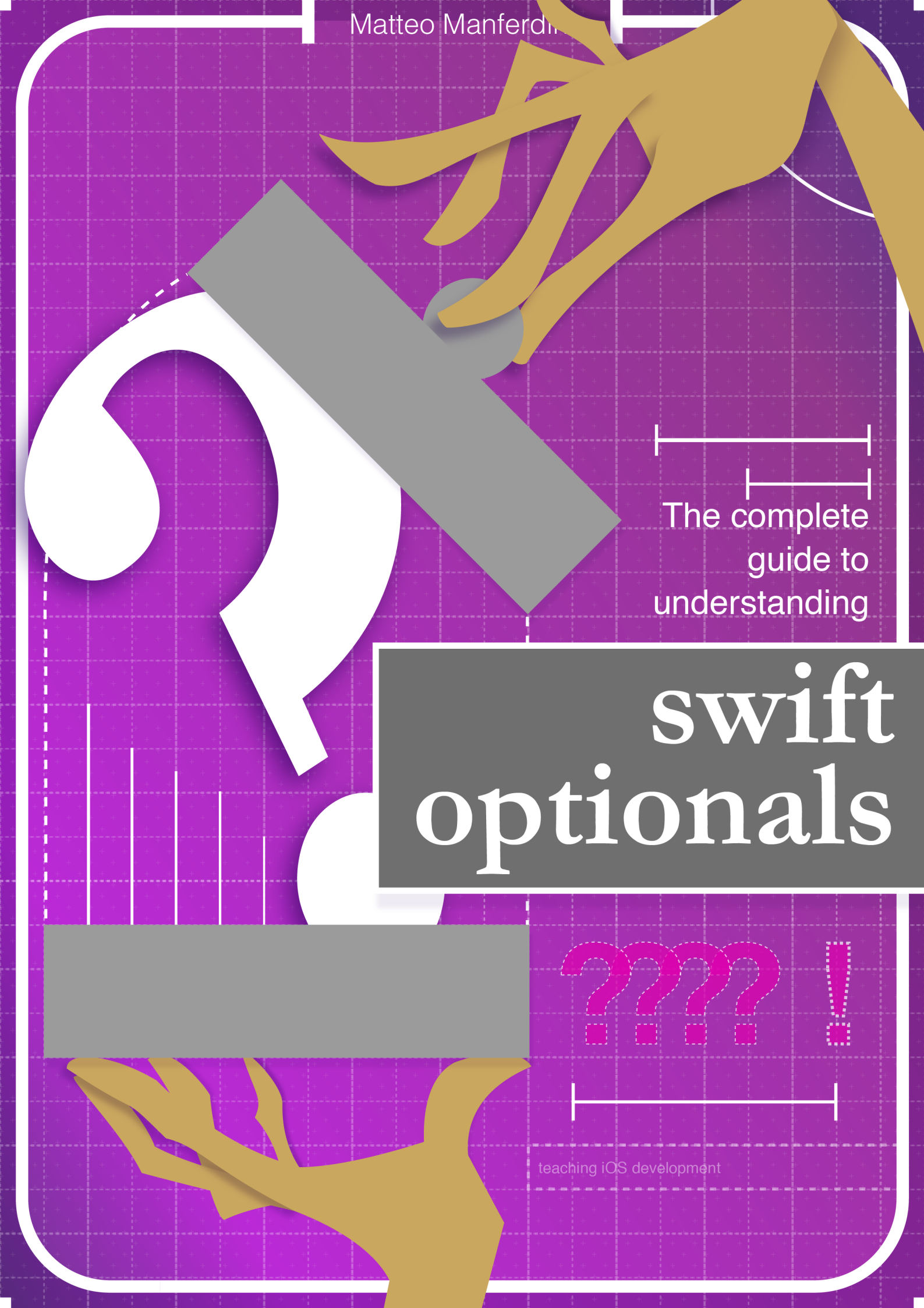
Matteo Manferdi



The complete guide to understanding

swift optionals

???!

teaching iOS development

A LITTLE PROMISE FROM YOU

It took me many years of experience to gather the knowledge that helped me make this guide and hours to actually produce it. But I am happy to offer it for you completely free of charge.

In exchange I ask you for a little promise.

If after reading this guide you will think I did a good job, I ask you to share this guide with someone you think would also benefit from it.

In this way I can spend less time looking for the people I can help and more actually producing more great free material. You will find some links at the end of this guide. Thank you.

Let's start.

1. DO YOU REALLY UNDERSTAND WHAT OPTIONALS ARE?

Since the introduction of Swift, optionals seem to have caused a lot of confusion in people learning the language.

This problem is not shared by beginners only. Many experienced programmers coming from other languages need time to get used to the concept of optionals. That's because optionals are a novel concept and they behave differently from nil (or null) values in other languages.

Admittedly, it also took a while for me to wrap my mind around optionals when I learned Swift.

In my case, the reason was that I did not take the required time to fully understand what optionals actually are. And I suspect that this is probably the case for many others.

My first approach was to just try and use them in some project I was working on.

That did not go well.

All the question marks and exclamation marks that popped-up here and there in my code only left me more and more confused.

While I understood a little of what they meant (or so I thought), in many places I did not know why I had to write code in a certain way or whether it was correct.

If you are new to programming, it might be hard to see why optionals exist in the first place. It's not clear what problems they solve and when to use them.

Besides this, all the different Swift operators for optionals add to the confusion because they can have various meanings in different contexts.

I wrote this guide to shed light on optionals and help you finally understand what they are and how they to use them.

We are going start from the basics and then move to more and more advanced uses. This guide covers **all the possible cases** in which you need optionals in Swift.

There are many, so let's dive in.

2. WHY DO WE NEED OPTIONALS IN THE FIRST PLACE?

In an ideal world, the code we write always works with complete and well-defined data and every operation returns a definite value.

Reality though is quite different.

Often, we have to deal with operations that might or might not produce a value.

- A function might not be able to calculate a result for every input.
- We might be looking for something that does not exist, like an entry in a list, or a file on a disk.
- We might have reached the end of a sequence, a file, or a stream of data coming from the network.
- The user of your app might not have provided all the required input.

And so on.

Let's start with a small, simple example.

A standard programming exercise you find in many textbooks is writing a function to calculate the factorial of a number.

The factorial of an integer is the product of all the positive integers up to such number. For example, the factorial of 5 is $1 \times 2 \times 3 \times 4 \times 5 = 120$. Pretty simple.

Let's write a function to calculate it in Swift:

```
func factorial(_ number: Int) -> Int {
     var result = 1
     for factor in 1...number {
          result = result * factor
     }
     return result
}
```

That's pretty straightforward. Let's now try to use it:

```
factorial(5)
// 120
factorial(7)
// 5040
factorial(-3)
// Error
```

That last one does not look good.

