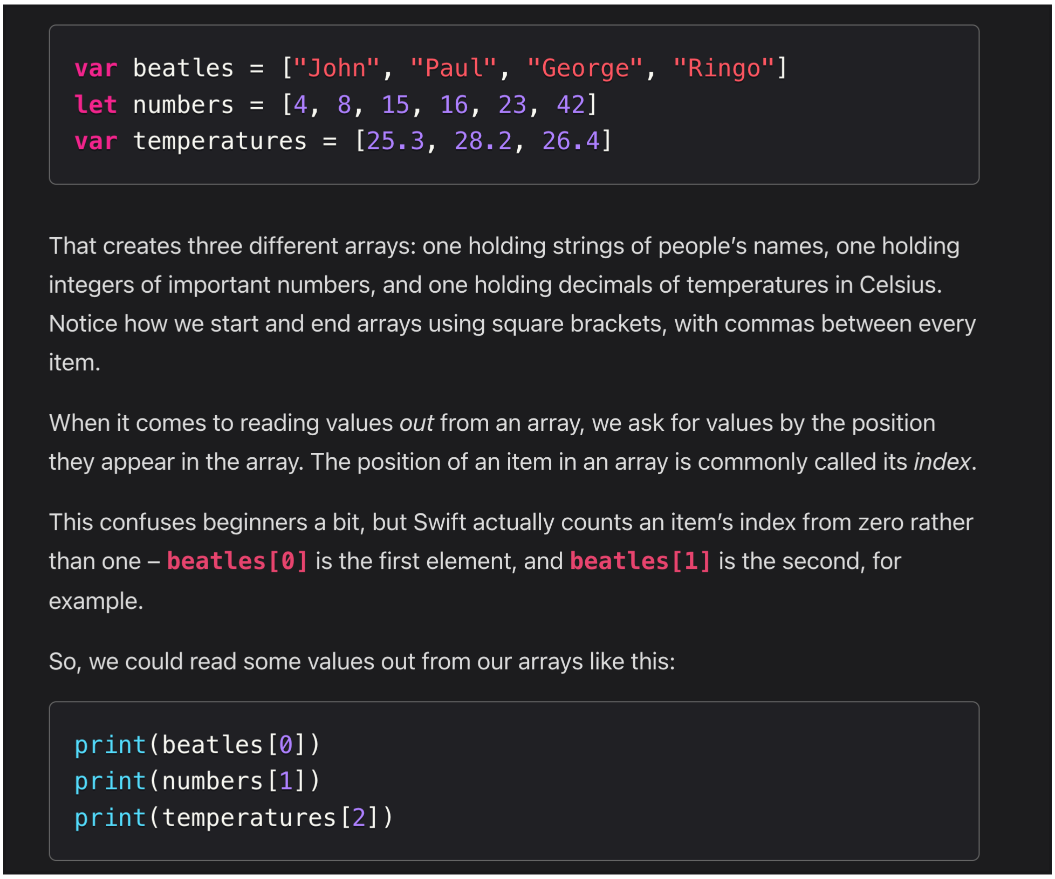
Day 3



```
var beatles = ["John", "Paul", "George", "Ringo"]
let numbers = [4, 8, 15, 16, 23, 42]
var temperatures = [25.3, 28.2, 26.4]
```

That creates three different arrays: one holding strings of people's names, one holding integers of important numbers, and one holding decimals of temperatures in Celsius. Notice how we start and end arrays using square brackets, with commas between every item.

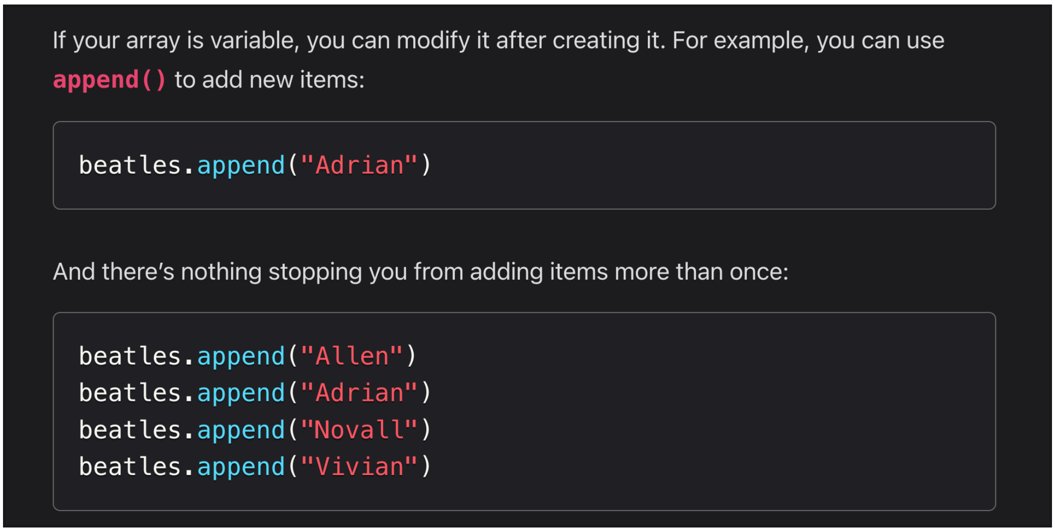
When it comes to reading values *out* from an array, we ask for values by the position they appear in the array. The position of an item in an array is commonly called its *index*.

This confuses beginners a bit, but Swift actually counts an item's index from zero rather than one – `beatles[0]` is the first element, and `beatles[1]` is the second, for example.

So, we could read some values out from our arrays like this:

```
print(beatles[0])
print(numbers[1])
print(temperatures[2])
```

Array in swift is like vectors where adding data is possible without specifying the size of array .

