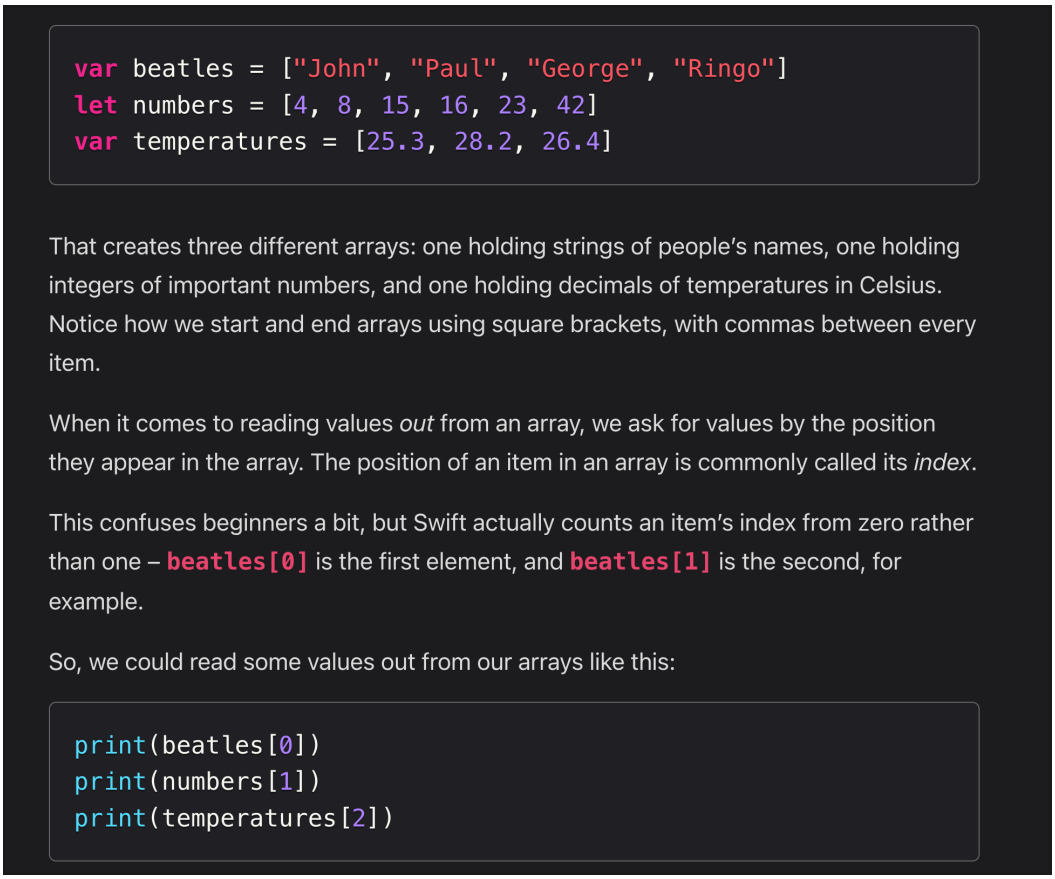
Day 3



```
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
```

```
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.

```
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:

```
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())
```

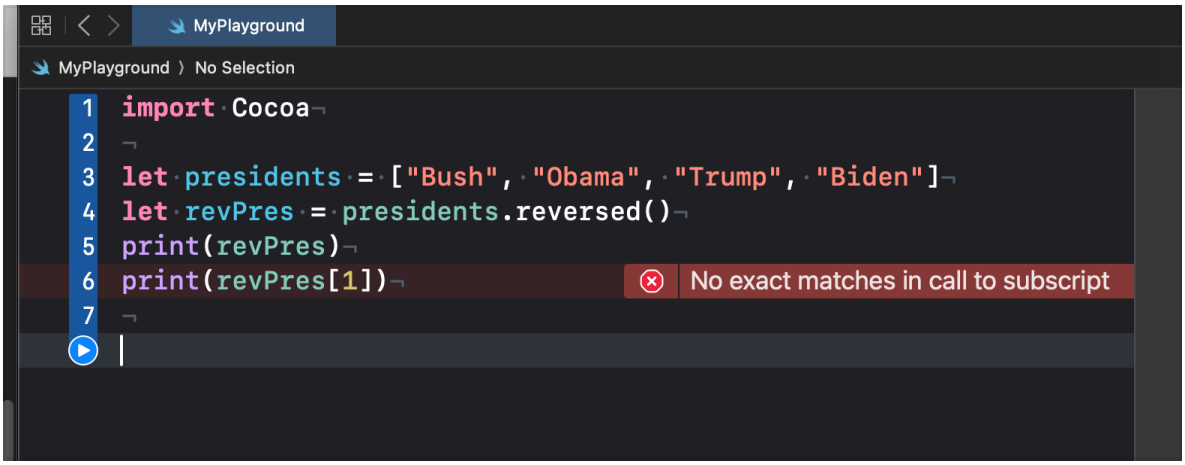
That returns a new array with its items sorted in ascending order, which means alphabetically for strings but numerically for numbers – the original array remains unchanged.