That error gives a complicated message in a Swift playground. In a real app though you would not be so lucky. The error would go undetected and cause your app to crash at runtime.

The reason is that the range in the **for** loop in the **factorial** function does not work when we feed a negative number to it.

And this is not even the only problem.

We cannot fix the function even if we replace that range with something else, for example, a **while** loop.

That's because the factorial of negative numbers does not exist in mathematics. We just don't have a definition for it.

So, our **factorial** function needs to return a unique value for negative inputs, to communicate that the result does not exist.

Programmers resort to many different particular values to represent results that do not exist. These are called *sentinel values*.

So, what sentinel value could we return from our **factorial** function?

We could return **-1** to represent "there is no factorial for this number." This would work because the factorial of a number is always a positive integer, according to the mathematical definition. And indeed, this is usually the choice in other programming languages.

But it's not an ideal solution.

When someone else uses this function, how does he know that **-1** means “no result”? It's impossible for someone that does not know the mathematical definition of the factorial function to tell whether this result is correct or not.

We could write some documentation for the function, but others might not read it and still misuse the function. The incorrect value would then propagate to other parts of their program, causing hard to find bugs.

This approach becomes even more problematic when **-1** is a valid return value for a function. And, often, we are not able to find an invalid value to use as a sentinel.

Many languages use a unique value, called **nil** (or sometimes **null**) to represent the absence of a value. In many of these languages though, a **nil** value is possible only for references to objects. Simple values like integers cannot be **nil**.

Swift solves this problem with optionals.

When we put a **?** after a type declaration, we state that there could be a value or no value at all. Like other languages, Swift also uses **nil** to mean “no value.”

That is our sentinel, and it is always going to work because **nil** is not a valid value for anything.

We can now rewrite our factorial function to return **nil** for negative numbers:

```
func factorial(_ number: Int) -> Int? {
     if (number < 0) {
          return nil
     }
     var result = 1
     for factor in 1...number {
          result = result * factor
     }
     return result
}

factorial(-3)
// nil
```

The return type of the function is not **Int** anymore, but **Int?** instead. This declaration now explicitly states: "the return value might be an integer or **nil**."

This is a crucial difference between Swift and all other languages that also use **nil** values. In those other languages, you never know if a value returned by a function could be **nil** or not. You need to read the documentation if any is provided at all.

In Swift, this is clear from the type. We always know which values could be **nil** and which ones are always valid.

A little side-note: as a reader suggested, the factorial function could also check its input through an assertion, making sure that a negative value is never passed.

Since the factorial is undefined for negative parameters, the function should not accept any at all.

This would be a valid approach, but assertions are a complex topic, and they are beside the point of this guide. Since this is just a simple example to understand Swift optionals, I prefer to return **nil**.

3. USING OPTIONAL VALUES

Now that we have seen why optionals are useful, and why we need them, let's see how to use them.

In Swift, the developer is not the only one to know which values are optional from a type declaration. The compiler also reads this information and can make a series of checks that are not possible in languages without optionals.

In practice, this means that the compiler can make sure that we never put a **nil** value where we are not supposed to.

It does not allow us to do it and gives us an error as soon as we try to run such code. Or, even better, Xcode checks our code in real time and shows the error as soon as we type it.

It is always good to delegate to the compiler as many checks as possible. This helps us make fewer mistakes.

Humans forget things all the time. The compiler, instead, won't let any **nil** value get into places where it is not expected.

Unfortunately, this might become frustrating when you don't understand what is going on.

The compiler might complain about something that looks correct to you. But in the end, it's the compiler that is always right. You need to understand what it tries to tell you.

This happens very often with optionals, causing a lot of confusion.

FORCED UNWRAPPING

Let's try to use the factorial function we just created:

```
factorial(3) + 7
// Value of optional type 'Int?' must be unwrapped to a value of type 'Int;
```

Unfortunately, it fails.

The compiler seems to be complaining about something, but it's not clear what.

Why cannot we add these two numbers? After all, we are sure that the factorial of **3** is **6**, so it seems that our operation should be possible.

The problem is that we are trying to add two things that have two distinct types for the compiler. Remember, the compiler is not going to let a **nil** slip anywhere.

The factorial function does not return an **Int** but an **Int?**. You might think that the **?** operator says "there might be a **nil** value," but to the compiler, these are two completely different types.

We will see later why they are different when we look into more advanced concepts. For now, and most of the time, you only need to keep in mind that an optional type and its base type are different.

And you won't forget it since the compiler will remind you all the time.

Since the operands of the addition have distinct types, we can't add them. At the same time, we know that it should be possible because we know the result of the factorial.

The error message comes with two suggestions:

Coalesce using '??' to provide a default when the optional value contains 'nil'

and

Force-unwrap using '!' to abort execution if the optional value contains 'nil'

Again, not the best ones for someone that does not know much about optionals.

Let's follow the second one. To perform the addition, we transform the optional value into a non-optional one using the *forced unwrapping operator*, or **!**

```
factorial(3)! + 7
// 13
```

The forced unwrapping operator tells the compiler: “I know that this optional has a value, so use it.”

It works here, but most of the times it's dangerous to use forced unwrapping. While in this simple example we know that the result will not be **nil**, most of the time we don't.

If you use forced unwrapping on a **nil** value, it will cause a runtime exception and make your app crash.

So, you should use it only when you can say: “I am so sure that this value will not be **nil** that I want my app to crash if it isn't."

As we will see, there are rare cases in which crashing is better than going on with a wrong value. But most of the time you won't be so sure.

We need a different approach.

OPTIONAL BINDING

Since forcing the unwrapping of a **nil** value causes a crash, we have to make sure that an optional has a value before we use it.

```
let result = factorial(3)
if result != nil {
     result! + 7
}
// 13
```

Here we make sure that **result** is not **nil** before we use the forced unwrapping.

This works, but it's a bit too repetitive. And this is something we need to do quite often, so it matters. Moreover, if we have to use the **result** constant again later, we have to use the **!** operator every time.

For this reason, **if** statements support *optional binding*, which we can use to check an optional and extract its value, if there is one, in a single instruction.

In this way, forced unwrapping is not necessary anymore.

```
if let result = factorial(3) {
     result + 7
}
// 13
```

This also works with multiple bindings, in case we need to check more than one optional at the same time:

```
if let x = factorial(3), let y = factorial(5), let z = factorial(7) {
     x + y + z
}
// 5166
```

Sometimes, instead of checking the results of a function, we need to check optionals stored in a constant/variable.

But finding a new name for the binding is not easy. Proper variable names are always a problem when writing code. Cleary, doubling the trouble for each optional is not desirable.