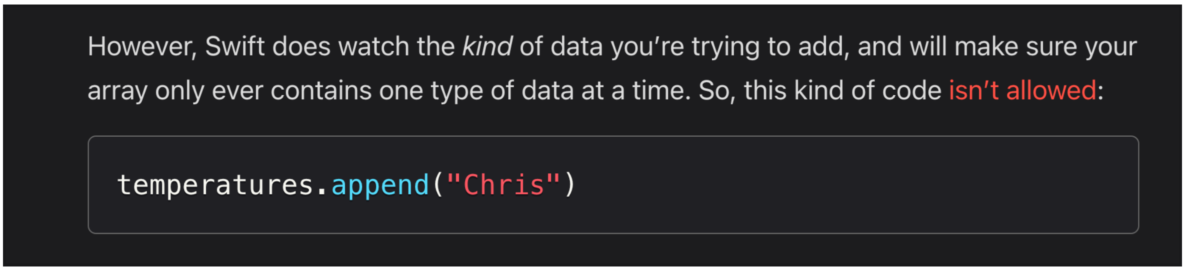


If your array is variable, you can modify it after creating it. For example, you can use `append()` to add new items:

```
beatles.append("Adrian")
```

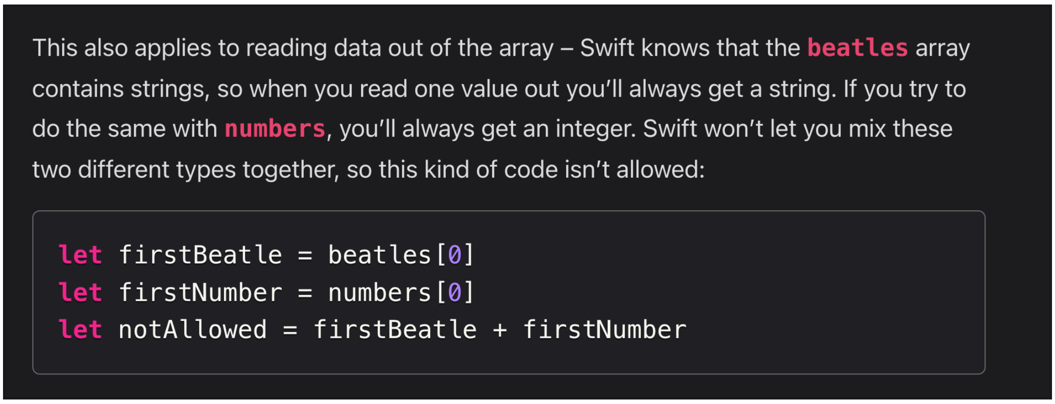
And there's nothing stopping you from adding items more than once:

```
beatles.append("Allen")
beatles.append("Adrian")
beatles.append("Novall")
beatles.append("Vivian")
```



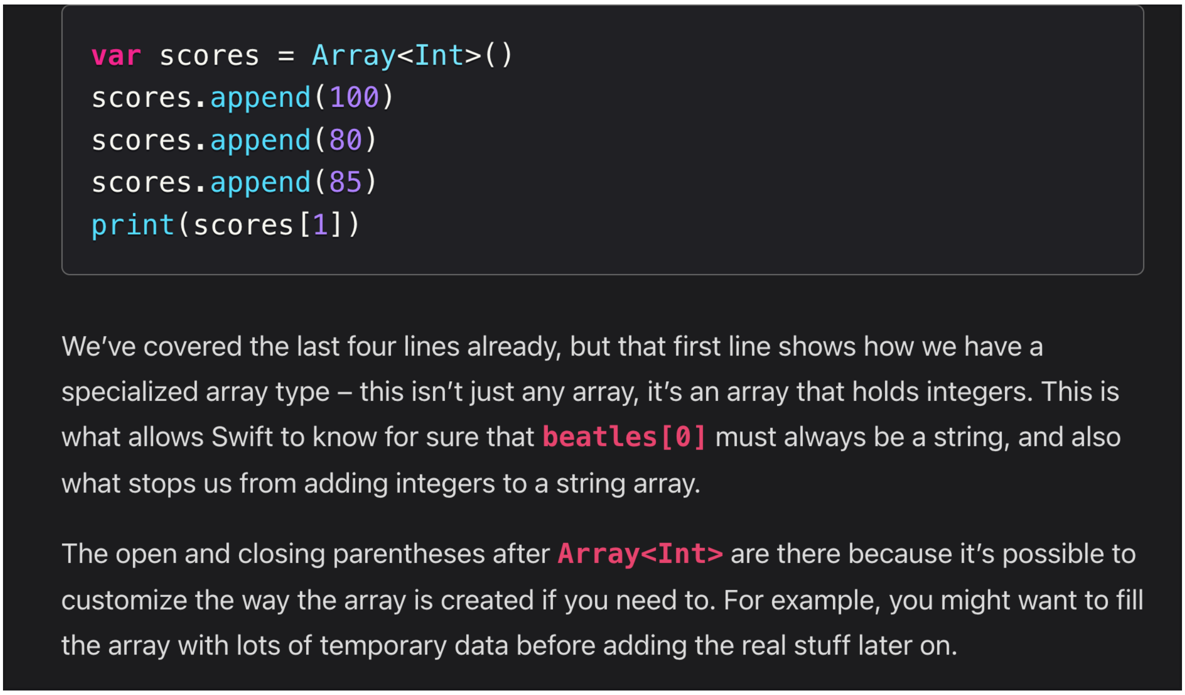
However, Swift does watch the *kind* of data you're trying to add, and will make sure your array only ever contains one type of data at a time. So, this kind of code isn't allowed:

```
temperatures.append("Chris")
```



This also applies to reading data out of the array – Swift knows that the `beatles` array contains strings, so when you read one value out you'll always get a string. If you try to do the same with `numbers`, you'll always get an integer. Swift won't let you mix these two different types together, so this kind of code isn't allowed:

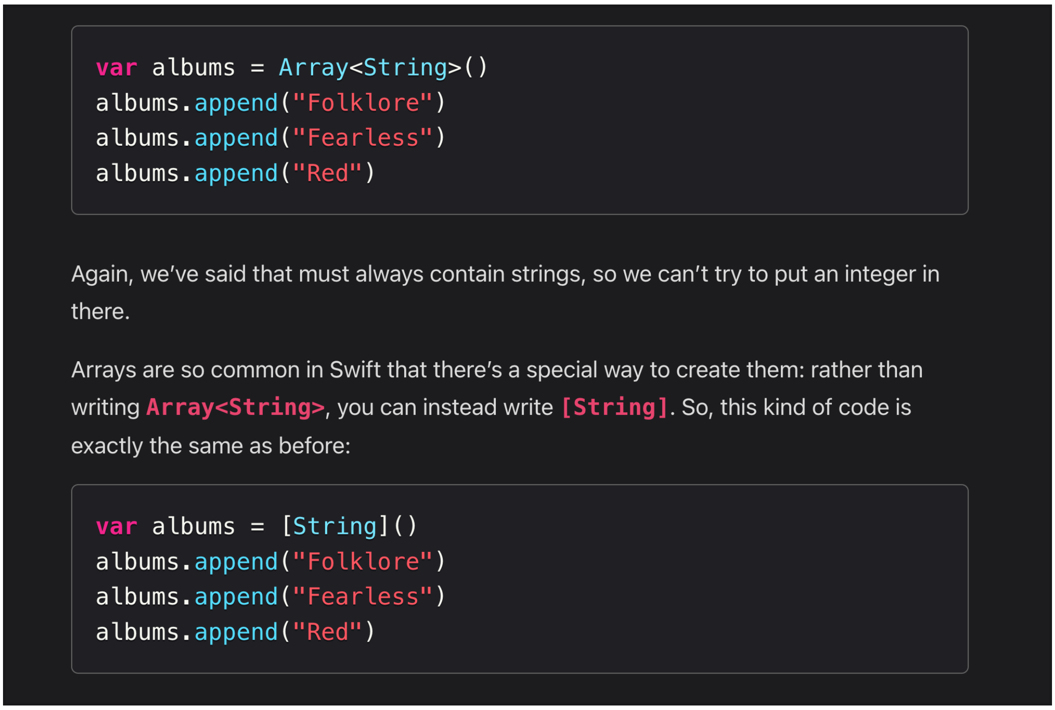
```
let firstBeatle = beatles[0]
let firstNumber = numbers[0]
let notAllowed = firstBeatle + firstNumber
```



```swift
var scores = Array<Int>()
scores.append(100)
scores.append(80)
scores.append(85)
print(scores[1])
```

