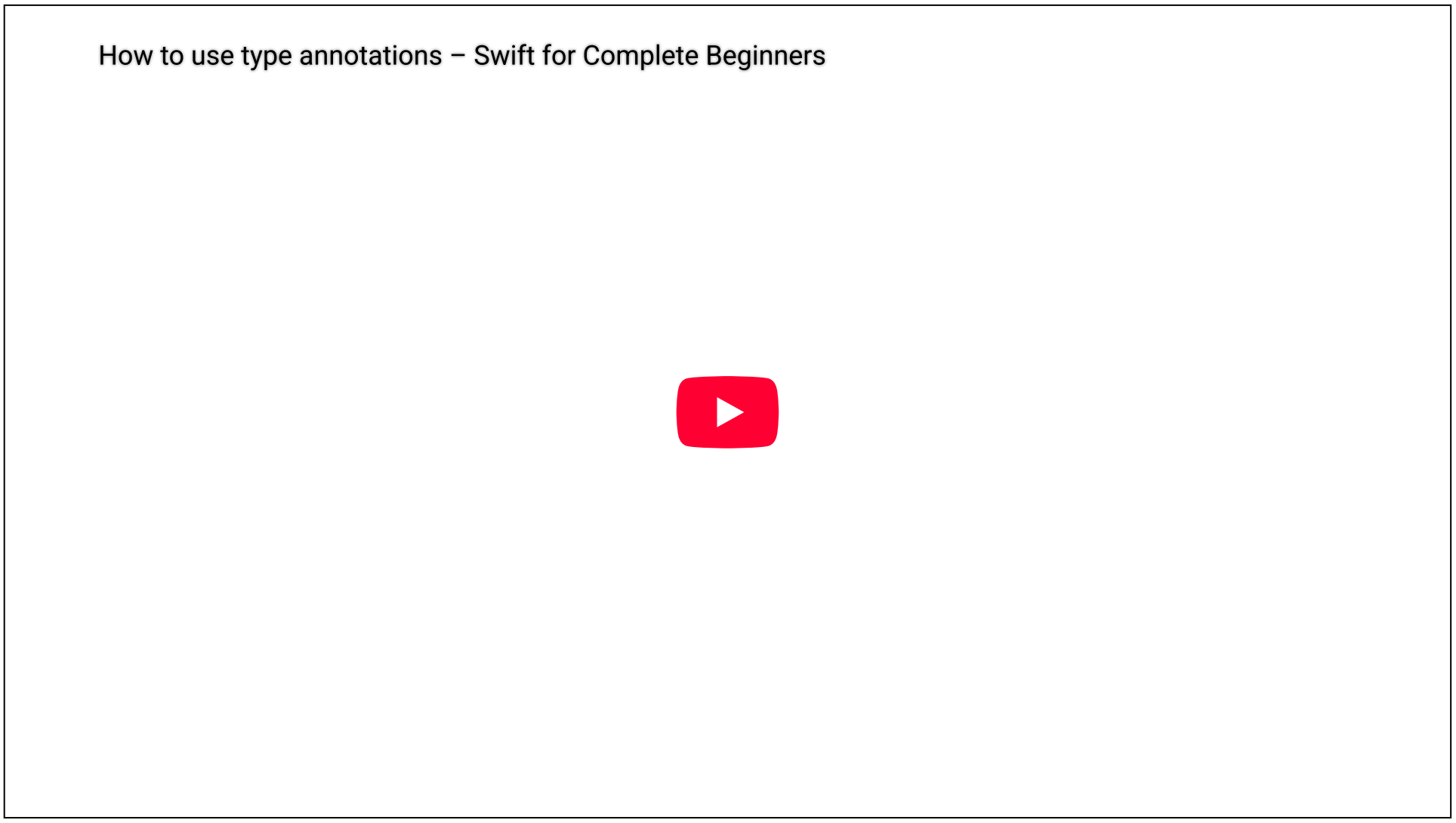
< How to create and use enums

Summary: Complex data >

How to use type annotations

Paul Hudson @twostraws December 28th 2021

Updated for Xcode 16.4



Swift is able to figure out what type of data a constant or variable holds based on what we assign to it. However, sometimes we don't want to assign a value immediately, or sometimes we want to override Swift's choice of type, and that's where type annotations come in.

So far we've been making constants and variables like this:

```
let surname = "Lasso"
var score = 0
```

This uses *type inference*: Swift *infers* that `surname` is a string because we're assigning text to it, and then infers that `score` is an integer because we're assigning a whole number to it.