Luckily, you don't have to. Optional binding creates new constants in the scope of the **if** body. So you can reuse the same names:

```
let x = factorial(3)
let y = factorial(5)
let z = factorial(7)
if let x = x, let y = y, let z = z {
     x + y + z
}
// 5166
```

We can also add boolean expressions optional binding as if it was a normal **if** statement, to add extra checks on the values of the bindings. For example, let's assume that we want to use the even results of the factorial function.

```
func isEven(_ number: Int) -> Bool {
     return number % 2 == 0
}

let x = factorial(3)
let y = factorial(5)
let z = factorial(7)

if let x = x, let y = y, let z = z,
     isEven(x) && isEven(y) && isEven(z) {
     x + y + z
}
// 5166
```

I added a little **isEven(_:)** function to help since we have to repeat the check three times:

On an unrelated note, the above example will almost always execute the body of the **if** because factorials are always even. Except for the factorials of **0** and **1** (which are both **1**), every factorial is even because it always contains **2** in its factors.

Optional binding also works in **while** statements. A **while** loop will continue while the binding produces a valid value and will stop when it finds **nil**.

For example, let's assume we have an array of numbers and we want to find the index of the first one that does not have a factorial.

```
let numbers = [2, 5, 3, -1, 9, 12, 4]
var index = 0
while let result = factorial(numbers[index]) {
     index += 1
}
print(index)
// 3
```

NIL COALESCING

Often, we want to use a default value when an optional is **nil**. Of course, we can use optional binding and but the default value in an **else** branch:

```
let result: Int
if let factorial = factorial(-7) {
     result = factorial
} else {
     result = 0
}
// result is 0
```

This is a bit tedious. That's why there is a shorter and more practical way of writing this code: the *nil coalescing operator.*

The **??** operator can do the same in a single line of code.

```
let result = factorial(-7) ?? 0
// result is 0
```

Practically, this means "if the factorial is **nil**, use **0**". If an optional has a value, **??** returns it. Otherwise, it returns the second value.

A nice feature of this operator is that if you can chain is as many times as you want.

For example, if you have more than one optional and you want to take the first valid value, you can chain multiple nil coalescing operators in the same instruction:

```
let x = factorial(-3) // nil
let y = factorial(5) // 120
let z = factorial(7) // 5040
let result = x ?? y ?? z ?? 0
// result is 120
```

Notice that the nil coalescing always produces a value, so its result is not an optional.

If that's what you want, you can to omit the default value at the end. In that case, the result will be **nil** if no instruction produces a value.

```
let x = factorial(-3)
let y = factorial(-5)
let z = factorial(-7)
let result = x ?? y ?? z
// result is nil
```

HAPPY PATH AND EARLY EXIT

Let's say we want to write a little function that takes a name, age, and city and creates a greeting message with them.

Our function should only work if all three values are present. If any is missing, it produces an error message instead.

```
func messageWith(name: String?, age: Int?, city: String?) -> String {
     if let name = name, let age = age, let city = city {
          return "Hi \(name), I see you are \(age) years old. Nice age. And
how
               is it living in \(city)?"
     } else {
          return "You did not provide all the information I need!"
     }
}

let message = messageWith(name: "Matteo", age: 35, city: "Amsterdam")
print(message)

// Hi Matteo, I see you are 35 years old. Nice age. And how is it living in
Amsterdam?
```

This works well, but the error message does say which information is missing. Let's write a better version that does.

We will generate the message using the information available. Then, we will complete the message reporting the first missing parameter.

Since this time we generate different messages, we cannot use multiple bindings in a single if statement. Instead, we need to check each parameter in a separate one.

Let's give it a try.

```
func messageWith(name: String?, age: Int?, city: String?) -> String {
     if let name = name {
          if let age = age {
               if let city = city {
                    return "Hi \(name), I see you are \(age) years old. Nice
                         age. And how is it living in \(city)?"
               } else {
                    return "Hey \(name), I know you are \(age) years old, but
                         you didn't tell me where you live"
               }
          } else {
               return "\(name), you forgot to tell me how old you are"
          }
     } else {
          return "Please introduce yourself, at least!"
     }
}
```

This works, but it's not readable because of the nesting of the **if** statements. And any extra check makes things worse, pushing the nest-

ing more and more to the right. That is why this is generally known as the *pyramid of doom*.

The problem here is that the code that assembles the full message gets lost in the pyramid, mixed to the error checking code.

The code that does the work is called the *happy path* or *golden path*. It is the path followed when no errors happen. The happy path is the part we care about when we read some code. The rest is there only to handle errors.

In our example, the business logic is only one line long, but this is rarely the case. Real code is more complicated than that, and we don't want to lose its meaning in a pile of error messages.

This is a typical pattern which occurs again and again in real code. Often, it gets even worse because in each intermediate step we need to handle the partial result.

This is how it looks like:

```
if first condition {
     do something with result
     if second condition {
          do something with second result
          if third condition {
               do something with third result
          } else {
               error handling for third condition
          }
     } else {
          error handling for second condition
     }
} else {
     error handling for first condition
}
```

All those "do something" instructions (in bold) represent the happy path. That's the code we want to be able to read comfortably.

Notice also that the error handling for each condition gets separated from the condition that generates it. Not only that, but error handling is arranged in reverse order. The last error statement is far away from the corresponding condition, which appears in the first if statement.

We want all the actual business logic to be easy to spot and the error handling to be kept close to the corresponding conditions.

We do so by checking for errors in our if statements instead of checking for success. When we spot an error, we address it and return from the function immediately.

This keeps the errors close to the checks and the business logic outside of if statements, pushing it to the left where it's easier to find.

```
if first error {
     first error handling
     return
}
do something with first result
if second error {
     second error handling
     return
}
do something with second result
if third error {
     third error handling
     return
}
do something with third result
```

So let's rewrite our function, pushing the happy path to the left:

```
func messageWith(name: String?, age: Int?, city: String?) -> String {
     if name == nil {
          return "Please introduce yourself, at least!"
     }
     if age == nil {
          return "\(name!), you forgot to tell me how old you are"
     }
     if city == nil {
          return "Hey \(name!), I know you are \(age!) years old, but you
               didn't tell me where you live"
     }
     return "Hi \(name!), I see you are \(age!) years old. Nice age. And how
is
          it living in \(city!)?"
}
```

This is much better, but it still has a little annoyance.

We lost the optional bindings, and we have to resort to forced unwrapping in every message. This works, because we make sure that each optional is not **nil**.

But when we use a value more than once this gets annoying. And moving this code around might put forced unwrapping in places where it should not go.

To solve this problem, Swift 2 introduced *guard* statements. Guard statements allow us to check for **nil** and return early while at the same time bind the a value to a constant.

```
func messageWith(name: String?, age: Int?, city: String?) -> String {
     guard let name = name else {
          return "Please introduce yourself, at least!"
     }
     guard let age = age else {
          return "\(name), you forgot to tell me how old you are"
     }
     guard let city = city else {
          return "Hey \(name), I know you are \(age) years old, but you didn't
               tell me where you live"
     }
     return "Hi \(name), I see you are \(age) years old. Nice age. And how is
          it living in \(city)?"
}
```

Thanks to these guard statements, we don't need to use forced unwrapping anymore.

