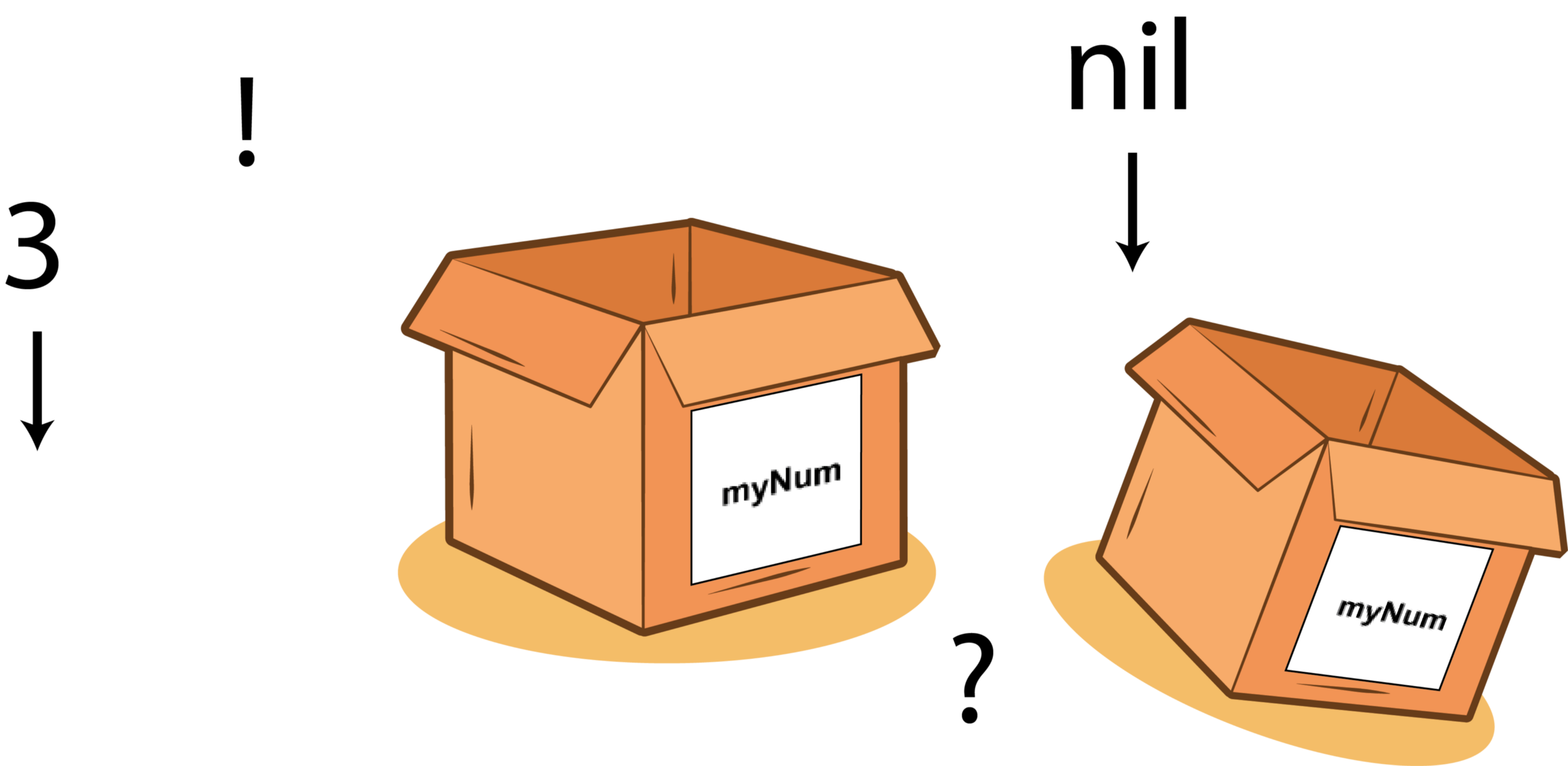
**Simply Optionals? In Swift (Or not).**

**Those ! and ? symbols mean something.**

All of these exclamation marks and question marks mean something. Read on to find out!