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

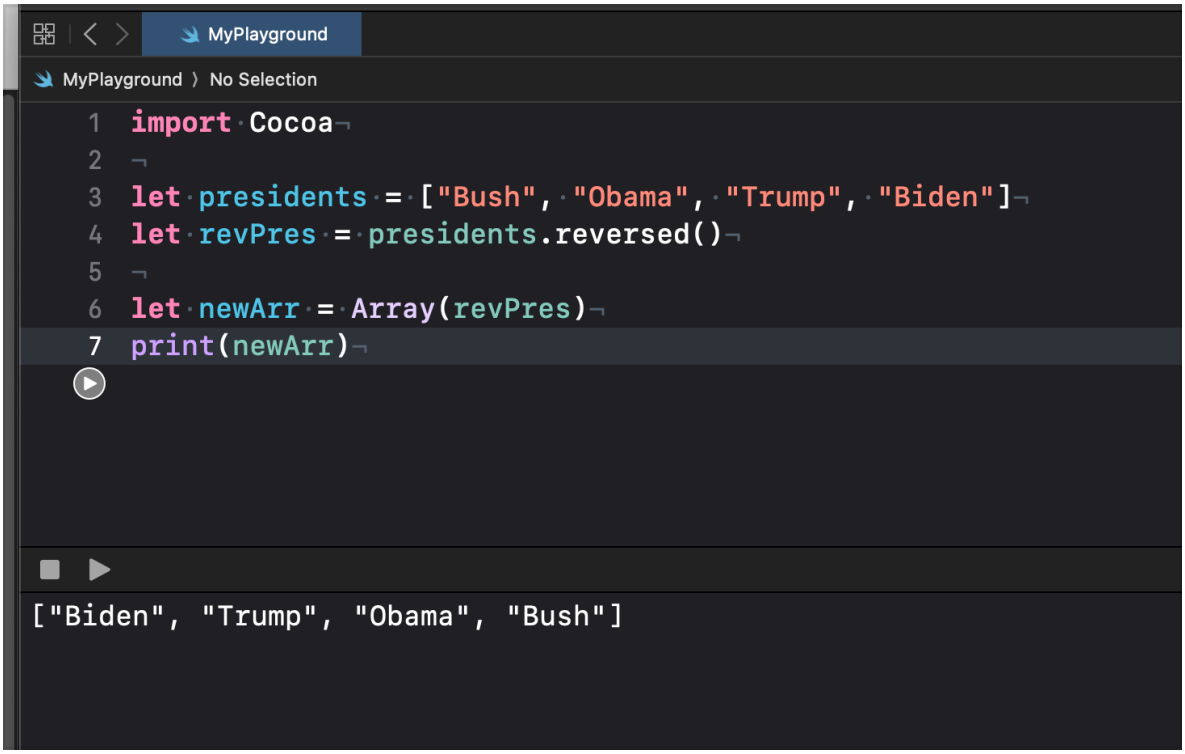
```
let presidents = ["Bush", "Obama", "Trump", "Biden"]
let reversedPresidents = presidents.reversed()
print(reversedPresidents)
```