We’ve covered the last four lines already, but that first line shows how we have a specialized array type – this isn’t just any array, it’s an array that holds integers. This is what allows Swift to know for sure that `beatles[0]` must always be a string, and also what stops us from adding integers to a string array.

The open and closing parentheses after `Array<Int>` are there because it’s possible to customize the way the array is created if you need to. For example, you might want to fill the array with lots of temporary data before adding the real stuff later on.

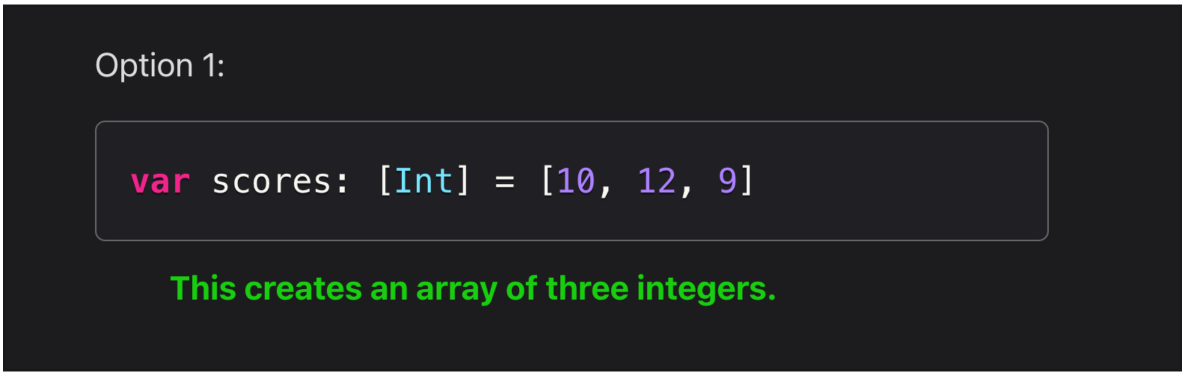


```swift
var albums = Array<String>()
albums.append("Folklore")
albums.append("Fearless")
albums.append("Red")
```

Again, we’ve said that must always contain strings, so we can’t try to put an integer in there.

Arrays are so common in Swift that there’s a special way to create them: rather than writing `Array<String>`, you can instead write `[String]`. So, this kind of code is exactly the same as before:

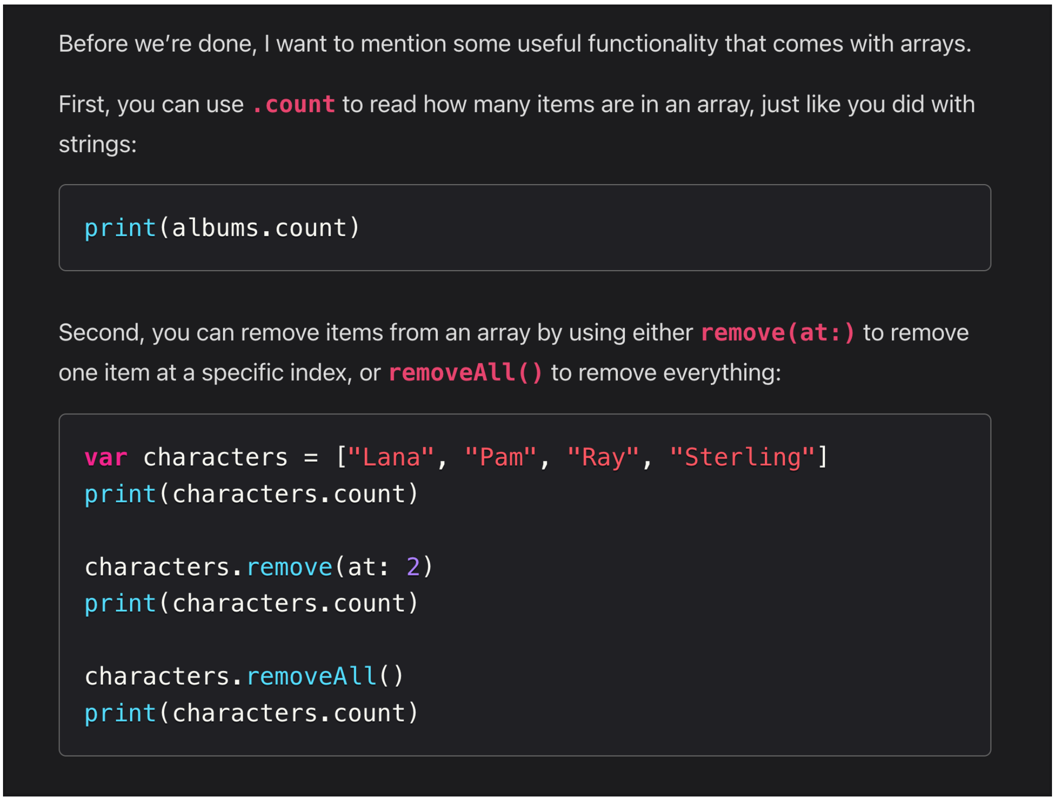
```swift
var albums = [String]()
albums.append("Folklore")
albums.append("Fearless")
albums.append("Red")
```



Option 1:

```
var scores: [Int] = [10, 12, 9]
```

This creates an array of three integers.