Type annotations let us be explicit about what data types we want, and look like this:

```
let surname: String = "Lasso"
var score: Int = 0
```

Now we're being explicit: `surname` must be a string, and `score` must be an integer. That's exactly what Swift's type inference would have done anyway, but sometimes it isn't – sometimes you will want to choose a different type.

For example, maybe `score` is a decimal because the user can get half points, so you'd write this:

```
var score: Double = 0
```

Without the **`: Double`** part Swift would infer that to be an integer, but we're overriding that and saying it's definitely a decimal number.

We've looked at a few types of data so far, and it's important you know their names so you can use the right type annotation when needed.

`String` holds text:

```
let playerName: String = "Roy"
```

`Int` holds whole numbers:

```
var luckyNumber: Int = 13
```

`Double` holds decimal numbers:

```
let pi: Double = 3.141
```

`Bool` holds either true or false:

```
var isAuthenticated: Bool = true
```

`Array` holds lots of different values, all in the order you add them. This must be specialized, such as **`[String]`**:

```
var albums: [String] = ["Red", "Fearless"]
```

`Dictionary` holds lots of different values, where you get to decide how data should be accessed. This must be specialized, such as **`[String: Int]`**:

```
var user: [String: String] = ["id": "@twostraws"]
```

`Set` holds lots of different values, but stores them in an order that's optimized for checking what it contains. This must be specialized, such as **`Set<String>`**:

```
var books: Set<String> = Set(["The Bluest Eye", "Foundation", "Girl, Woman, Other"])
```

Knowing all these types is important for times when you don't want to provide initial values. For example, this creates an array of strings:

```
var soda: [String] = ["Coke", "Pepsi", "Irn-Bru"]
```

Type annotation isn't needed there, because Swift can see you're assigning an array of strings. However, if you wanted to create an *empty* array of strings, you'd need to know the type:

```
var teams: [String] = [String]()
```

Again, the type annotation isn't required, but you still need to know that an array of strings is written as **`[String]`** so that you can make the thing. Remember, you need to add the open and close parentheses when making empty arrays, dictionaries, and sets, because it's where Swift allows us to customize the way they are created.

Some people prefer to use type annotation, then assign an empty array to it like this:

```
var cities: [String] = []
```

I prefer to use type inference as much as possible, so I'd write this:

```
var clues = [String]()
```

As well as all those, there are *enums*. Enums are a little different from the others because they let us create new types of our own, such as an enum containing days of the week, an enum containing which UI theme the user wants, or even an enum containing which screen is currently showing in our app.

Values of an enum have the same type as the enum itself, so we could write something like this:

```
enum UIStyle {
    case light, dark, system
}

var style = UIStyle.light
```

This is what allows Swift to remove the enum name for future assignments, so we can write **`style = .dark`** – it knows any new value for **`style`** must be some kind **`UIStyle`**

Now, there's a very good chance you'll be asking *when* you should use type annotations, so it might be helpful for you to know that I prefer to use type inference as much as possible, meaning that I assign a value to a constant or variable and Swift chooses the correct type automatically. Sometimes this means using something like **`var score = 0.0`** so that I get a **`Double`**.

The most common exception to this is with constants I don't have a value for yet. You see, Swift is really clever: you can create a constant that doesn't have a value just yet, later on *provide* that value, and Swift will ensure we don't accidentally use it until a value is present. It will also ensure that you only ever set the value once, so that it remains constant.

For example:

```
let username: String
// lots of complex logic
username = "@twostraws"
// lots more complex logic
print(username)
```

That code is legal: we're saying **`username`** will contain a string at some point, and we provide a value before using it. If the assignment line – **`username = "@twostraws"`** – was missing, then Swift would refuse to build our code because **`username`** wouldn't have a value, and similarly if we tried to set a value to **`username`** a second time Swift would also complain.

This kind of code *requires* a type annotation, because without an initial value being assigned Swift doesn't know what kind of data `username` will contain.

Regardless of whether you use type inference or type annotation, there is one golden rule: Swift must at all times know what data types your constants and variables contain. This is at the core of being a type-safe language, and stops us doing nonsense things like `5 + true` or similar.

Important: Although type annotation can let us override Swift's type inference to a degree, our finished code must still be possible. For example, this is not allowed:

```
let score: Int = "Zero"
```

Swift just can't convert "Zero" to an integer for us, even with a type annotation requesting it, so the code just won't build.

< How to create and use enums

Summary: Complex data >

Was this page useful? Let us know!

☆☆☆☆☆

Average rating: 4.8/5