When you use a guard statement, you must exit early. This means that you always have to include an exit statement:

- **return** to exit from a function; or
- **continue** or **break** inside for and while loops.

4. OPTIONALS IN STRUCTURES AND CLASSES

OPTIONAL CHAINING ON PROPERTIES

Optionals are not only used as return values for functions. We can also use them for the properties of structures and classes.

In these cases, we use optionals to say that a variable or property might, sometimes, not hold a value.

As an example, let's create a structure to hold the information for people subscribed to a public library. Every person has a library card which contains the list of books the person borrowed.

```
struct Person {
     var libraryCard: LibraryCard?
}

struct LibraryCard {
     var numberOfBorrowedBooks = 0
}
```

A person might be in the database for different reasons (maybe they are employees who don't read books). So, the **libraryCard** property of **Person** is optional and will be **nil** for people with no card.

We now want to print how many books a person has borrowed in the past. To get to the **numberOfBorrowedBooks** property, we have to traverse the optional **libraryCard** property of **Person**, which might be **nil**.

We can, of course, use an optional binding. When **libraryCard** is not **nil**, then we access **numberOfBorrowedBooks** on the unwrapped value.

But this becomes tedious pretty soon.

For this reason, Swift offers an alternative mechanism that allows us to go through an optional, straight to the property we want to access.

This mechanism is called *optional chaining*, and you use it by putting the **?** operator after an optional property:

```
let john = Person()
if let numberOfBorrowedBooks = john.libraryCard?.numberOfBorrowedBooks {
     print("John has borrowed \(numberOfBorrowedBooks) books")
} else {
     print("John does not have a library card")
}

// prints "John does not have a library card"
```

In this case, the meaning of the **?** operator is different than the one it has when we put it after a type declaration.

While the latter meant "this value might not exist" when we use ? after a variable or a property, it means instead "if there is a value, access the specified property, otherwise return **nil**."

This is one of the things that make optionals confusing. The **?** operator has two different meanings, depending on where you use it.

So, in the example above, when **libraryCard** is not **nil**, we can access its **numberOfBorrowedBooks** property. Otherwise, we get **nil**.

This is different from using the **!** operator, which forces the optional to unwrap and cause a crash when it finds **nil**.

Keep also in mind that, since accessing the **numberOfBorrowedBooks** property might fail, the value returned by optional chaining is also an optional. This is true even when the final property we are accessing is not.

In our example, the **numberOfBorrowedBooks** property has type Int, but the value returned by **john.libraryCard?.numberOfBorrowedBooks** has type **Int?**

This is why in the example I still used optional binding for the result of the chaining.

It is also possible to set a property through optional chaining.

While an assignment would typically not have a return type, attempting to set a property through optional chaining returns a value of type **Void?**.

This allows us to check if the assignment was successful:

```
var john = Person()
if (john.libraryCard?.numberOfBorrowedBooks = 1) != nil {
     print("The assignment was successful")
} else {
     print("It was not possible to assign a new identifier")
}

// "It was not possible to assign a new identifier"
```

OPTIONAL CHAINING ON METHODS

Optional chaining can also be used to call methods on an optional. Let's expand our **LibraryCard** structure to include a list of borrowed books:

```
struct Person {
     var libraryCard: LibraryCard?
}

struct LibraryCard {
     var borrowedBooks: [Book] = []
     var numberOfBorrowedBooks: Int {
          return borrowedBooks.count
     }

     mutating func add(_ book: Book) {
          borrowedBooks.append(book)
     }
}

struct Book {
     let title: String
}
```

To get to the **add(_:)** method of **LibraryCard**, we still have to traverse the **libraryCard** property of **Person**, which is optional.

Like in the case of assignment, the **add(_:)** method has a **Void** return type. But through optional chaining this will become **Void?**, allowing us to check for the success of the method call.

```
var john = Person()
let mobyDick = Book(title: "Moby Dick")
if (john.libraryCard?.add(mobyDick)) != nil {
     print("John has borrowed \(mobyDick.title)")
} else {
     print("John has not a library card and cannot borrow books")
}

// prints "John has not a library card and cannot borrow books"
```

MULTIPLE LEVELS OF CHAINING

Optional chaining can be concatenated to drill down as many levels of optionals as you wish.

Let's add a property to **LibraryCard** to retrieve the last book a person has borrowed:

```
struct LibraryCard {
     var borrowedBooks: [Book] = []
     var numberOfBorrowedBooks: Int {
          return borrowedBooks.count
     }

     var lastBorrowedBook: Book? {
          return borrowedBooks.last
     }

     mutating func add(_ book: Book) {
          borrowedBooks.append(book)
     }
}
```

We can now use multiple levels of chaining to fetch the title of the last book John borrowed, by repeatedly using the **?** operator on any optional we find on our path:

```
var john = Person()
john.libraryCard = LibraryCard()
let mobyDick = Book(title: "Moby Dick")
john.libraryCard?.add(mobyDick)

if let bookTitle = john.libraryCard?.lastBorrowedBook?.title {
     print("The last book John has borrowed is \(bookTitle)")
} else {
     print("John has not borrowed any books")
}

// prints "The last book John has borrowed is Moby Dick"
```

As we have already seen, even when the type of a property or a method is not optional, optional chaining will make it so. If a type is already optional though, it will not make it "more" optional.

In this case, we have two levels of chaining, but the return type is still **String?** and not **String??** (if you are wondering if this is even possible: yes, it is. More on this later).

Finally, you can use optional chaining on subscripts of arrays or dictionaries. Everything I showed here for properties and methods applies to subscripts as well.

OPTIONAL CHAINING ON OPTIONAL PROTOCOL REQUIREMENTS

Until now, we have used optionals for things that might not have a value, like variables/properties or functions/methods.

Optional chaining can also be used for properties and methods in protocols that are not required. These are called *optional protocol requirements*.

This means that the types that conform to a protocol are not forced to implement these requirements. They can choose not to. So in this case, it's not only values that might not exist. Properties and methods might be missing altogether.

Optional protocol requirements only happen in Objective-C protocols. In Swift, the preferred solution is to split requirements into multiple protocols and then use protocol composition.

This also means that you can use them only in Objective-C classes. Swift structures and classes cannot conform to Objective-C protocols.

Still, many protocols in the iOS SDK come from Objective-C. You can also define new Objective-C protocols in Swift. So this is a feature you need to know.

To access optional protocol requirements, you still use the **?** operator, but you put it in a different place. This is again one of the parts of optionals that is confusing if you don't understand the underlying reason.

Let's say we are making an app that deals with shapes. Most shapes, like squares and triangles, have an area. But some unique shapes have no area, i.e., lines and points.

