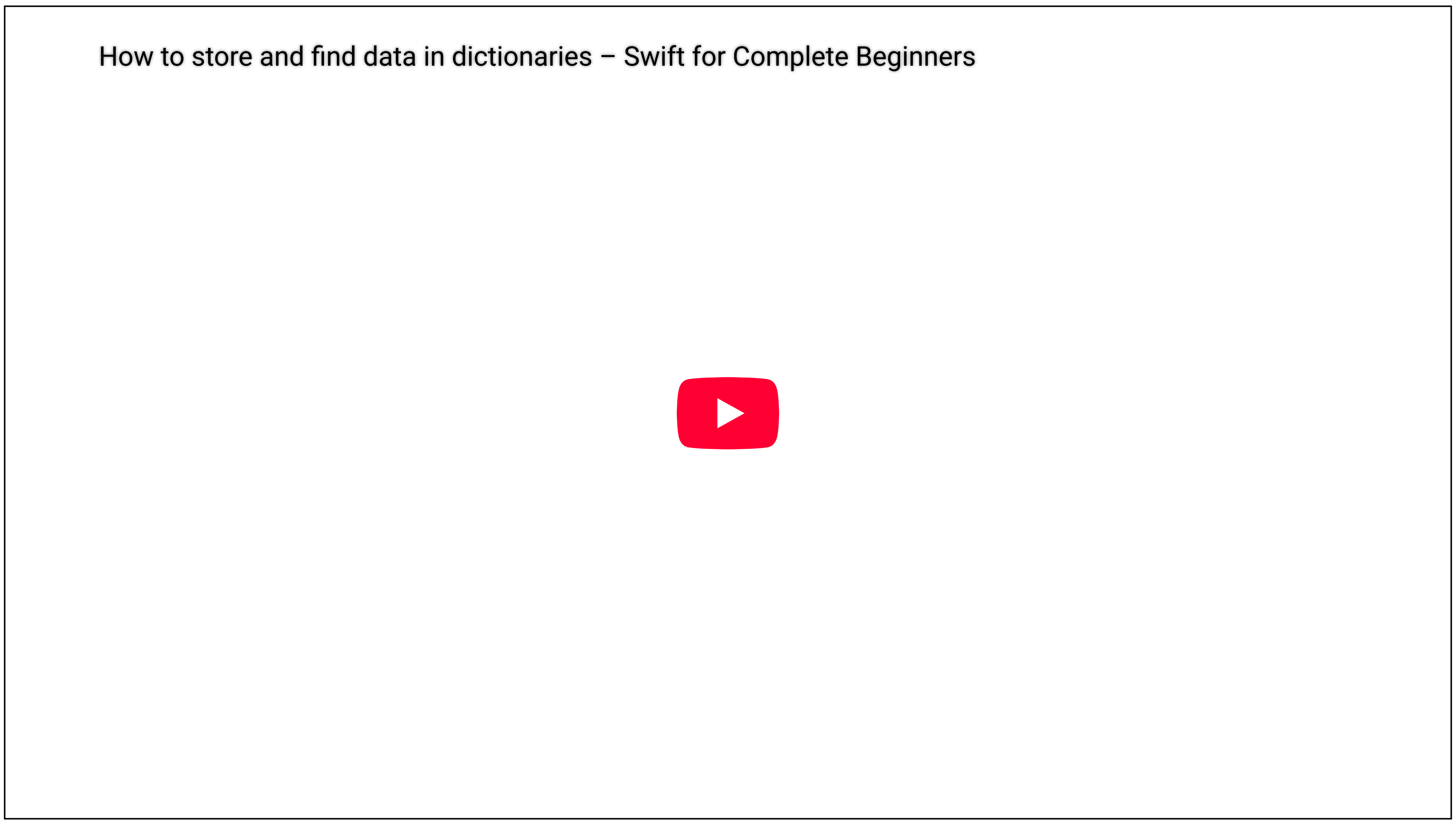
< How to store ordered data in arrays How to use sets for fast data lookup >

How to store and find data in dictionaries

Paul Hudson @twostraws October 25th 2021

Updated for Xcode 16.4


How to store and find data in dictionaries – Swift for Complete Beginners

You've seen how arrays are a great way to store data that has a particular order, such as days of the week or temperatures for a city. Arrays are a great choice when items should be stored in the order you add them, or when you might have duplicate items in there, but very often accessing data by its position in the array can be annoying or even dangerous.

For example, here's an array containing an employee's details:

```swift
var employee = ["Taylor Swift", "Singer", "Nashville"]
```

I've told you that the data is about an employee, so you might be able to guess what various parts do:

```swift
print("Name: \(employee[0])")
print("Job title: \(employee[1])")
print("Location: \(employee[2])")
```

But that has a couple of problems. First, you can't really be sure that `employee[2]` is their location – maybe that's their password. Second, there's no guarantee that item 2 is even there, particularly because we made the array a variable. This kind of code would cause serious problems:

```
print("Name: \(employee[0])")
employee.remove(at: 1)
print("Job title: \(employee[1])")
print("Location: \(employee[2])")
```

That now prints Nashville as the job title, which is wrong, and will cause our code to crash when it reads **`employee[2]`**, which is just *bad*.

Swift has a solution for both these problems, called *dictionaries*. Dictionaries don’t store items according to their position like arrays do, but instead let *us* decide where items should be stored.

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

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

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"
]
```

As you can see, we’re now being really clear: the name is Taylor Swift, the job is Singer, and the location is Nashville. Swift calls the strings on the left – name, job, and location – the *keys* to the dictionary, and the strings on the right are the *values*.

When it comes to reading data out from the dictionary, you use the same keys you used when creating it:

```
print(employee2["name"])
print(employee2["job"])
print(employee2["location"])
```

If you try that in a playground, you’ll see Xcode throws up various warnings along the lines of “Expression implicitly coerced from 'String?' to 'Any'”. Worse, if you look at the output from your playground you’ll see it prints **`Optional("Taylor Swift")`** rather than just **`Taylor Swift`** – what gives?

Well, think about this:

```
print(employee2["password"])
print(employee2["status"])
print(employee2["manager"])
```

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"])
```

All the examples have used strings for both the keys and values, but you can use other data types for either of them. For example, we could track which students have graduated from school using strings for names and Booleans for their graduation status:

```
let hasGraduated = [
    "Eric": false,
    "Maeve": true,
    "Otis": false,
]
```

Or we could track years when Olympics took place along with their locations:

```
let olympics = [
    2012: "London",
    2016: "Rio de Janeiro",
    2021: "Tokyo"
]

print(olympics[2012, 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.

Because each dictionary item must exist at one specific key, dictionaries don't allow duplicate keys to exist. Instead, if you set a value for a key that already exists, Swift will overwrite whatever was the previous value.

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"
```

Finally, just like arrays and the other data types we've seen so far, dictionaries come with some useful functionality that you'll want to use in the future – `count` and `removeAll()` both exists for dictionaries, and work just like they do for arrays.

< How to store ordered data in arrays

How to use sets for fast data lookup >

Was this page useful? Let us know!

☆☆☆☆☆

Average rating: 4.7/5

Click here to visit the Hacking with Swift store >>

Twitter

Mastodon

Email

Sponsor the site

About Glossary Code License Privacy Policy Refund Policy Update Policy Code of Conduct

?

You are not logged in

Log in or create account