You might struggle with the question mark and exclamation mark that litter Swift code. You might even read my [article](https://medium.com/@stevenpcurtis.sc/re-implement-optionals-in-swift-41129477934c)on how you might implement optionals, but want a little more detail.

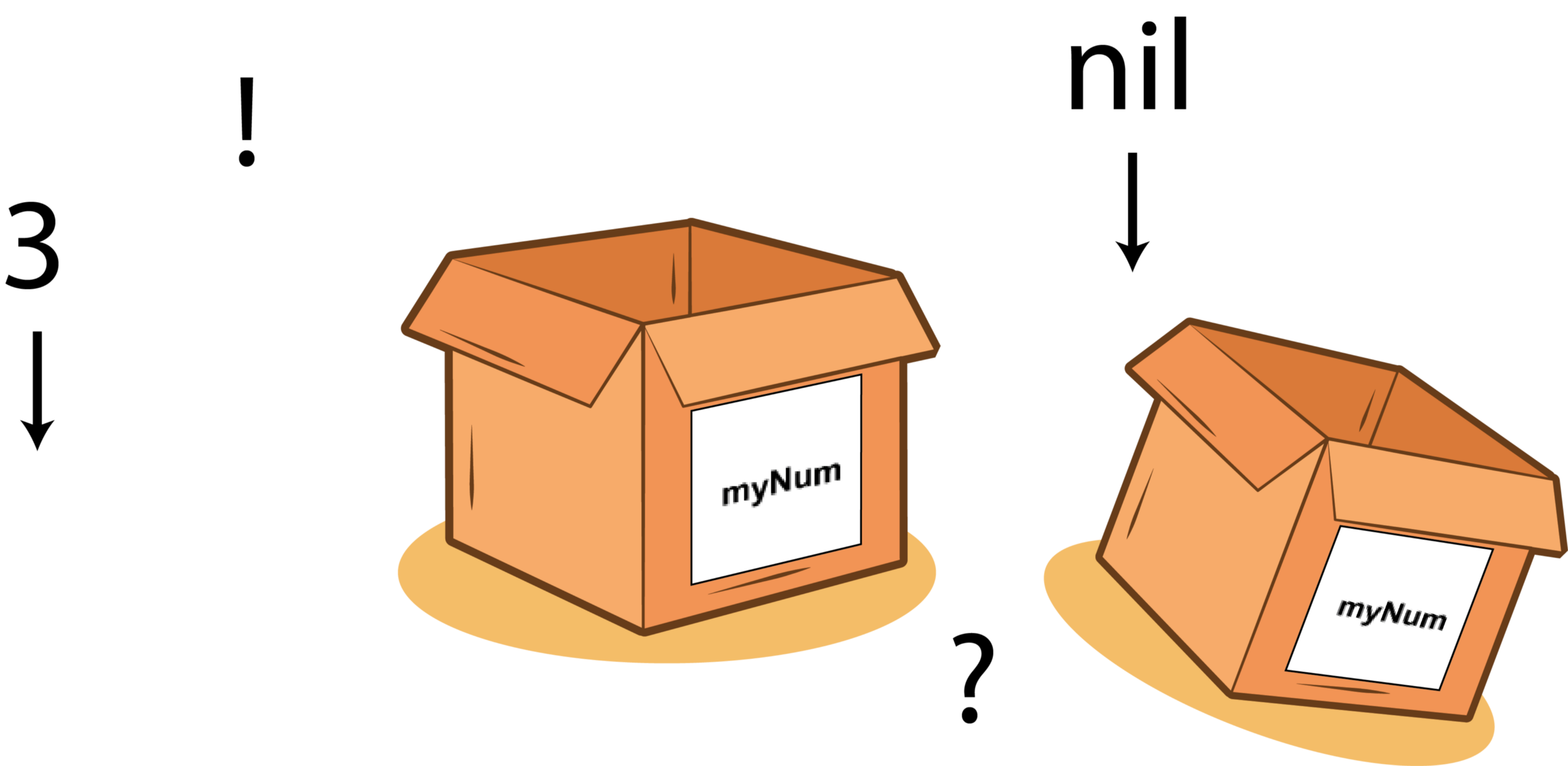
Optionals have been implemented in the language to deal with the problem of (nil, null?, **missing**) missing values. This means that an optional value can either contain a value or the absence of a value.