We can express this with a protocol with an optional requirement.

```
import Foundation

@objc protocol Shape {
     @objc optional func area() -> Float
}

class Square: Shape {
     let side: Float

     init(side: Float) {
          self.side = side
     }

     func area() -> Float {
          return side * side
     }
}

class Line: Shape {
     let length: Float

     init(length: Float) {
          self.length = length
     }
}
```

Notice that, this time, we did not declare the return type of the **area()** method to be optional. If a shape has an area, this method always returns a **Float**. A **Square**, for example, always has an area.

But objects that do not have an area will not implement the **area()** method at all. Since it's the whole method to be optional and not its return value, we use the **optional** keyword before it.

You can see that the **Line** class conforms to **Shape**, but does not implement the **area()** method.

We can now make an array of shapes and try to print their area. If they don't have one, we print a special message.

```
let shapes: [Shape] = [Square(side: 2.0), Line(length: 3.0), Square(side:
4.0)]

for (index, shape) in shapes.enumerated() {
     if let area = shape.area?() {
          print("The area of shape \(index) is \(area)")
     } else {
          print("Shape \(index) has no area")
     }
}

// The area of shape 0 is 4.0
// Shape 1 has no area
// The area of shape 2 is 16.0
```

Notice that, this time, the **?** operator is before the parentheses in the method call and not after like it would happen in standard optional chaining. Here, the ? operator checks if the method exists.

The result, though, is still the same we would get with standard chaining. The **area()** call returns a **Float?** on which we use optional binding as usual.

5. AVOIDING THE CHECKS FOR NIL

IMPLICITLY UNWRAPPED OPTIONALS

It's clear by now that optionals add to your code many:

- **?** and **!** operators, and
- **if** and **guard** statements with optional bindings.

While these all add safety, they also make code harder to read. This is one of the trade-offs of optionals.

Sometimes though, we know that after we assign a value to an optional stored property, it will never become **nil** again. In this case, unwrapping and binding are redundant, since that's not an optional anymore.

For these cases, we can declare the property as *implicitly unwrapped*.

We do so by placing a **!** after its type, instead of the usual **?**. This allows the property to start with a **nil** value at initialization time.

But, while the property is still optional, we can use it in code as if it was not. This means no unwrapping, bindings, ifs or guards.

In practice, declaring an optional as implicitly unwrapped is the same as putting the **!** operator after every use. The compiler does it for us.

This makes implicitly unwrapped optionals quite dangerous.

An implicitly unwrapped optional is still an optional. We just made it more convenient to use. You can always assign to it a **nil** value, at any point. And since every use is a forced unwrapping, any of those will cause a crash.

There are a few cases though in which implicitly unwrapped optionals are useful and sometimes needed. We will see them in the following sections.

UNOWNED REFERENCES IN REFERENCE CYCLES

Under ARC, two objects holding a strong reference to each other create a *strong reference cycle*.

This causes the two objects to stay in memory for the lifetime of the app. And when your app passes the memory threshold, the iOS operating system kills it.

Cycles are not always bad and are sometimes needed. It's the strong ones we need to avoid.

To do so, we declare some of the properties involved in a cycle to be *weak* or *unowned*.

- **weak** properties must be declared optional. They automatically become **nil** when an object is removed from memory by ARC.
- **unowned** properties instead are not optionals. But in specific cases, they require other properties to be optional.

There are three particular cases in which we can create strong reference cycles, which we fix with **weak** or **unowned** properties.

1. Both classes have an optional reference to the other one.

This is the easiest to solve since both references are optional.

For example, a person lives in an apartment, and an apartment has a tenant. So the **Person** and **Apartment** objects create a reference cycle.

But these two objects can also exist independently. A person might be homeless, and an apartment might be empty. So both can have optional references.

Here we need to mark one of the references to be **weak,** to avoid a strong reference cycle. Which one you chose depends on the specifics of your code.

```
class Person {
     var apartment: Apartment?
}

class Apartment {
     weak var tenant: Person?
}
```

2. **One of the two references must always have a value, but the other one can be an optional**

This happens when the object with the non-optional property only exists when the other one exists.

For example, a bank account can have an associated credit card, so this property can be optional. A credit card, instead, always needs an associated bank account.

Only optionals can be **weak**, of which we have one. But if we make it **weak**, the credit card object does not stay in memory, since it depends on the existence of a bank account.

We solve this problem by making the non-optional reference **unowned**, which breaks the strong cycle.

```
class Account {
     var card: CreditCard?
}

class CreditCard {
     unowned let account: Account

     init(account: Account) {
          self.account = account
     }
}
```

3. **Both references must always have a value.**

This is the trickiest case and is rare. You can usually get away with the two examples above. But when you want to be strict in your code, you need to know this technique.

Let's see an example: a **Country** must have a **capital**, and a **City** must belong to a **Country**. None of the two classes is allowed to have a **nil** reference to the other one.

This also means that we cannot create the two instances independently since that would require an optional property.

To create the two references at once, one of the two objects must create the other in its initializer. To create the cycle, the first object passes a **self** reference to the initializer of the second one.

But here comes a problem. Swift does not allow you to reference **self** in an initializer until an instance is completely initialized. But to be completely initialized, the first object needs to create an instance of the second, to which it needs to pass **self** as a parameter, which, as I just said, is forbidden.

This is where implicitly unwrapped optionals come to the rescue.

Making the property in the parent object implicitly unwrapped, allows it to temporarily be **nil** until we have the second instance.

```
class Country {
     var capitalCity: City! {
          didSet {
               assert(capitalCity != nil)
          }
     }

     init() {
          self.capitalCity = City(country: self)
     }
}

class City {
     unowned let country: Country

     init(country: Country) {
          self.country = country
     }
}
```

To tell the truth, the first reference is still optional, even if we can use it as if it were not one. Having a cycle with no optionals is impossible.

If you want though, you can force it to never be **nil** again with an assertion. This helps you catch mistakes while developing an app.

INITIALIZATION OF IBOUTLETS

Another typical scenario for implicitly unwrapped optionals are outlets connecting a view controller that comes from a storyboard or a nib file to its views.

The view controller is the first object that gets instantiated. All the views in its interface get only instantiated after that. That means that outlets cannot be connected at initialization time, so they need to be **nil**.

In general, though, once outlets are populated, they never become **nil** again. That is why they are declared as implicitly unwrapped otionals.

FAILABLE INITIALIZERS

Sometimes, the initialization of an object can fail. In that case, the initializer should return **nil** as soon as possible. At that point though not all the properties might have been initialized.

Swift does not allow an **init** method to return until all properties are initialized, even if you want it to fail and return **nil**.

Implicitly unwrapped optionals solve this problem as well. We will see this in more detail later, in the chapter on failable initializers.