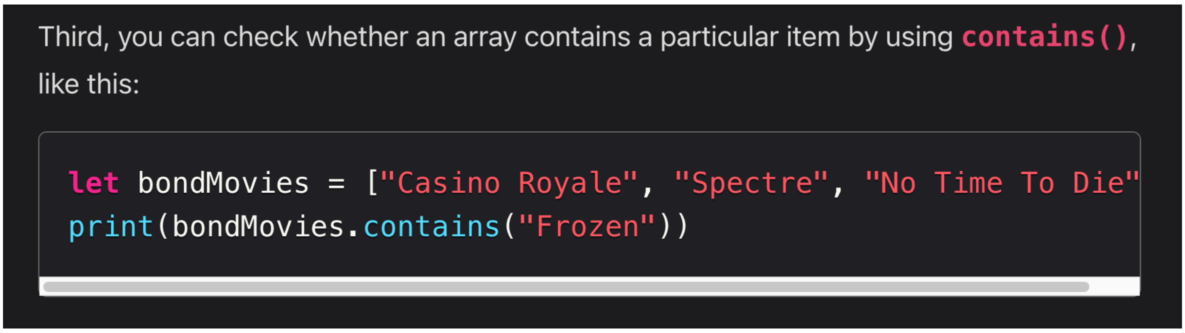
Before we’re done, I want to mention some useful functionality that comes with arrays.

First, you can use `.count` to read how many items are in an array, just like you did with strings:

```
print(albums.count)
```

Second, you can remove items from an array by using either `remove(at:)` to remove one item at a specific index, or `removeAll()` to remove everything:

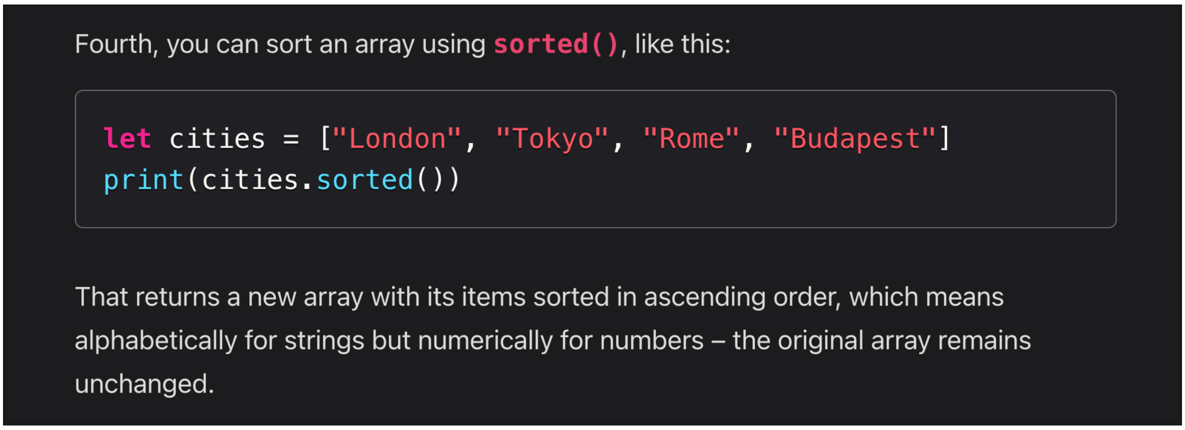
```
var characters = ["Lana", "Pam", "Ray", "Sterling"]
print(characters.count)

characters.remove(at: 2)
print(characters.count)

characters.removeAll()
print(characters.count)
```



Third, you can check whether an array contains a particular item by using **`contains()`**, like this:

```
let bondMovies = ["Casino Royale", "Spectre", "No Time To Die"
print(bondMovies.contains("Frozen"))
```

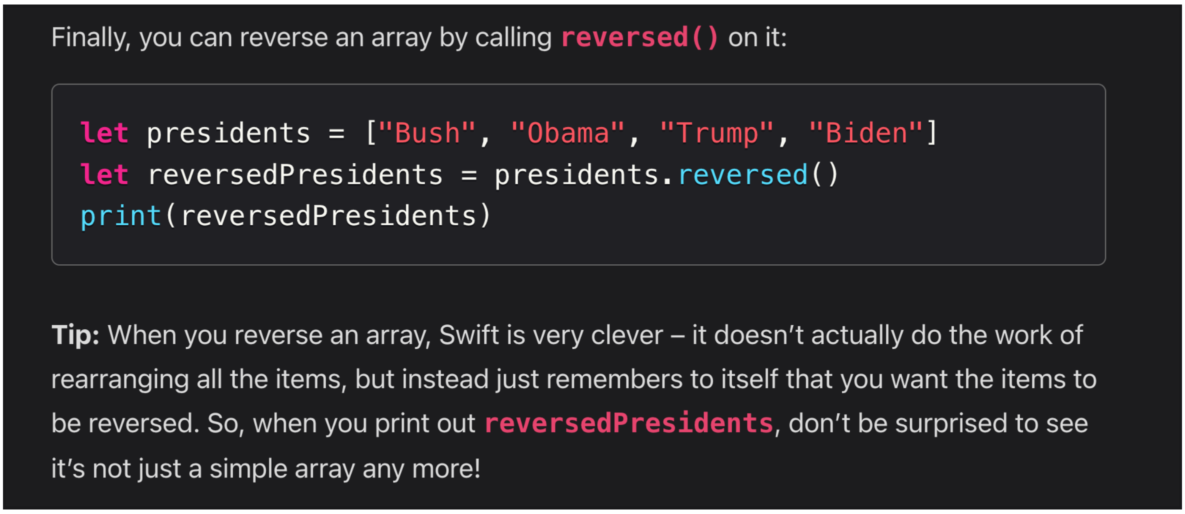
This will give truo or false .