***Read on for that detail.***

Difficulty: Beginner | **Easy**| Normal | Challenging

**Simply Optionals? In Swift (Or not).**

**Those ! and ? symbols mean something.**

All of these exclamation marks and question marks mean something. Read on to find out!

You might struggle with the question mark and exclamation mark that litter Swift code. You might even read my [article](https://medium.com/@stevenpcurtis.sc/re-implement-optionals-in-swift-41129477934c)on how you might implement optionals, but want a little more detail.

Optionals have been implemented in the language to deal with the problem of (nil, null?, **missing**) missing values. This means that an optional value can either contain a value or the absence of a value.

***Read on for that detail.***

Difficulty: Beginner | **Easy**| Normal | Challenging

**Prerequisites:**

* Be able to produce a “Hello, World!” iOS application (guide [HERE](https://medium.com/@stevenpcurtis.sc/your-first-swift-application-without-a-mac-79598ad839f8))
* Knowledge of variables in any relevant language (guide [HERE](https://medium.com/@stevenpcurtis.sc/what-is-a-variable-3447ac1331b9))

**Terminology**

Force unwrapping: Not only the title of a great [Medium article](https://medium.com/@stevenpcurtis.sc/avoiding-force-unwrapping-in-swift-6dae252e970e), but a method in which an optional is forced to return a concrete value. If the optional is nil, this will cause a crash.

Initialization: The process of preparing an instance of a class, structure, or enumeration for use

nil coalescing: A method that provides a default value for an optional

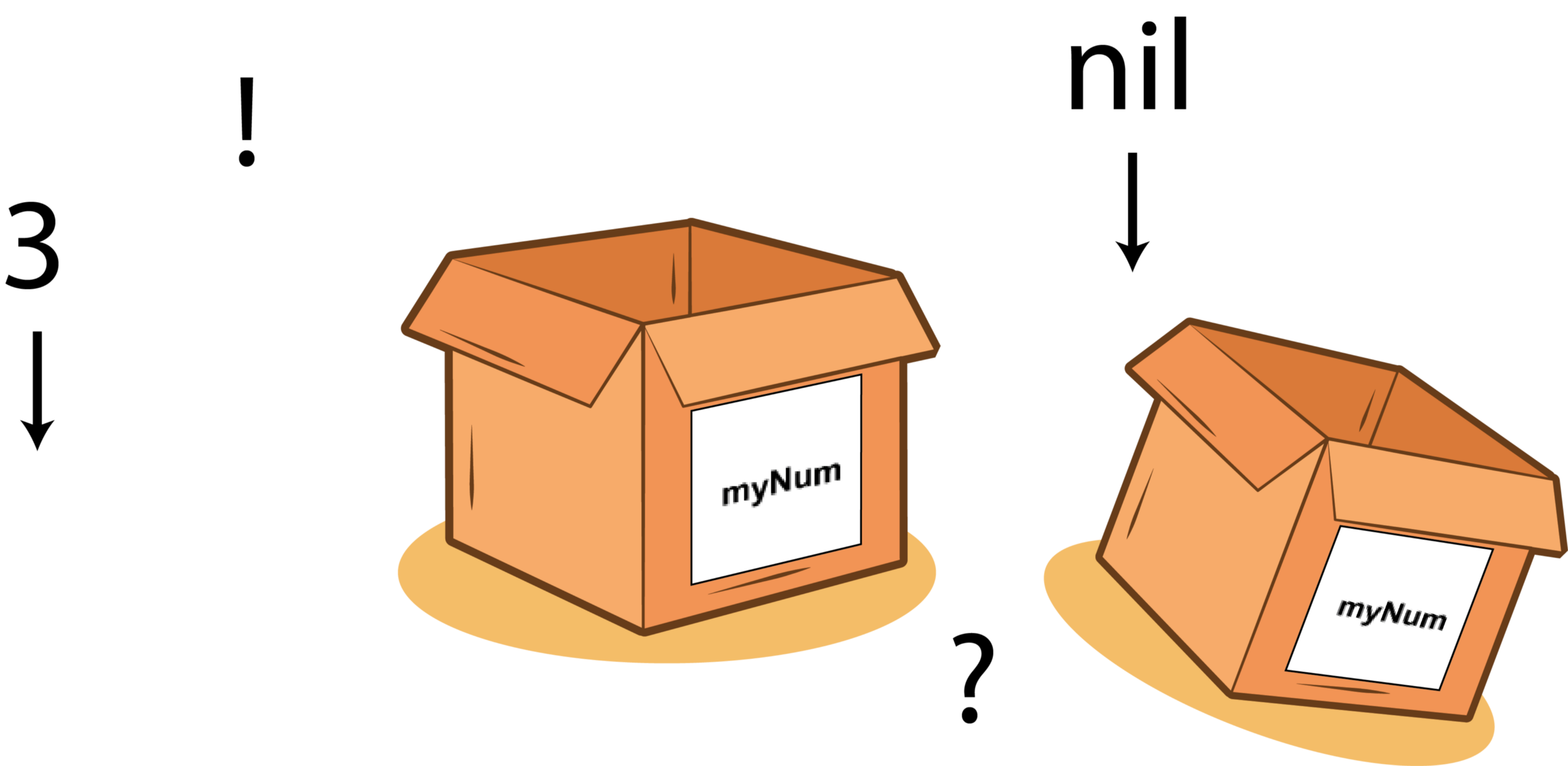
Optional binding: Using an if let pattern to unwrap an optional safely