A FINAL NOTE ON IMPLICITLY UNWRAPPED OPTIONALS

Implicitly unwrapped optionals are dangerous and should be used sparingly.

Keep in mind that they are still optionals, even if you don't need to unwrap them. So they might become **nil** at any point and, since they get unwrapped with no checks, they can cause a crash.

They are only intended for those cases where you are sure that a value will always exist after initialization. When you are sure of that, implicitly unwrapped optionals are very convenient.

This is not always easy to know in advance, though.

It is easy to fall into the temptation of using implicitly unwrapped optionals for cases that do not fall under this rule. They often look like a very convenient way to stop the compiler from complaining.

Remember that compiler checks are there for a reason. Your goal is not to silence them since it defeats the whole point of optionals.

And even if you are sure you will never forget why an optional is implicitly unwrapped, you or your teammates might not know that and write code that assigns **nil** to one of them.

When in doubt, use a regular optional instead.

6. CHECKING TYPES AND DOWNCASTING

When you work with subtypes, you might have a value or an object for which you only know the supertype.

Subtypes happen when:

- you use subclasses, so the supertype is a superclass;
- you use classes or value types that conform to protocols, in which case the supertype is the protocol.

There are cases though when you want to check whether your value has the specific subtype and access the properties or methods of the subtype when that's the case.

To do that, you have to *downcast* the value/object from its generic supertype to the specific subtype. Swift offers two type cast operators for this: **as!** and its optional version **as?**.

You use the first operator when you are sure of the type of an object, and you want to cast it to a different type.

That's rarely the case, though. Using the **as!** operator is the same as forcing the unwrapping of an optional. If it fails, it causes a runtime exception.

In most cases then we use the **as?** operator. This returns an optional, which will be **nil** if the downcasting fails.

Let's go back to our library example. Libraries nowadays lend not only books but also movies. So we need **Book** and **Movie** types. We will use classes for this example since they provide a cleaner example.

Both books and movies are items our library contains, and have common properties. We summarize these in a **LibraryItem** superclass.

```
class LibraryItem {
     let title: String

     init(title: String) {
          self.title = title
     }
}

class Book: LibraryItem {
     let author: String

     init(title: String, author: String) {
          self.author = author
          super.init(title: title)
     }
}

class Movie: LibraryItem {
     let director: String

     init(title: String, director: String) {
          self.director = director
          super.init(title: title)
     }
}
```

We can put the whole catalog of our library into an array.

Arrays can contain only objects of with the same type. In our case, that type is the **LibraryItem** superclass.

If we want to print the catalog though, we want to know if each item is either a book or a movie, to be able to access the correct properties of each object.

Since we don't know the type of each object in advance, we check it with the **as?** operator. Since this returns an optional, we then use optional binding to check if the downcast succeeded:

```
let catalog = [
     Book(title: "Moby Dick", author: "Herman Melville"),
     Movie(title: "2001: A Space Odissey", director: "Stanley Kubrick")
]

for item in catalog {
     if let book = item as? Book {
          print("Book: \(book.title), written by \(book.author)")
     } else if let movie = item as? Movie {
          print("Movie: \(movie.title), directed by \(movie.director)")
     }
}
// "Book: Moby Dick, written by Herman Melville"
// "Movie: 2001: A Space Odissey, directed by Stanley Kubrick"
```

7. VALUES AND OBJECTS THAT CANNOT BE INITIALIZED

FAILABLE INITIALIZERS

Sometimes, it is useful to have the initialization of a structure, class or enumeration fail. This is helpful to prevent the creation of values or objects with the wrong internal state.

In Swift, you can make an initializer *failable*. That means that initialization might fail and return **nil**.

You make an initializer failable by placing the **?** operator after the **init**, before the parentheses.

As an example, let's assume that we have a structure representing animals. We want to prevent initialization when we pass the wrong species name.

Non-optional parameters already do part of the checking, since they don't allow **nil** values to be passed to the initializer. But in this case, we also want to make sure that the string passed to the initializer is not empty.

```
struct Animal {
     let species: String

     init?(species: String) {
          guard !species.isEmpty else {
               return nil
          }
          self.species = species
     }
}
```

Since this initializer can fail, we need to check the value returned when we create **Animal** value:

```
if let giraffe = Animal(species: "Giraffe") {
     print("A new \(giraffe.species) was born!")
}
// Prints "A new Giraffe was born!"
```

If we want to make the **Animal** type a class, the above code does not work.

In a class, a failable initializer must provide a value for each property before triggering a failure. But the **species** property cannot be initialized until after the check that triggers the failure.

The solution is to make the **species** property an implicitly unwrapped optional:

```
struct Animal {
     let species: String!

     init?(species: String) {
          guard !species.isEmpty else {
               return nil
          }
          self.species = species
     }
}
```

The only difference with the previous code is that here we have a **!** after the **String** type of the **species** property. All the rest stays the same.

8. ADVANCED OPTIONAL TECHNIQUES

ITERATING OVER ARRAYS CONTAINING OPTIONALS

From time to time, we need to work with arrays of optionals. This means that some of the values in a collection might be **nil**.

Often, we don't care about **nil** values, and we only want to use the valid ones.

For example, when we convert an array of strings into integers, some of those strings might not be convertible. The **init(_:)** initializer of **Int** is failable and returns **nil** in those cases. So the resulting array will contain **nil** values for the strings on which the conversion fails.

We can only consider the valid integers using the already familiar optional binding:

```
let strings = ["2", "7", "four", "3", "giraffe", "9", "screwdriver"]
let integers = strings.map { Int($0) }
// integers is now [2, 7, nil, 3, nil, 9, nil] and has type [Int?]

for value in integers {
     if let integer = value {
          print(integer)
     }
}
// Prints 2, 7, 3, 9
```

If you are not familiar with the **map** method of the **Array<T>** type, all it does is take a function and return a new array where the elements are the result of applying that function to each element of the original array.

Here we apply to each element in the **strings** array a function that converts it to an integer. What we get back is an array of **Int?**, which contains **nil** values where the conversion fails.

Using optional binding works, but it's a bit tedious. We have a couple of ways to avoid it and make our code more concise and readable.

The first is to add a **where** clause to the **for** loop:

```
for integer in integers where integer != nil {
     print(integer)
}
```

Swift also supports a more concise way to express this, allowing optional binding in the **for** loop with a **case** statement:

```
for case let integer? in integers {
     print(integer)
}
```

There is another way to filter out all the **nil** values at once, which we will see later.

DOUBLY NESTED OPTIONALS

At this point, you might think that you have a solid grasp of optionals and how to use them. You know what they are and all the techniques Swift offers to deal with them.

So you start using them in your code.

But then something weird happens.

