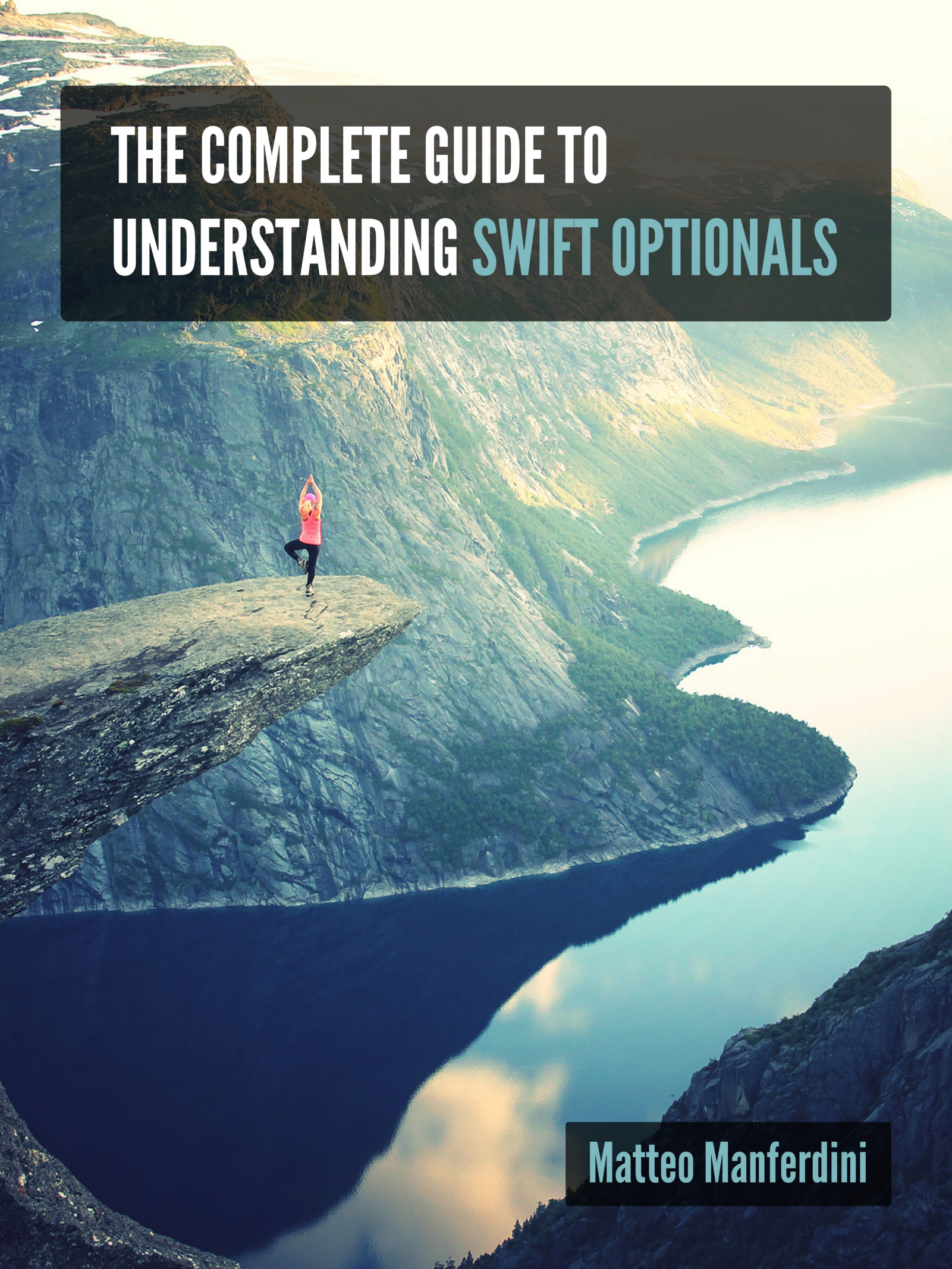
THE COMPLETE GUIDE TO
UNDERSTANDING SWIFT OPTIONALS
Matteo Manferdini

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. It is not a problem shared only by beginner programmers. Many experienced programmers themselves need some time to get used to the concept of optionals, due to their novelty and how differently they behave from nil (or null) values in other languages. I also have to admit that it took me a while too to wrap my mind around them at the beginning.

In my case, which is probably the case for many others, the reason was that I did not take the required time to fully understand what optionals actually are. My first approach was to just try and use them in some project I was working on. That did not go well. Question and exclamation marks would pop up here and there in my code, leaving me puzzled on the reasons behind them. Although I understood a little what they meant (or so I thought), in many places I did not really know why I actually had to write code in a certain way or if it was correct.

Especially if you are coming to programming for the first time, it might be hard to see why optionals exist, what problems they solve and when to use them. Besides this, the different operators cause confusion because they can have different meanings in different context.