Optional chaining: A technique where if any value within the optional chain is nil, the whole chain returns nil. If every element returns a value, the calculated value will be returned

Optionals: Swift introduced optionals that handle the absence of a value, simply by declaring if there is a value or not. An optional is a type on it’s own!

**Simply Optionals? In Swift (Or not).**

**Those ! and ? symbols mean something.**

All of these exclamation marks and question marks mean something. Read on to find out!

You might struggle with the question mark and exclamation mark that litter Swift code. You might even read my [article](https://medium.com/@stevenpcurtis.sc/re-implement-optionals-in-swift-41129477934c)on how you might implement optionals, but want a little more detail.

Optionals have been implemented in the language to deal with the problem of (nil, null?, **missing**) missing values. This means that an optional value can either contain a value or the absence of a value.

***Read on for that detail.***

Difficulty: Beginner | **Easy**| Normal | Challenging

**Prerequisites:**

* Be able to produce a “Hello, World!” iOS application (guide [HERE](https://medium.com/@stevenpcurtis.sc/your-first-swift-application-without-a-mac-79598ad839f8))
* Knowledge of variables in any relevant language (guide [HERE](https://medium.com/@stevenpcurtis.sc/what-is-a-variable-3447ac1331b9))

**Terminology**

Force unwrapping: Not only the title of a great [Medium article](https://medium.com/@stevenpcurtis.sc/avoiding-force-unwrapping-in-swift-6dae252e970e), but a method in which an optional is forced to return a concrete value. If the optional is nil, this will cause a crash.

Initialization: The process of preparing an instance of a class, structure, or enumeration for use

nil coalescing: A method that provides a default value for an optional

Optional binding: Using an if let pattern to unwrap an optional safely

Optional chaining: A technique where if any value within the optional chain is nil, the whole chain returns nil. If every element returns a value, the calculated value will be returned

Optionals: Swift introduced optionals that handle the absence of a value, simply by declaring if there is a value or not. An optional is a type on it’s own!