Fourth, you can sort an array using **`sorted()`**, like this:

```
let cities = ["London", "Tokyo", "Rome", "Budapest"]
print(cities.sorted())
```

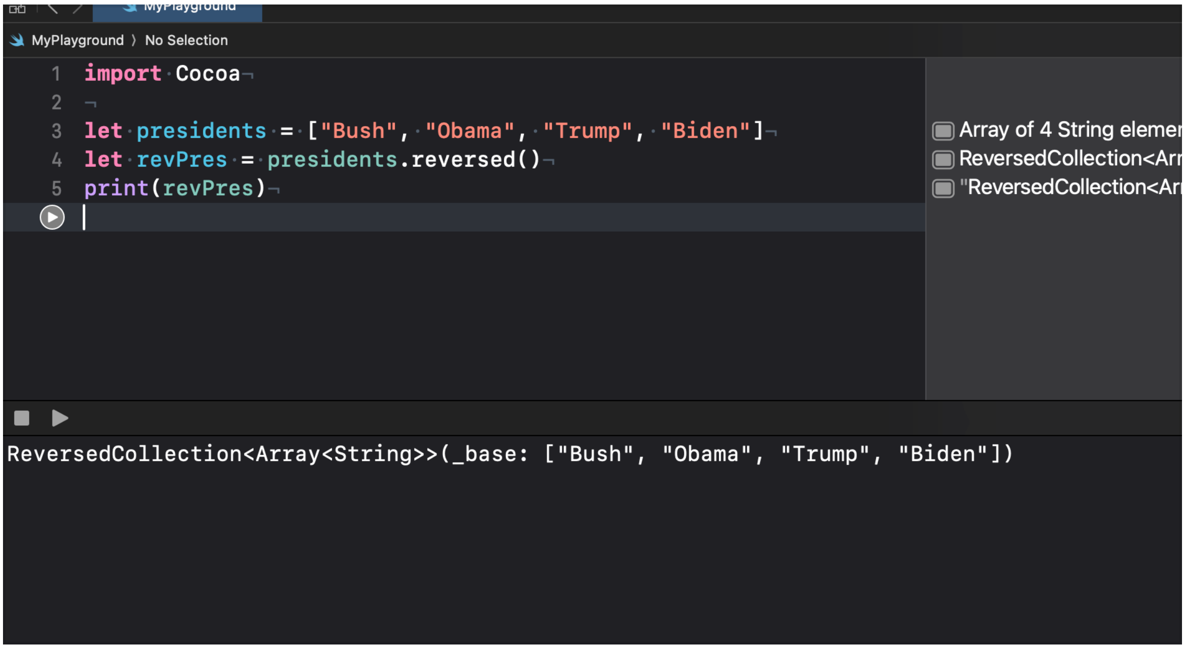
That returns a new array with its items sorted in ascending order, which means alphabetically for strings but numerically for numbers – the original array remains unchanged.



Finally, you can reverse an array by calling **`reversed()`** on it:

```
let presidents = ["Bush", "Obama", "Trump", "Biden"]
let reversedPresidents = presidents.reversed()
print(reversedPresidents)
```

Tip: When you reverse an array, Swift is very clever – it doesn't actually do the work of rearranging all the items, but instead just remembers to itself that you want the items to be reversed. So, when you print out **`reversedPresidents`**, don't be surprised to see it's not just a simple array any more!

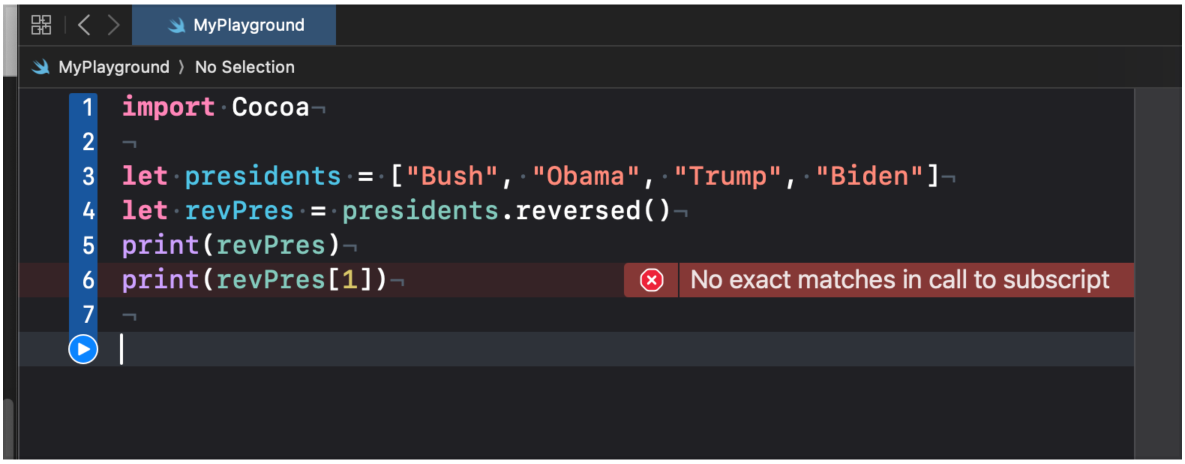


```swift
import Cocoa

let presidents = ["Bush", "Obama", "Trump", "Biden"]
let revPres = presidents.reversed()
print(revPres)
```

```
Array of 4 String elemen
ReversedCollection<Arr
"ReversedCollection<Ar
```

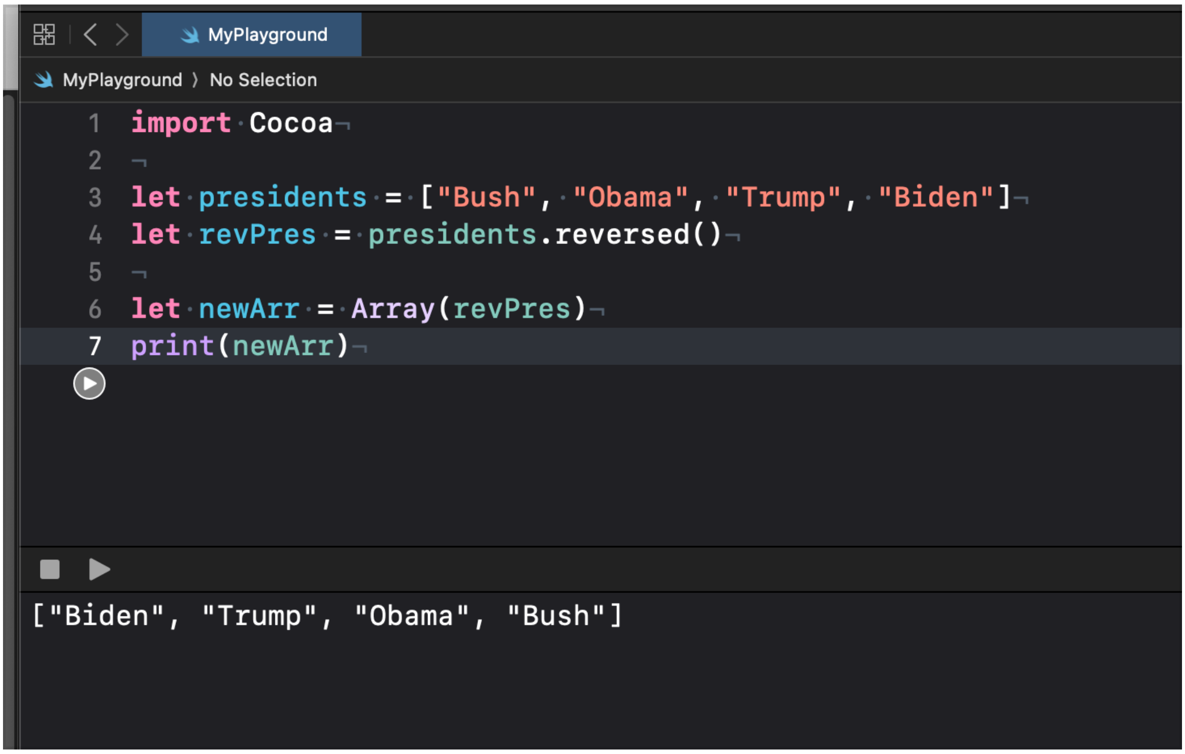
```
ReversedCollection<Array<String>>(_base: ["Bush", "Obama", "Trump", "Biden"])
```

It's remembering that array is reversed .



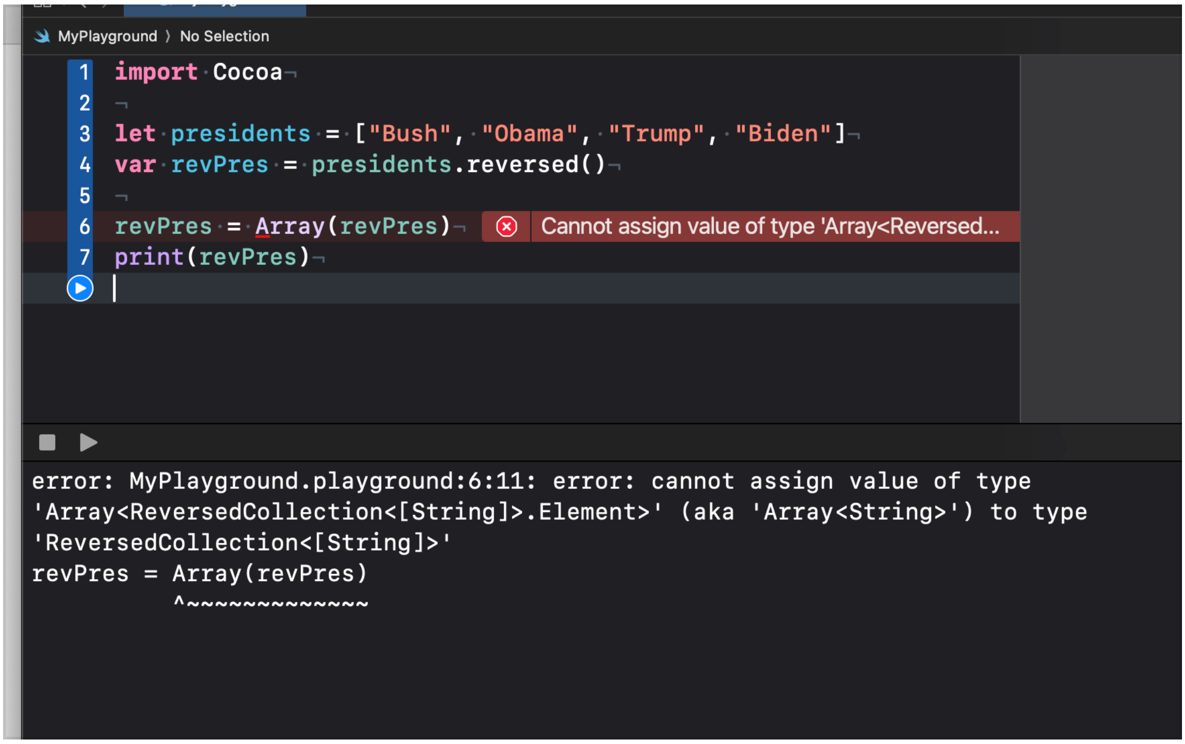
```swift
import Cocoa

let presidents = ["Bush", "Obama", "Trump", "Biden"]
let revPres = presidents.reversed()
print(revPres)
print(revPres[1])   ⊗ No exact matches in call to subscript

```

This will give error. As revPres is not an array . It's a collect ion of some sort.
We have to convert it into array.



```swift
import Cocoa

let presidents = ["Bush", "Obama", "Trump", "Biden"]
let revPres = presidents.reversed()

let newArr = Array(revPres)
print(newArr)
```

```
["Biden", "Trump", "Obama", "Bush"]
```



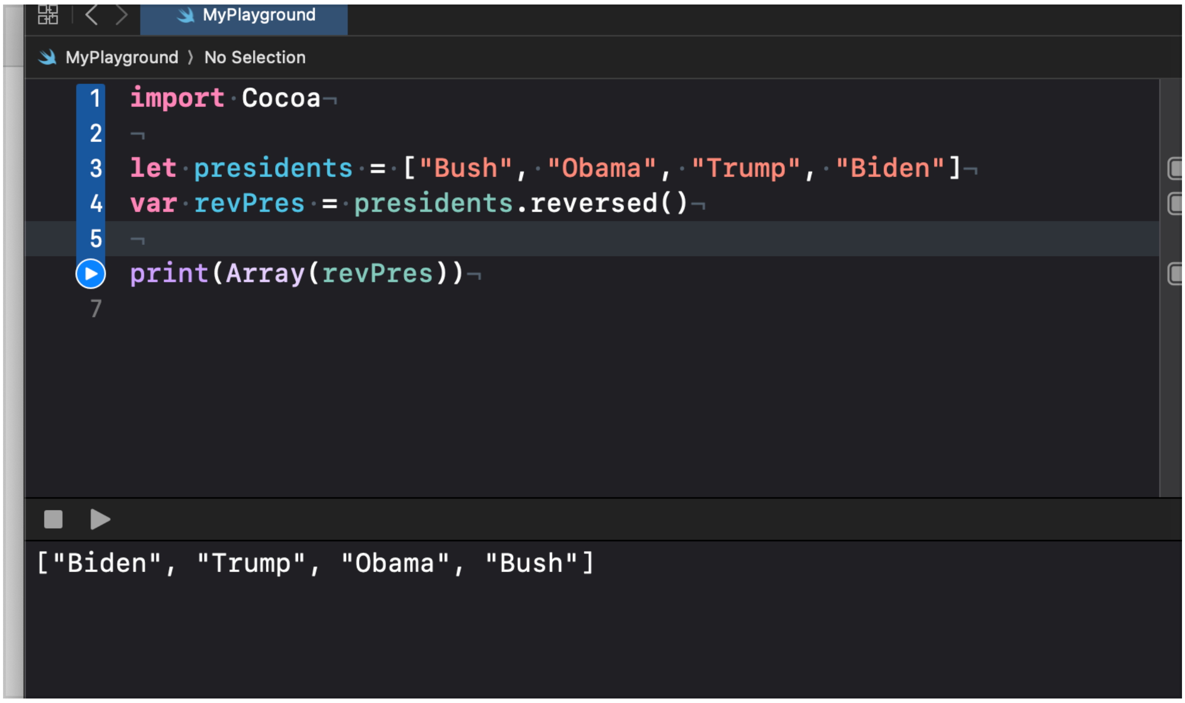
```swift
import Cocoa

let presidents = ["Bush", "Obama", "Trump", "Biden"]
var revPres = presidents.reversed()

revPres = Array(revPres)
print(revPres)
```