Let's take our array of optionals from the previous section and print it:

```
let strings = ["2", "7", "four", "3", "giraffe", "9", "screwdriver"]
let integers = strings.map { Int($0) }
print(integers)
// [Optional(2), Optional(7), nil, Optional(3), nil, Optional(9), nil]
```

Something a bit odd is already happening here.

As we expect, all the odd strings became **nil**. Other values though are printed as **Optional(2)** instead of just their value.

We know that the type of the **numbers** array is **[Int?]**. Maybe this is just a feature of Swift that tells us when a value is optional?

Let's see what happens when we print the last value in the array alone, which we know is **nil**:

```
print(integers.last)
// Prints "Optional(nil)"
```

Wait, what is going on here?

We expected **nil**, but we got an **Optional(nil)**. What does that even mean? A **nil** that could also be **nil**? It does not make much sense.

To figure out what is going on here let's look at the declaration of the **last** property of the **Array** type in the Swift standard library:

```
public struct Array<Element> {
    ...
    public var last: Element? { get }
    ...
}
```

The type of **last** is an optional **Element**, which is a generic representing the type of the elements in the array. Since our array contains values of type **Int?**, the **last** property returns a value of type **Int??**.

So it seems that calling **last** on our **numbers** array produces a twice optional value. These kinds of optionals are called *doubly nested optionals*. As you can see, they can easily pop up in your code, so you need to be prepared.

Let's try and see if we can also produce them directly:

```
let doublyNested: Int?? = 3
print(doublyNested)
// Prints "Optional(Optional(4))"

let triplyNested: Int??? = 7
print(triplyNested)
// Prints "Optional(Optional(Optional(7)))"
```

Indeed, we can nest optionals inside other optionals as deep as we want. To understand how this is possible and what it means, we have to look at what optionals are under the hood.

HOW OPTIONALS WORK UNDER THE HOOD

At first sight, optionals might seem like some magic performed by the Swift compiler. Through some wizardry (the optional operators) we can use **nil** values as a sentinel for any type.

Most languages do not have this feature. They only provide **nil** values for pointers/references. That's because references are memory addresses and **0** is not a valid one. So, in those languages, **nil** is just a **0**.

This leaves out simple types like integers, floating point numbers, characters, and booleans. For these, **0** is an acceptable value (with booleans, **0** usually means **false**).

Swift, instead, has a different approach.

Optionals are not so obscure as you might think. They are implemented using standard language features that you can also use, which means that we can uncover how they work under the hood without looking at the compiler's code.

At the end of the *Swift Programming Language* guide on swift.org, there is a *Language Reference* section that many usually skip.

That's not a surprise since it's a dry and dull definition of Swift's grammar.

In there, there is a chapter on types, where you can read (emphasis mine):

"The type Optional<Wrapped> is **an enumeration with two cases**, none and some(Wrapped), which are used to represent values that may or may not be present. Any type can be explicitly declared to be (or implicitly converted to) an optional type."

This solves the mystery: optionals just an enumeration with two cases. The **.none** case is actually the **nil** value, while valid values are wrapped inside the **.some(_:)** case.

This explains why, for example, we can't add an **Int** and an **Int?**. The first one is an integer, while the second one has type **Optional<Int>**.

When you look at it this way, it's clear that these are two different types. More in general, any optional type is an entirely different type than its non-optional counterpart.

This allows the compiler to prevent us from using optionals when they are not allowed. Assigning an **Int?** value to an **Int** variable is, for the

compiler, the same as assigning a **String** value to an **Int** variable. The two types don't match.

This is also why we talk about *unwrapping* optionals: the value we need is wrapped inside the **Optional** enumeration. We need to unwrap it before we can use it.

All the operators we have seen in this guide are syntactic sugar for the **Optional** enumeration that makes our life more comfortable. Without them, we can still use optionals. It's just clunkier.

For example, since an optional value is one of the cases in an enumeration, we can match it using a **switch** statement, as we do for any other enumeration:

```
switch factorial(3) {
case let .some(result):
     print("The result of the factorial is \(result)")
case .none:
     print("The factorial has no result")
}
```

This is equal to optional binding.

So you can use optionals even without the syntactic sugar built around the **?**, **??** and **!** operators, and optional binding. But that would get annoying quickly.

Thankfully, the Swift creators took this into account when designing the language.

MAPPING OPTIONALS

We have seen that **Optional<Wrapped>** is a type like any other. And, like all other types, it comes with some functions.

One of these is the **map(_:)** function. You might be familiar with this function from arrays, and I use it in an example at the beginning of this chapter.

If you don't know what it does, here is a short explanation. The **map(_:)** applies a transformation function to all the elements of an array, and returns a new array with the results.

What does it mean to map an optional though? That explanation is clear for arrays, but not so much for optionals.

Let's look at the definition of **map(_:)** for both types (the one for the **Array<Element>** type is defined in the **Collection** protocol).

```
func map<T>(_ transform: (Element) -> T) -> [T] // Arrays

func map<U>(_ transform: (Wrapped) -> U) -> U? // Optionals
```

It's clear that these two methods are very similar.

- On an array containing values of type **Element**, the **map(_:)** method takes a function that transforms values with type **Element** into elements of type **T** and returns an array of **T** elements.

- On an optional containing a value of type **Wrapped**, the **map(_:)** method takes a function that transforms a value with type **Wrapped** into elements of type **U** and returns an optional of **U**.

As you can see, this is practically the same function.

In both cases, the **map(_:)** method:

- is applied to a container containing values of one type;

- takes a function transforming that values of the same type into values of another type;

- returns a new container with the transformed values.

Optionals are also containers. The only difference is that an array contains many elements, while an optional contains only one.

Ok, enough theory.

What are the practical implications of this discovery? How can we use the **map(_:)** function on optionals?

Notice that the function that **map(_:)** takes does not care about optionals. The **transform** parameter has type **(Wrapped) -> U**, where neither **Wrapped** nor **U** are optional types.

We can use **map(_:)** to apply to an optional a function that generally would not take optionals. Using this function, **map(_:)** transforms an optional into another. If the optional contains **nil**, then **map(_:)** returns another **nil**.

This has the advantage that we don't need to unwrap an optional before applying the transform.

Let's take, as an example a simple function that calculates the square of a number.

```
func square(_ number: Int) -> Int {
     return number * number
}
```

Our **square(_:)** function works only with values with type **Int** and does not accept optionals. We can just calculate the square of numbers, and **nil** is not a number.

It might happen though that we have a value of type **Int?** of which we want to calculate the square. Normally, we would have to unwrap the optional first:

```
let optionalInt: Int? = 3
let result: Int?
if let number = optionalInt {
     result = square(number)
} else {
     result = nil
}
```

This pattern of "take an optional and transform it if it is not **nil**" often occurs in Swift code. With **map(_:)** we can avoid unwrapping:

```
let result = optionalInt.map(square)
```

This also works with functions that return optionals — for example, our **factorial(_:)** function. With map, we can calculate the factorial of a number first, and then its square, without unwrapping the intermediate result.

```
let result = factorial(5).map(square)
print(result)
// Prints: Optional(14400)
```

You can chain as many functions as you want using **map(_:)**. Naturally, the last result is an optional, but, you only have to unwrap that one instead of every intermediate result.