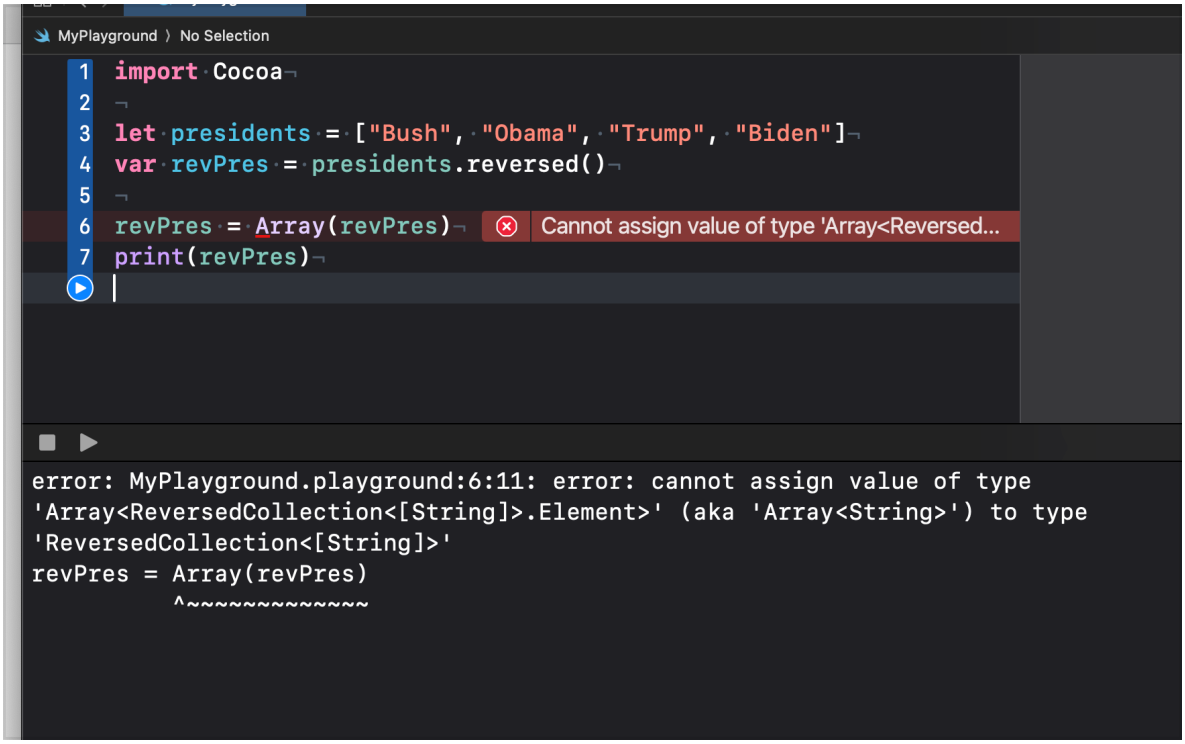
Tip: When you reverse an array, Swift is very clever – it doesn't actually do the work of rearranging all the items, but instead just remembers to itself that you want the items to be reversed. So, when you print out **`reversedPresidents`**, don't be surprised to see it's not just a simple array any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

It’s remembering that array is reversed .



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.