**Using optionals in Swift**

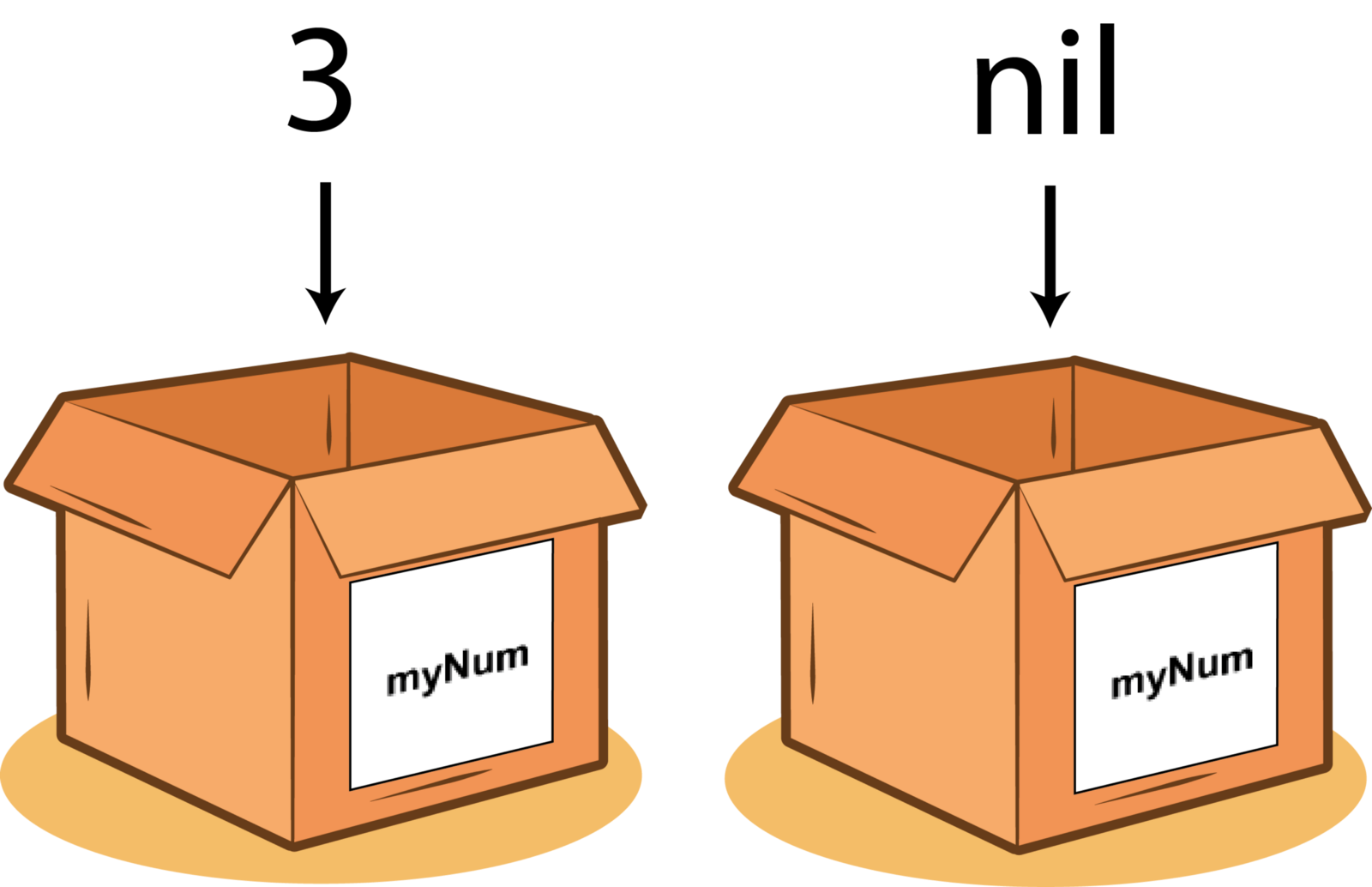
You can define optionals in Swift, for example we can declare that an integer is an optional with the following letassignment statement.

let myNum: Int? = 3

so the ?after the Intdata type states that the variable named myNum can either be populated with an Integer, or nothing.

So to make this ***second***possibility real we can set the same optional to nil

let myNum: Int? = nil

Placing the number 3 into myNum, which can contain an Integer OR an optional. So we can place an Integer into the variable, and as should be expected the variable will contain that Integer. If nil is placed within the variable, the variable will contain nil.

**An optional without being initialized**

If we don’t initialize our variable

let myNum: Int?

then it contains nil. Without us clearly stating that.

***Neat: right?***

### Something and nothing

You will often hear that optionals are values that can either contain **none**or **some**. In this case, some represents that the optional has been set with a value and none means that the optional has not been set. This is idea that has been explained with illustrations above.