FLATMAP

Optionals have another mapping function called **flatMap(_:)**. Its definition is slightly different from the one of **map(_:)**.

```
func flatMap<U>(_ transform: (Wrapped) -> U?) -> U?
```

The only difference here is that the **transform** function returns an optional, while in **map(_:)** that is not the case.

That's a significant difference. In the example in the previous section, chaining the **square(_:)** function to the **factorial(_:)** one, produced an optional.

What if we want to calculate the factorial of that result?

If we use **map(_:)**, we get a doubly-nested optional.

```
let result = factorial(2)
     .map(square)
     .map(factorial)
print(result)
// Prints Optional(Optional(24))
```

That's because **map(_:)** returns an optional of whatever type returned by the **transform** function. Since **factorial(_:)** returns an **Int?**, **map(_:)** returns an **Int??**.

That's where we can use **flatMap(_:)**, which returns the same type as the **transform** function**.**

```
let result = factorial(2)
     .map(square)
     .flatMap(factorial)
print(result)
// Prints Optional(Optional(24))
```

That's why the function is called **flatMap(_:)**. It flattens optionals, preventing deep nesting.

(By the way, the example above works only with the numbers 0, 1 and 2. From 3 on, the result of the last factorial becomes too big to be contained in an **Int** value.)

This opens the door to concepts of functional programming like functors, applicative functors, and monads. These are complex topics (with horrible names) that would need their own guide to explain, so I will leave them out.

Admittedly, you are not going to use this kind of code often in your apps, unless you use advanced techniques on your types like method chaining. These are used by some popular reactive frameworks, which I usually advise to avoid.

But there is one last useful tip for **flatMap(_:)**. Like **map(_:)**, **flatMap(_:)** also works on any other container, including arrays.

You can use **flatMap(_:)** to remove any **nil** value from an array. We had an example of this already, where we used a **for** loop with a **where** clause or pattern matching.

You can use **flatMap(_:)** instead.

```
let strings = ["2", "7", "four", "3", "giraffe", "9", "screwdriver"]
let integers = strings.flatMap { Int($0) }
print(integers)
// [2, 7, 3, 9]
```

You can actually flatten any container — for example, arrays containing other arrays.

```
let nested = [[1 ,2, 3], [4, 5, 6], [7, 8, 9]]
let integers = nested.flatMap { $0 }
print(integers)
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

9. CLOSING NOTES

Even though optionals might be a new concept to some developers, they do not solve a new problem. Representing non-existing values is a problem that has existed in programming for a long time.

Many languages, like Objective-C, solve this problem by adding **nil** values for reference types, like objects. But while Swift has **nil** values too, optionals approach the problem differently.

One immediate benefit is that Swift allows us to use **nil** values with any type and not only references.

But what other benefits do optionals bring to the table?

The first is that, in Swift, it's always clear from a type if a value can be **nil** or not.

In many other languages, there is no such guarantee. Some functions return **nil** values; others never will. Some others might even choose some other value as a sentinel (for example **-1**).

The same goes for parameters. Some function might accept **nil** as a parameter, and some others might not.

In both cases, the signature of a function does not give us any clue. To be sure, you have to inspect the code or read its documentation. Or, as many developers do, guess and hope that it will not cause unexpected problems (not the best strategy, but let's be real. We all do this).

It often happens though that you don't have access to the source code. Documentation is often lacking, if it exists at all, and might not mention nil values. I often found myself wondering whether **nil** values were acceptable or not while reading the documentation for some Objective-C class or function, with no clear answer.

In Swift, this problem does not exist.

If the type of a return value or a parameter is not optional, we are sure that a **nil** value will never be present. To use **nil** values, we have to allow it in the syntax explicitly.

This saves us from a lot of mental baggage we might accumulate while writing code. When I wrote code in other languages, especially in Objective-C, I had to continually keep in mind which values could be **nil** and which ones couldn't.

In Objective-C, calling methods on a **nil** value does nothing, so **nil** values propagate throughout your code base. This makes code shorter, but sometimes **nil** values end up in unexpected places, causing obscure bugs.

This is why Swift has optionals. Swift forces you to deal with **nil** values as soon as they surface in your code. This prevents them from making way into other parts of your code.

You decide where **nil** values can go.

A second advantage optionals have is that the compiler does many needed checks for us.

Even if you try to think about all the possible edge cases, some **nil** value could be able to slip out. We are only humans, and we cannot keep all this information in mind all the time.

We can, of course, try to use assertions to catch these mistakes, but that doesn't go far. If we don't trigger a specific assertion while testing our app, we can still ship bugs.

The compiler, though, can stop us as soon as we make a mistake.

I know that it's annoying, in the beginning, to have the compiler nagging you about optionals. On the other hand, if it didn't, you could cause bugs that would later require more of your time and frustration to fix.

Hopefully, after reading this guide, you will be able to fix any compiler complaints with more robust code quickly.

Swift is a highly opinionated language, and some people will disagree with those opinions. Optionals are one of those.

Coming from Objective-C, I thought, at first, that checking for **nil** values all the time, like in Java, was annoying. In the end, though, I found myself appreciating the value that optionals bring to the table.

They need a bit more thinking when you start using them, but you will get used to them quickly. In my opinion, the benefits outweigh the initial annoyance.

Reading comments and blog posts around the internet, many people seem to hate optionals. I think that most of the times, this comes just from poor understanding.

When you know precisely why some values might be optional and how to deal with them, a lot of the frustration goes away.

You can still bypass the compiler complaints if you want. When you are sure, explicitly unwrapping optionals are there to avoid unnecessary checks. If you know the implications of what you are doing, this is fine, and I do that too, sometimes.

The point here is that when you understand how optionals work and why you have the tools to better reason about your code and make decisions.

Like many other things, this requires understanding the benefits and the pitfalls of your decisions. This allows you to take routes that might seem dangerous at first, but are much safer than they look.

PLEASE SHARE THIS GUIDE

I hope you enjoyed this guide and it helped you improve your understanding of optionals. It took me many hours of work to put it together, but I am happy to offer it to you free of charge.

If you found it useful, please take a moment to share it with someone that could find it useful. In this way, I can dedicate more time into creating more free articles and guides to help you in your iOS development journey.

Think about colleagues and friends that could find this guide useful and send them this link through email, in forums or through a private message, so that they can get this guide, and all the other material I only share with my email list:

matteomanferdini.com/complete-guide-to-swift-optionals

You can also use one of these links to share the guide on your favorite social network.

Share on Facebook

Share on Twitter

Share on LinkedIn