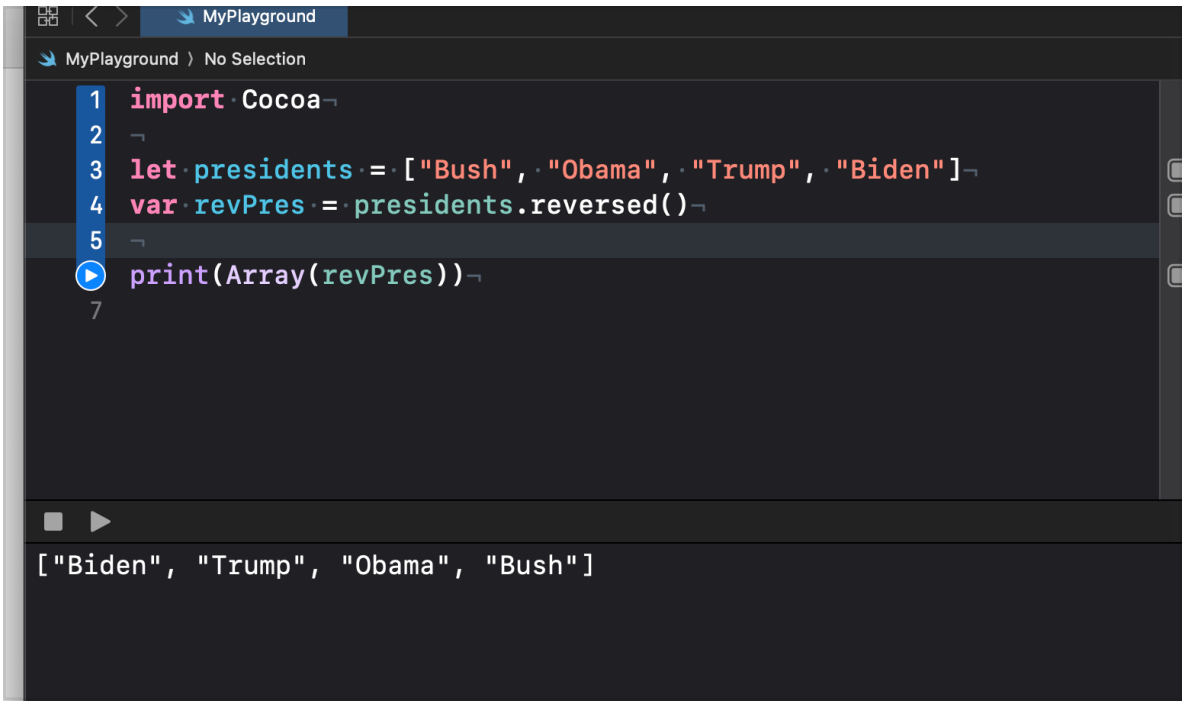




Also this is wrong as reverse is not array so we cannot convert its

type.

The correct code is —>



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"
```

For example, here's a dictionary that stores the exam results for a student:

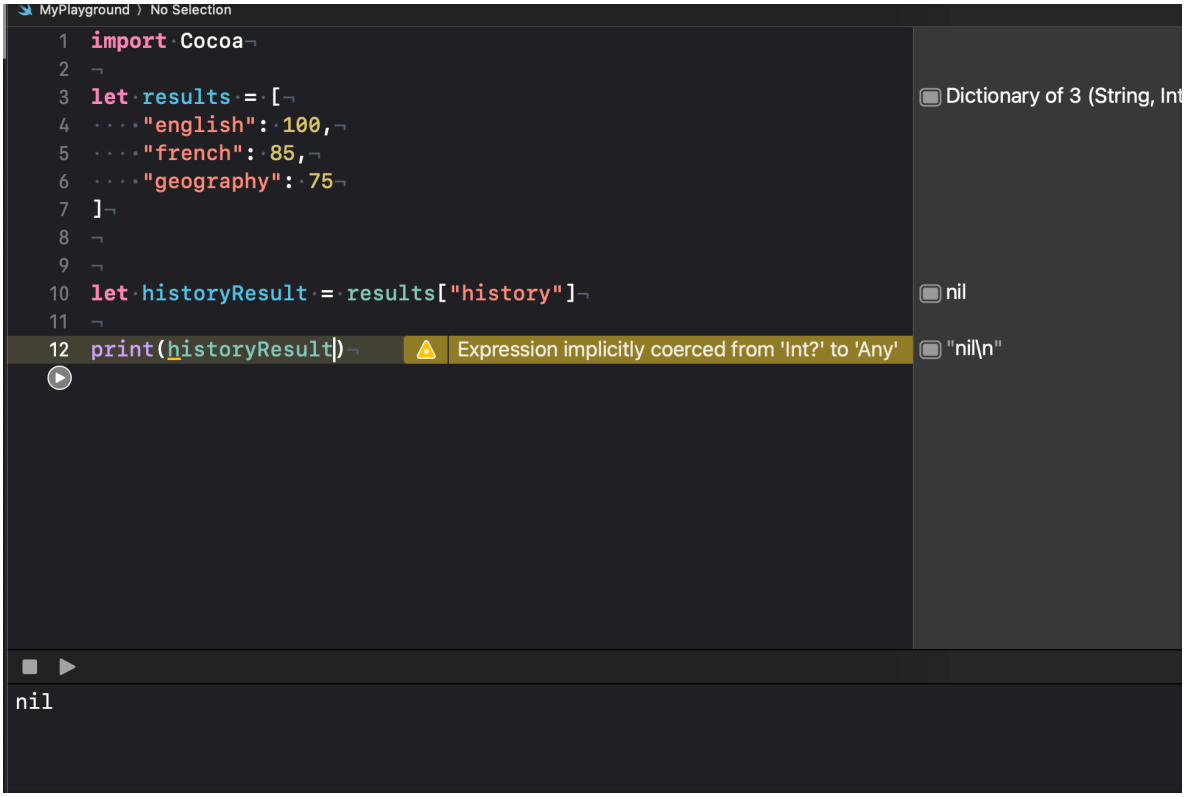
```
let results = [
    "english": 100,
    "french": 85,
    "geography": 75
]
```

As you can see, they sat three exams and scored 100%, 85%, and 75% for English, French, and Geography. If we wanted to read their history score, how we do it depends on what we want:

1. If a missing value means the student failed to take the test, then we could use a default value of 0 so that we always get an integer back.
2. If a missing value means the student has yet to take the exam, then we should skip the default value and instead look for a nil value.

So, it's not like you *always* need a default value when working with dictionaries, but when you do it's easy:

```
let historyResult = results["history", default: 0]
```



SETS

First, here's how you would make a set of actor names:

```
let people = Set(["Denzel Washington", "Tom Cruise", "Nicolas
```

> So far you've learned about two ways of collecting data in Swift: arrays and dictionaries. There is a third very common way to group data, called a *set* – they are similar to arrays, except you can't add duplicate items, and they don't store their items in a particular order.

```
var people = Set<String>()
people.insert("Denzel Washington")
people.insert("Tom Cruise")
people.insert("Nicolas Cage")
people.insert("Samuel L Jackson")
```

Notice how we're using **`insert()`**? When we had an array of strings, we added items by calling **`append()`**, but that name doesn't make sense here – we aren't adding an item to the end of the set, because the set will store the items in whatever order it wants.

Second, instead of storing your items in the exact order you specify, sets instead store them in a highly optimized order that makes it very fast to locate items. And the difference isn't small: if you have an array of 1000 movie names and use something like `contains()` to check whether it contains "The Dark Knight" Swift needs to go through every item until it finds one that matches – that might mean checking all 1000 movie names before returning false, because The Dark Knight wasn't in the array.

In comparison, calling `contains()` on a set runs so fast you'd struggle to measure it meaningfully. Heck, even if you had a million items in the set, or even 10 million items, it would still run instantly, whereas an array might take minutes or longer to do the same work.

Most of the time you'll find yourself using arrays rather than sets, but sometimes – just sometimes – you'll find that a set is exactly the right choice to solve a particular problem, and it will make otherwise slow code run in no time at all.

Tip: Alongside `contains()`, you'll also find `count` to read the number of items in a set, and `sorted()` to return a sorted array containing the the set's items.

\

Sets

Question 2/12: This will create a set with two items – true or false?

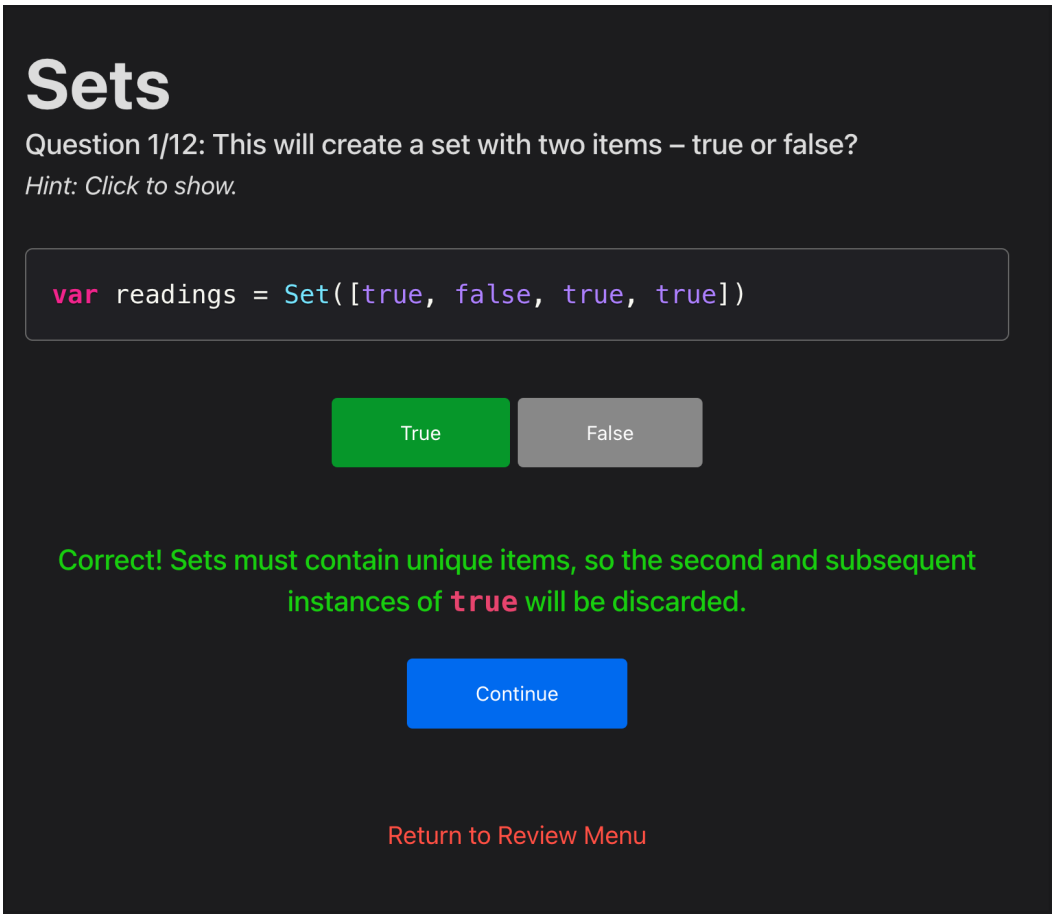
Hint: Click to show.