**Forced unwrapping**

You might find some idiots using [Star Wars to explain force unwrapping](https://medium.com/@stevenpcurtis.sc/avoiding-force-unwrapping-in-swift-6dae252e970e). This is an abridged version, this time using Integers to explain.

let myNum: Int? = 3  
print (myNum)

Now this prints the following:

Optional(3)

which may not be to your taste — you just want to print out the number 3. So in this case you might be tempted to use !to remove the Optional part of the printout, which would look like the following:

let myNum: Int? = 3  
print (myNum!)

Which seams like a great solution. The console now prints the following:

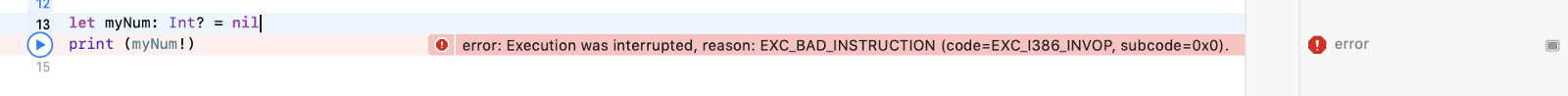
3

Now the problem is we used the exclamation mark (!) and this isforce unwrappingwhich means that you are insisting to the compiler that there is a value in myNum. So what happens if we do the same and there *isn’t*a declared value for myNum?

let myNum: Int? = nil  
print (myNum!)

You get a crash at *runtime*.

This means that when you try to run the code you get a frightening error message, and if you are using the code from Playgrounds you get this:



whereas if you are running code in an App it will just close the App on your user.

***This isn’t good***. ***If you are employed to create iOS Apps and they close on your user, perhaps you won’t be creating Apps for much longer.***

**Optional Binding**

One way of avoiding force unwrapping (again, I covered further techniques in my article [HERE](https://medium.com/p/6dae252e970e/edit)) is optional binding.

This is a method of safe unwrapping that uses an *if let*pattern to make sure there is a value in an optional variable before doing anything with it.

In the first case (where myNum is nil) the print statement will ***not***be executed:

let myNum: Int? = nil  
if let num = myNum {  
 print (num)  
}

in the second case (where myNum has a value, specifically 3 in the example) the print statement **will**be executed

let myNum: Int? = 3  
if let num = myNum {  
 print (num)  
}

Meaning that the code above will print the following to the console:

3

**nil coalescing**

You might want to make sure that there is a default value when an optional value is nil.

let myNum: Int? = 3  
print (myNum ?? 5)

Now here the output is fairly obvious:

3

but if the value is initially set to nil the given value of 5 would be printed

let myNum: Int? = nil  
print (myNum ?? 5)

so the output is as follows:

5

**Implicitly Unwrapped Optionals**

You might have set an optional, and know it will never be nil. We can tell the compiler this with an exclamation mark where the type is declared.

With our Integers this would look like the following:

var myNum: Int! = 3

Now it seems pretty pointless to do this with a plain Integer, so we need an example from the real world…

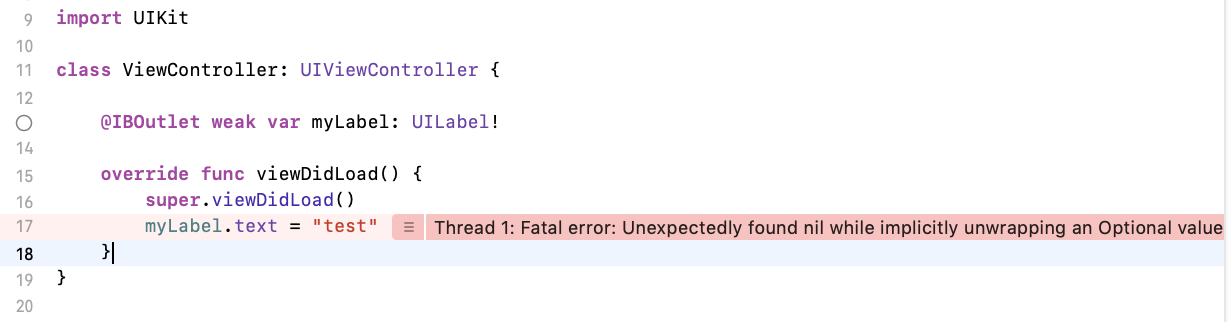
You might actually have seen these used when people set up outlets in iOS applications.