Cannot assign value of type 'Array<Reversed...

```
error: MyPlayground.playground:6:11: error: cannot assign value of type
'Array<ReversedCollection<[String]>.Element>' (aka 'Array<String>') to type
'ReversedCollection<[String]>'
revPres = Array(revPres)
          ^~~~~~~~~~~~~~
```

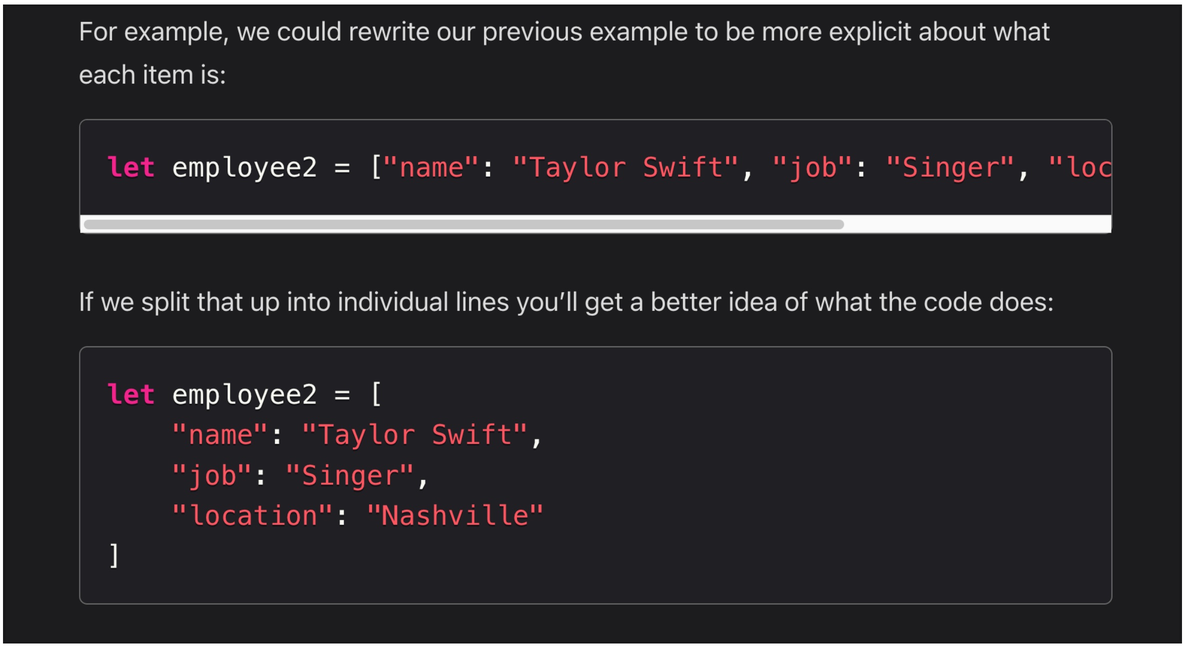
Also this is wrong as reverse is not array so we cannot convert its type.

The correct code is —>



```
import Cocoa

let presidents = ["Bush", "Obama", "Trump", "Biden"]
var revPres = presidents.reversed()

print(Array(revPres))
```

```
["Biden", "Trump", "Obama", "Bush"]
```

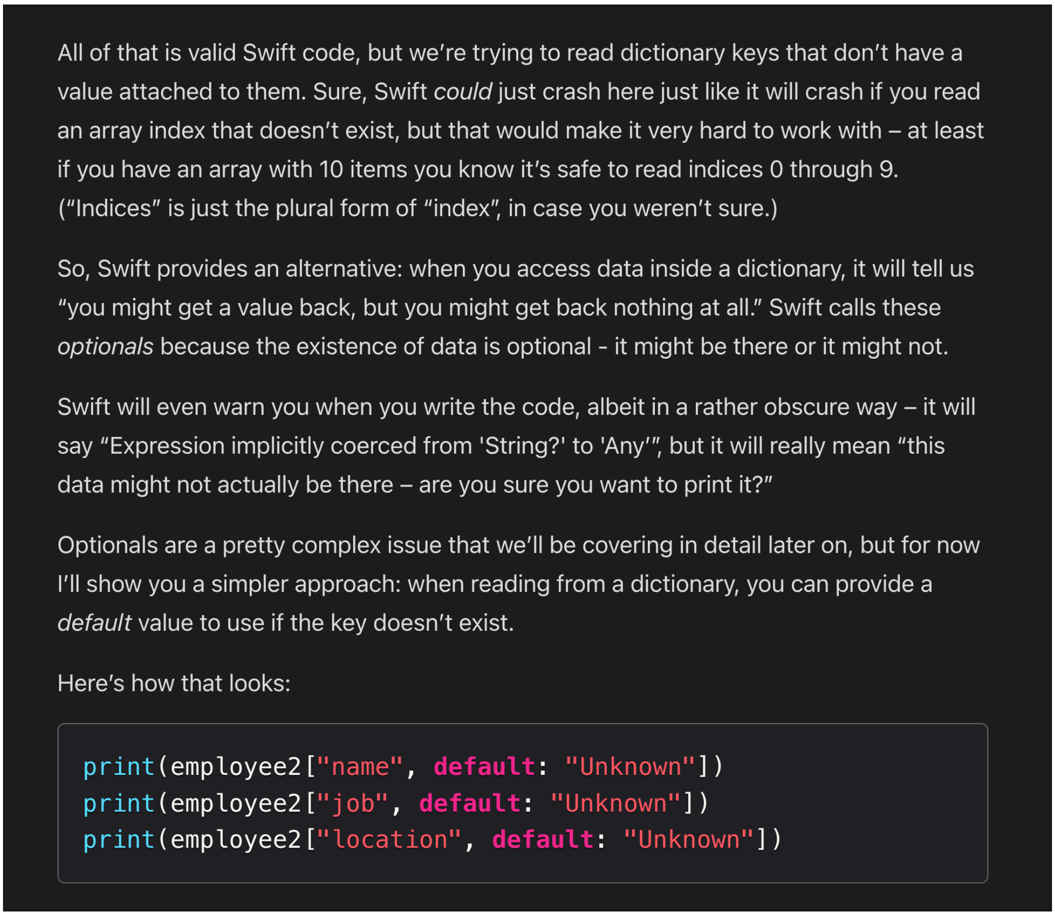


For example, we could rewrite our previous example to be more explicit about what each item is:

```
let employee2 = ["name": "Taylor Swift", "job": "Singer", "loc
```

If we split that up into individual lines you'll get a better idea of what the code does:

```
let employee2 = [
    "name": "Taylor Swift",
    "job": "Singer",
    "location": "Nashville"
]
```



All of that is valid Swift code, but we're trying to read dictionary keys that don't have a value attached to them. Sure, Swift *could* just crash here just like it will crash if you read an array index that doesn't exist, but that would make it very hard to work with – at least if you have an array with 10 items you know it's safe to read indices 0 through 9. ("Indices" is just the plural form of "index", in case you weren't sure.)

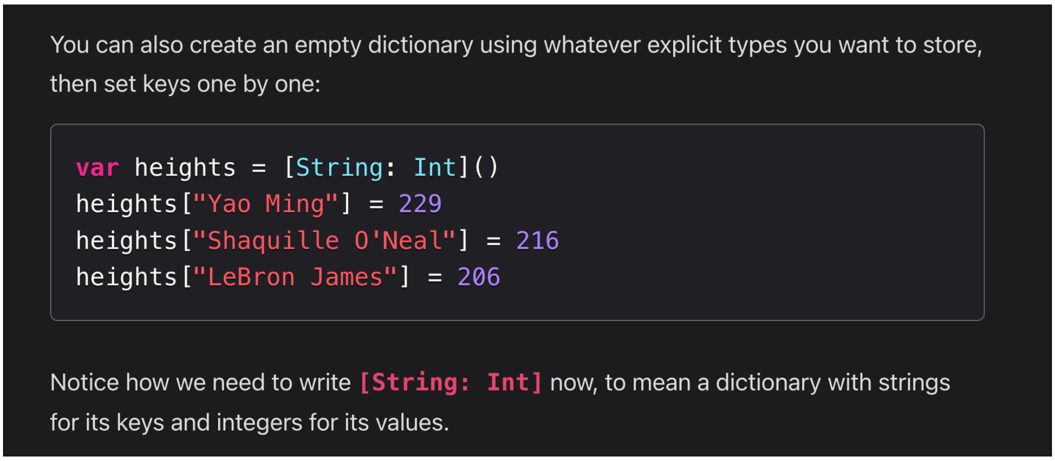
So, Swift provides an alternative: when you access data inside a dictionary, it will tell us "you might get a value back, but you might get back nothing at all." Swift calls these *optionals* because the existence of data is optional - it might be there or it might not.

Swift will even warn you when you write the code, albeit in a rather obscure way – it will say "Expression implicitly coerced from 'String?' to 'Any'", but it will really mean "this data might not actually be there – are you sure you want to print it?"

Optionals are a pretty complex issue that we'll be covering in detail later on, but for now I'll show you a simpler approach: when reading from a dictionary, you can provide a *default* value to use if the key doesn't exist.

Here's how that looks:

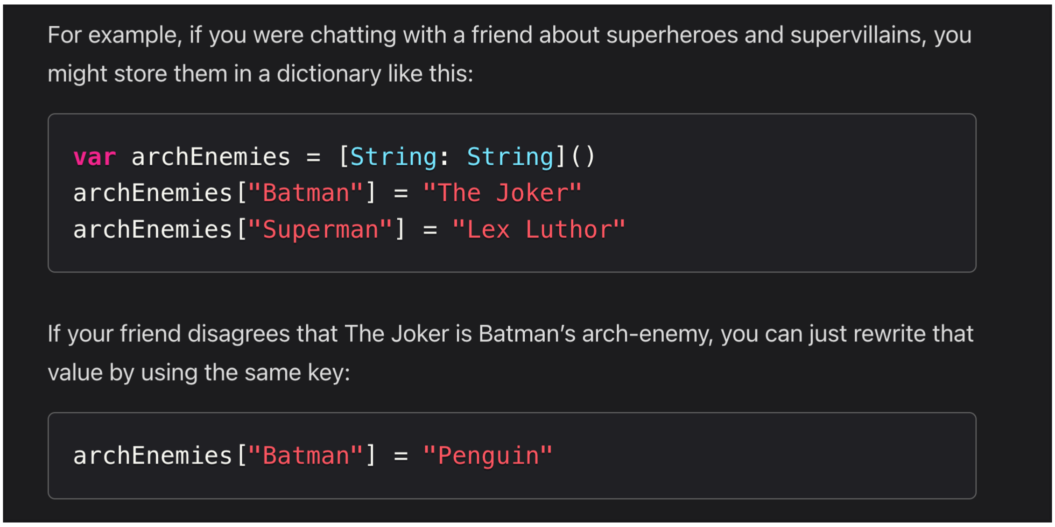
```
print(employee2["name", default: "Unknown"])
print(employee2["job", default: "Unknown"])
print(employee2["location", default: "Unknown"])
```



You can also create an empty dictionary using whatever explicit types you want to store, then set keys one by one:

```
var heights = [String: Int]()
heights["Yao Ming"] = 229
heights["Shaquille O'Neal"] = 216
heights["LeBron James"] = 206
```

Notice how we need to write `[String: Int]` now, to mean a dictionary with strings for its keys and integers for its values.



For example, if you were chatting with a friend about superheroes and supervillains, you might store them in a dictionary like this:

```
var archEnemies = [String: String]()
archEnemies["Batman"] = "The Joker"
archEnemies["Superman"] = "Lex Luthor"
```

If your friend disagrees that The Joker is Batman's arch-enemy, you can just rewrite that value by using the same key:

```
archEnemies["Batman"] = "Penguin"
```