```
var attendees = Set([100, 100, 101, 100])
```

True | False

Correct! Sets must contain unique items, so the second and subsequent instances of 100 will be discarded.

Continue


Sets
Question 1/12: This will create a set with two items – true or false?
Hint: Click to show.
var readings = Set([true, false, true, true])
True
False
Correct! Sets must contain unique items, so the second and subsequent instances of true will be discarded.
Continue
Return to Review Menu

Enums

WWDC24 SALE: **Save 50% on all my Swift books and bundles!** >>

This is where enums come in: they let us define a new data type with a handful of specific values that it can have. Think of a Boolean, that can only have true or false – you can't set it to "maybe" or "probably", because that isn't in the range of values it understands. Enums are the same: we get to list up front the range of values it can have, and Swift will make sure you never make a mistake using them.

So, we could rewrite our weekdays into a new enum like this:

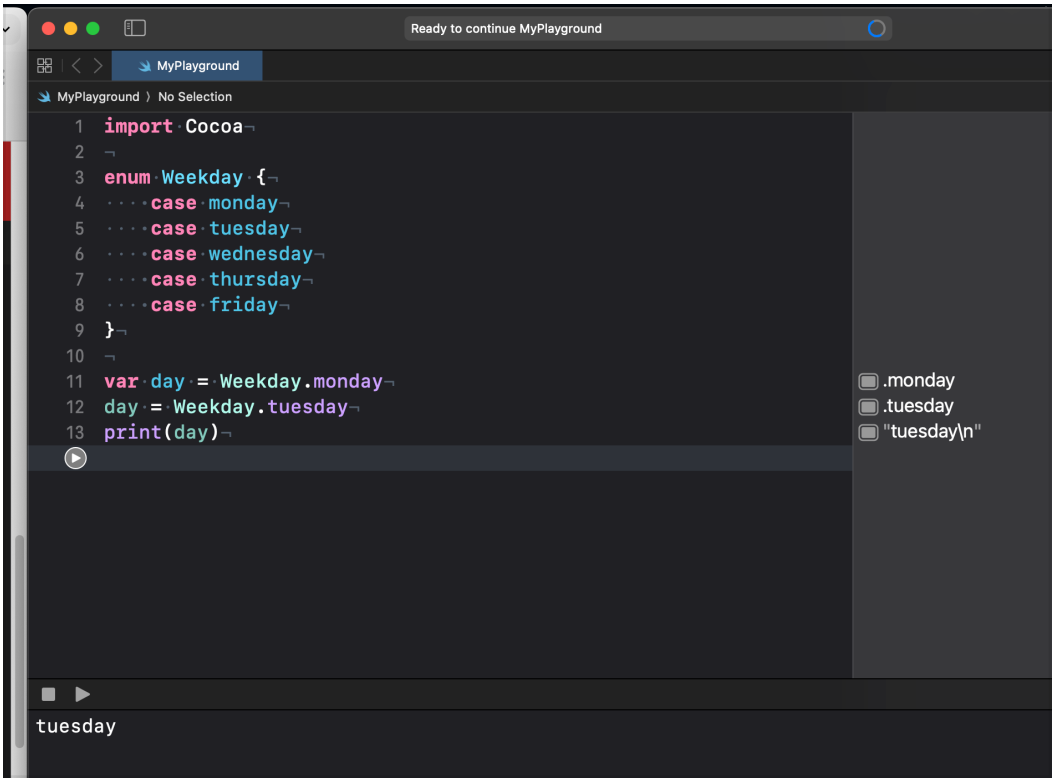
```
enum Weekday {
    case monday
    case tuesday
    case wednesday
    case thursday
    case friday
}
```

That calls the new enum `Weekday`, and provides five cases to handle the five weekdays.

Now rather than using strings, we would use the enum. Try this in your playground:

```
var day = Weekday.monday
day = Weekday.tuesday
day = Weekday.friday
```

\



Swift does two things that make enums a little easier to use. First, when you have many cases in an enum you can just write `case` once, then separate each case with a comma:

```
enum Weekday {
    case monday, tuesday, wednesday, thursday, friday
}
```

Second, remember that once you assign a value to a variable or constant, its data type becomes fixed – you can't set a variable to a string at first, then an integer later on. Well, for enums this means you can skip the enum name after the first assignment, like this:

```
var day = Weekday.monday
day = .tuesday
day = .friday
```

Swift knows that `.tuesday` must refer to `Weekday.tuesday` because `day` must always be some kind of `Weekday`.

```
var day = Weekday.monday
day = .tuesday
day = .friday
```

Swift knows that `.tuesday` must refer to `Weekday.tuesday` because `day` must always be some kind of `Weekday`.

Although it isn't visible here, one major benefit of enums is that Swift stores them in an optimized form – when we say `Weekday.monday` Swift is likely to store that using a single integer such as 0, which is much more efficient to store and check than the letters M, o, n, d, a, y.

Why does Swift need enums?

Paul Hudson @twostraws March 11th 2021

Updated for Xcode 16

Enums are an extraordinarily powerful feature of Swift, and you'll see them used in a great many ways and places. Many languages don't have enums and get by just fine, so you might wonder why Swift needs enums at all!

Well, at their simplest an enum is simply a nice name for a value. We can make an enum called `Direction` with cases for `north`, `south`, `east`, and `west`, and refer to those in our code. Sure, we could have used an integer instead, in which case we'd refer to 1, 2, 3, and 4, but could you really remember what 3 meant? And what if you typed 5 by mistake?

So, enums are a way of us saying `Direction.north` to mean something specific and safe. If we had written `Direction.thatWay` and no such case existed, Swift would simply refuse to build our code – it doesn't understand the enum case. Behind the scenes, Swift can store its enum values very simply, so they are much faster to create and store than something like a string.

As you progress, you'll learn how Swift lets us add more functionality to enums – they are more powerful in Swift than in any other language I have seen.