Concretely this can look like the following:

**@IBOutlet** **weak** **var** myLabel: UILabel!

Which means that we assume that the outlet has been set for the label (look at the simple example in the repo for how this can work in the real world).

You may have seen the danger of this in your code — when you disconnect the outlet since you are basically*assuming the outlet is set — that the UILabel is not nil* — you once again get a nasty crash.



However outlets are set this way as we expect them to be set during the instantiation process for the viewcontroller in our app.

**Optional chaining**

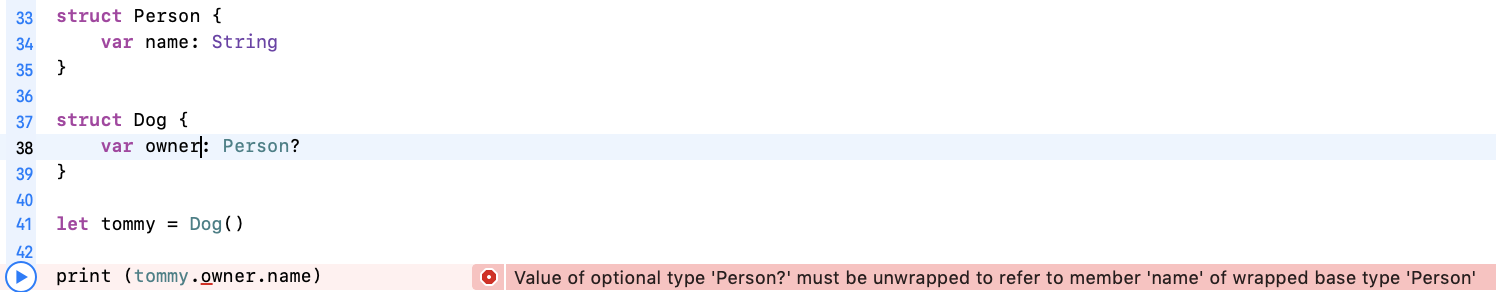
You might have some experience of [structs and classes](https://medium.com/swift-coding/when-to-use-class-or-struct-in-swift-e6037147c1d), which would help you a little following this guide:

struct Person {  
 var name: String  
}

struct Dog {  
 var owner: Person?  
}

let tommy = Dog()  
print (tommy.owner.name)

which gives a nasty-looking error which needs to be handled



This means it needs to be unwapped. But, as you might have guessed, this can cause a rather predictable error!

struct Person {  
 var name: String  
}

struct Dog {  
 var owner: Person?  
}

let tommy = Dog()  
print (tommy.owner!.name) // Gives a runtime error

So this can be simply avoided by replacing our force-unwrapping singnifying !with ?to optionally unwrap. That is, if any value in the chain is nilthe whole value chain will resolve to nil.

struct Person {  
 var name: String  
}

struct Dog {  
 var owner: Person?  
}

let tommy = Dog()  
print (tommy.owner?.name) // print nil

and likewise we can create a dog with a owner, and the relevant print statement will print the name of the owner

struct Person {  
 var name: String  
}

struct Dog {  
 var owner: Person?  
}

var bear = Dog()  
let prince = Person(name: "Dave")  
bear.owner = prince  
print (bear.owner!.name) // prints Dave  
print (tommy.owner?.name) // prints Optional("Dave")

the second of the print statements in the code block above prints safely to the console. That is, it will not crash at runtime.

***Excellent***.

**Conclusion…**

Optionals provide the Swift language with the flexibility to represent values and the absence of the same. There are several different ways that you might interact with optionals, and here is how you might choose between them:

* forced unwrapping, which should be used when you are certain that an optional is not nil
* optional binding, which can be used to only perform an operation when an optional has a value
* implicitly unwrapped optionals can be used when you want to avoid unwrapping everywhere in your code
* nil coalescing is a way to provide default values in the case of an optional being nil

**The code repo**

Here it is to help you out: