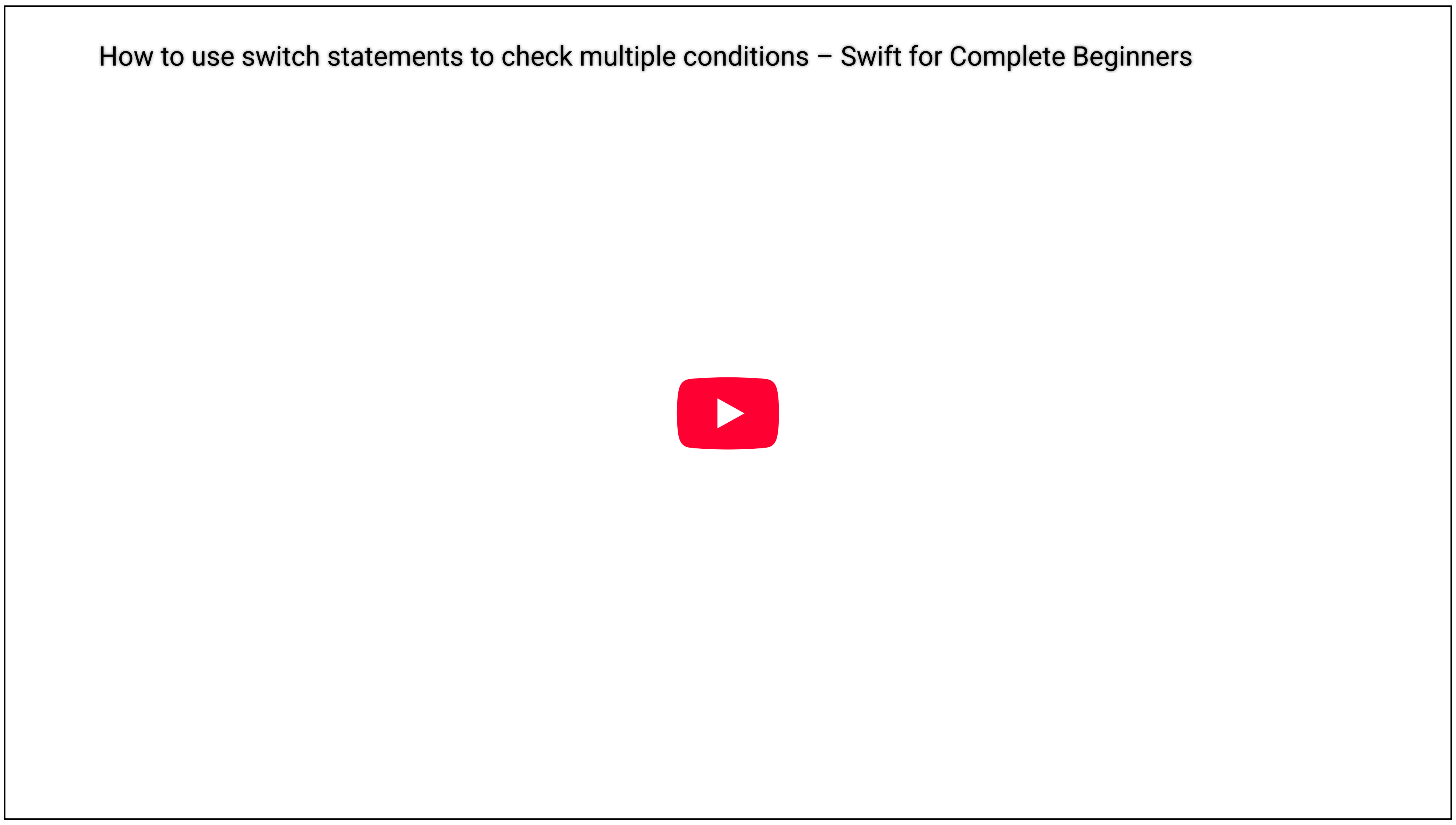
< How to check multiple conditions

How to use the ternary conditional operator for quick tests >

# How to use switch statements to check multiple conditions

Paul Hudson @twostraws April 11th 2024

*Updated for Xcode 16.4*


How to use switch statements to check multiple conditions – Swift for Complete Beginners

You can use **`if`** and **`else if`** repeatedly to check conditions as many times as you want, but it gets a bit hard to read. For example, if we had a weather forecast from an enum we could choose which message to print based on a series of conditions, like this:

```swift
enum Weather {
    case sun, rain, wind, snow, unknown
}

let forecast = Weather.sun

if forecast == .sun {
    print("It should be a nice day.")
} else if forecast == .rain {
    print("Pack an umbrella.")
} else if forecast == .wind {
    print("Wear something warm")
} else if forecast == .rain {
    print("School is cancelled.")
} else {
    print("Our forecast generator is broken!")
}
```

That works, but it has problems:

1. We keep having to write `forecast`, even though we're checking the same thing each time.
2. I accidentally checked `.rain` twice, even though the second check can never be true because the second check is only performed if the first check failed.
3. I didn't check `.snow` at all, so we're missing functionality.

We can solve all three of those problems using a different way of checking conditions called `switch`. This also lets us check individual cases one by one, but now Swift is able to help out. In the case of an enum, it knows all possible cases the enum can have, so if we miss one or check one twice it will complain.

So, we can replace all those `if` and `else if` checks with this:

```swift
switch forecast {
case .sun:
    print("It should be a nice day.")
case .rain:
    print("Pack an umbrella.")
case .wind:
    print("Wear something warm")
case .snow:
    print("School is cancelled.")
case .unknown:
    print("Our forecast generator is broken!")
}
```

Let's break that down:

1. We start with `switch forecast`, which tells Swift that's the value we want to check.
2. We then have a string of `case` statements, each of which are values we want to compare against `forecast`.
3. Each of our cases lists one weather type, and because we're switching on `forecast` we don't need to write `Weather.sun`, `Weather.rain` and so on – Swift knows it must be some kind of `Weather`.
4. After each case, we write a colon to mark the start of the code to run if that case is matched.

5. We use a closing brace to end the `switch` statement.

If you try changing `.snow` for `.rain`, you'll see Swift complains loudly: once that we've checked `.rain` twice, and again that our `switch` statement is not *exhaustive* – that it doesn't handle all possible cases.

If you've ever used other programming languages, you might have noticed that Swift's `switch` statement is different in two places:

1. All `switch` statements *must* be exhaustive, meaning that all possible values must be handled in there so you can't leave one off by accident.
2. Swift will execute the first case that matches the condition you're checking, but no more. Other languages often carry on executing other code from all subsequent cases, which is usually entirely the wrong default thing to do.

Although both those statements are true, Swift gives us a little more control if we need it.

First, *yes* all `switch` statements *must* be exhaustive: you must ensure all possible values are covered. If you're switching on a string then clearly it's not possible to make an exhaustive check of all possible strings because there is an infinite number, so instead we need to provide a *default* case – code to run if none of the other cases match.

For example, we could switch over a string containing a place name:

```
let place = "Metropolis"

switch place {
case "Gotham":
    print("You're Batman!")
case "Mega-City One":
    print("You're Judge Dredd!")
case "Wakanda":
    print("You're Black Panther!")
default:
    print("Who are you?")
}
```

That `default:` at the end is the default case, which will be run if all cases have failed to match.

**Remember: Swift checks its cases in order and runs the first one that matches.** If you place `default` before any other case, that case is useless because it will never be matched and Swift will refuse to build your code.

Second, if you explicitly *want* Swift to carry on executing subsequent cases, use `fallthrough`. This is *not* commonly used, but sometimes – just sometimes – it can help you avoid repeating work.

For example, there's a famous Christmas song called The Twelve Days of Christmas, and as the song goes on more and more gifts are heaped on an unfortunate person who by about day six has a rather full house.

We could make a simple approximation of this song using `fallthrough`. First, here's how the code would look *without* `fallthrough`:

```swift
let day = 5
print("My true love gave to me…")

switch day {
case 5:
    print("5 golden rings")
case 4:
    print("4 calling birds")
case 3:
    print("3 French hens")
case 2:
    print("2 turtle doves")
default:
    print("A partridge in a pear tree")
}
```

That will print “5 golden rings”, which isn’t quite right. On day 1 only “A partridge in a pear tree” should be printed, on day 2 it should be “2 turtle doves” *then* “A partridge in a pear tree”, on day 3 it should be “3 French hens”, “2 turtle doves”, and… well, you get the idea.

We can use `fallthrough` to get exactly that behavior:

```swift
let day = 5
print("My true love gave to me…")

switch day {
case 5:
    print("5 golden rings")
    fallthrough
case 4:
    print("4 calling birds")
    fallthrough
case 3:
    print("3 French hens")
    fallthrough
case 2:
    print("2 turtle doves")
    fallthrough
default:
    print("A partridge in a pear tree")
}
```

That will match the first case and print “5 golden rings”, but the `fallthrough` line means `case 4` will execute and print “4 calling birds”, which in turn uses `fallthrough` again so that “3 French hens” is printed, and so on. It’s not a perfect match to the song, but at least you can see the functionality in action!

< How to check multiple conditions

How to use the ternary conditional operator for quick tests >

Was this page useful? Let us know!

☆☆☆☆☆

Average rating: 4.6/5

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