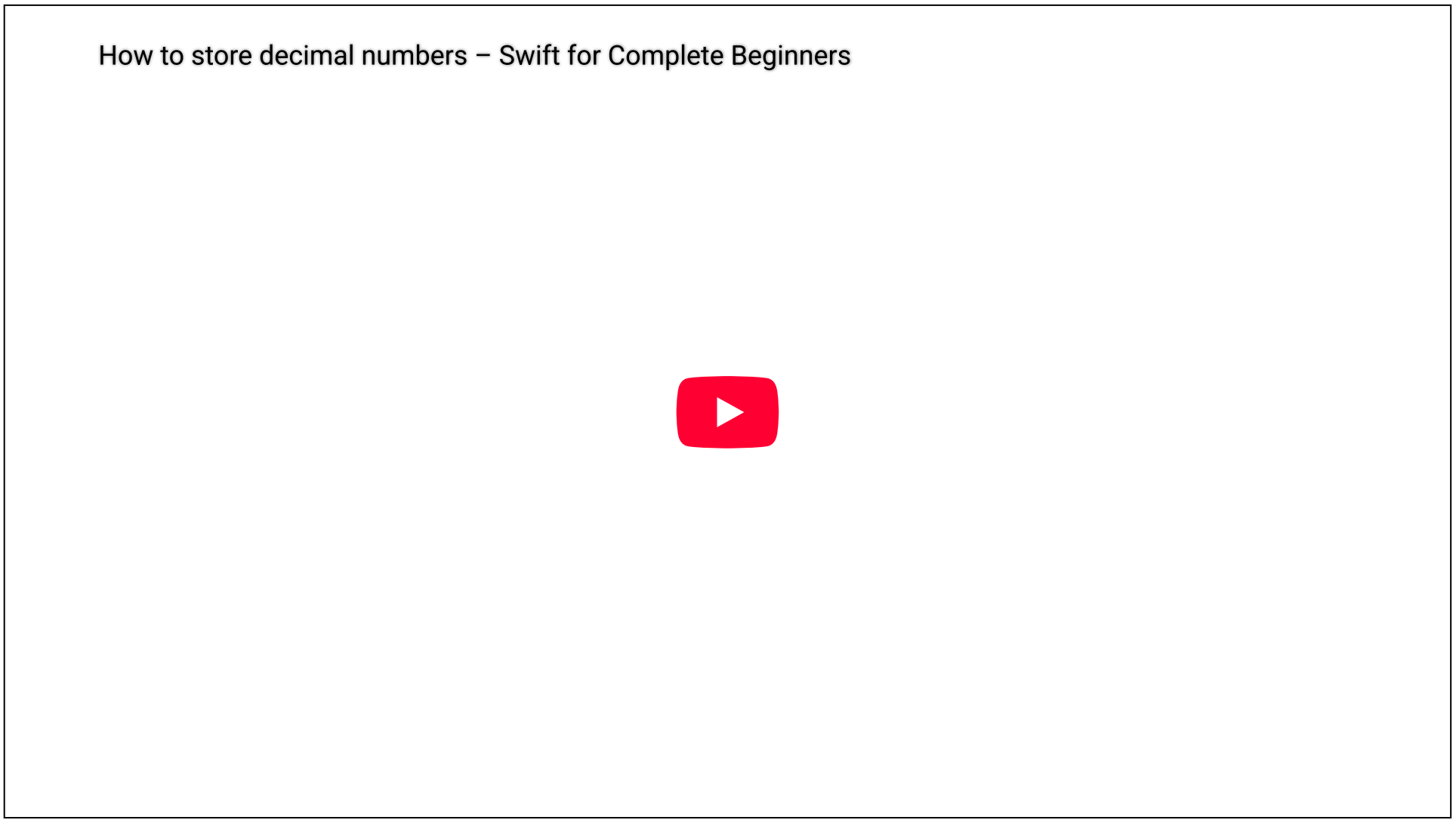
< How to store whole numbers

How to store truth with Booleans >

How to store decimal numbers

Paul Hudson @twostraws October 1st 2021

Updated for Xcode 16.4



When you're working with decimal numbers such as 3.1, 5.56, or 3.141592654, you're working with what Swift calls *floating-point* numbers. The name comes from the surprisingly complex way the numbers are stored by your computer: it tries to store very large numbers such as 123,456,789 in the same amount of space as very small numbers such as 0.0000000001, and the only way it can do that is by moving the decimal point around based on the size of the number.

This storage method causes decimal numbers to be notoriously problematic for programmers, and you can get a taste of this with just two lines of Swift code:

```
let number = 0.1 + 0.2
print(number)
```

When that runs it won't print 0.3. Instead, it will print 0.30000000000000004 – that 0.3, then 15 zeroes, then a 4 because… well, like I said, it's complex.

I'll explain more *why* it's complex in a moment, but first let's focus on what matters.

First, when you create a floating-point number, Swift considers it to be a **`Double`**. That's short for "double-precision floating-point number", which I realize is quite a strange name – the way we've handled floating-point numbers has changed a lot over the years, and although Swift does a good job of simplifying this you might sometimes meet some older code that is more complex. In this case, it means Swift allocates twice the amount of storage as some older languages would do, meaning a **`Double`** can store absolutely massive numbers.

Second, Swift considers decimals to be a wholly different type of data to integers, which means you can't mix them together. After all, integers are always 100% accurate, whereas decimals are not, so Swift won't let you put the two of them together unless you specifically ask for it to happen.

In practice, this means you can't do things like adding an integer to a decimal, so this kind of code will produce an error:

```
let a = 1
let b = 2.0
let c = a + b
```

Yes, *we* can see that **b** is really just the integer 2 masquerading as a decimal, but Swift still won't allow that code to run. This is called *type safety*: Swift won't let us mix different types of data by accident.

If you want that to happen you need to tell Swift explicitly that it should either treat the **Double** inside **b** as an **Int**:

```
let c = a + Int(b)
```

Or treat the **Int** inside **a** as a **Double**:

```
let c = Double(a) + b
```

Third, Swift decides whether you wanted to create a **Double** or an **Int** based on the number you provide – if there's a dot in there, you have a **Double**, otherwise it's an **Int**. Yes, even if the numbers after the dot are 0.

So:

```
let double1 = 3.1
let double2 = 3131.3131
let double3 = 3.0
let int1 = 3
```

Combined with type safety, this means that once Swift has decided what data type a constant or variable holds, it must always hold that same data type. That means this code is fine:

```
var name = "Nicolas Cage"
name = "John Travolta"
```

But this kind of code is not:

```
var name = "Nicolas Cage"
name = 57
```

That tells Swift **name** will store a string, but then it tries to put an integer in there instead.

Finally, decimal numbers have the same range of operators and compound assignment operators as integers:

```
var rating = 5.0
rating *= 2
```

Many older APIs use a slightly different way of storing decimal numbers, called `CGFloat`. Fortunately, Swift lets us use regular `Double` numbers everywhere a `CGFloat` is expected, so although you will see `CGFloat` appear from time to time you can just ignore it.

In case you were curious, the reason floating-point numbers are complex is because computers are trying to use binary to store complicated numbers. For example, if you divide 1 by 3 we know you get 1/3, but that can't be stored in binary so the system is designed to create very close approximations. It's extremely efficient, and the error is so small it's usually irrelevant, but at least you know why Swift doesn't let us mix `Int` and `Double` by accident!

< How to store whole numbers

How to store truth with Booleans >

Was this page useful? Let us know!

☆☆☆☆☆

Average rating: 4.8/5