I wrote this guide to shed a light on optionals and help you finally understand what they are and how they to use them. We are going to go from the basics to more advanced uses, looking at every case where we could find optionals in Swift. There are many of them, 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 on complete and well defined data, where every operation returns a well defined value. Reality though is quite different, and many times we have to deal with operations that might or might not return a value. It might be that a function cannot calculate a result for some input. It might be that we are looking for something that cannot be found, like a name in a list or a file on a disk. It might be that we reach the end of a sequence, a file or a stream of data coming from the network. It might be that the user of our app does not provide some input where he is expected to.

Let's start with a small simple example, a common programming exercise you find in many textbooks: a function to calculate the factorial of a number. The factorial of a number is simply 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
}
```

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. In fact that error gives a more complicated message in a Swift playground and would cause causes a crash in a real app. The reason is that the range in the **for** loop in the **factorial** function does not work when we feed it a negative number. But this is not the only problem. We cannot fix the function even if we replace that range with something else, because the factorial of negative numbers does not exist in mathematics. We simply have no definition for it. For this reason **factorial** needs to return a special value in case of a negative input, to communicate that the result does not exist.

Programmers resort to many different special values to represent results that do not exist. These values 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 by the mathematical definition of factorial. This would work and indeed has been the choice for many programs written in the past in other languages.

This is not an ideal solution though. If someone else uses this function, how does he know that **-1** means "no result"? Indeed it would not be possible for someone that does not know the definition of the factorial to know whether the result of the function is correct or not. We could write some documentation for the function, but people might not read it and still use it incorrectly. The incorrect value would then propagate to other parts of their program, causing hard to find bugs. This approach is even more problematic when **-1** is actually a valid value for some other function and we have cannot find an invalid one to use as a sentinel.

Many languages use a special value, called **nil** (or **null** in some of them), to represent the absence of a value. In many of those languages, though, a **nil** is possible only for references to objects. Simple types 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. In Swift "no value" is also represented by **nil**. That is our sentinel and is

always going to work, because **nil** is not a valid value for anything. We can now rewrite our factorial function to return no value 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 is not **Int** anymore, but is now **Int?**. This declaration now explicitly states: "the return value might be an integer or a **nil**". This is a key difference with all other languages that also use **nil** values. In those languages you never know if a value could be **nil** or not. You need to read the documentation, if any is provided at all. In Swift this is made clear by the type, so we always know which values could be **nil** and which ones will always be valid.

(A little side note: as a reader suggested, the factorial function could check its input through an assertion, making sure that a negative value is never passed. This is because the factorial is undefined for negative parameters, so it should not accept them at all and leave the checking of the input to the caller. This would also be a valid approach, but since this is a simple example to understand optionals, I prefer it this way. In the end what is the right approach here is a matter of opinion).

3. USING OPTIONAL VALUES

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

In Swift, the developer is not the only one to know which values are optional by looking at the declaration of a type. The compiler knows this too and can make a series of checks that are not possible in other languages that don't have optionals. This means that the compiler can now make sure we never put a **nil** where we are not supposed to. It simply does not allow us to do it, giving us an error as soon as we try to run such code (or even better in Xcode, in real time as we type our code). This is a good thing: it is always good to delegate to the compiler as many checks as possible, freeing ourselves from them. This helps us making less mistakes. Humans miss things all the time, while the compiler is not going to let any **nil** through places where it is not expected.

Unfortunately, this might also be frustrating if we don't understand what is going on. The compiler might complain about something that to looks correct to you. But the compiler is the one who is right in the end, so you need to understand what he is trying to tell you.

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

FORCED UNWRAPPING

Let's start by trying to use our newly made factorial function:

```
factorial(3) + 7
// Value of optional type 'Int?' not unwrapped; did you mean to use '!' or '?'?
```

Unfortunately, this fails. The compiler seems to be complaining about something but it's not clear what. Why cannot we just 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 here is that we are trying to add two things that for the compiler have two distinct types. Remember, as I said, the compiler is not going to let a **nil** slip through anywhere. The result of the factorial function is not of type **Int** but of type **Int?**. In our mind we might think that the **?** just says "there might be a **nil** value", but to the compiler this tells that the type is different. We will see later why these two types are different, when we will look into more advanced used of optionals. For now, and really for most of the time, you just need to keep in mind that an optional type is different that the original type it comes from. The compiler will stop you when you try to treat them as equal.

Since for the compiler these are two distinct types, it does not allow us to add them, but we know that in this case it should be possible, because we know the result of the factorial. The error message from the compiler is actually giving us a hint. To be able to perform this addition we have to transform the optional value into a non optional one. We do this using the *forced unwrapping operator* **!**

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

The forced unwrapping operator tells the compiler: "I know that this optional has a value, so use it." This works here, but is actually dangerous most of the times. While in this simple example we know for sure that the result will not be **nil**, most of the time we actually won't. If you use forced unwrapping on a **nil**, this will cause a runtime exception and a crash in your app.

So, in your mind you have to think about forced unwrapping like this: "I am so sure that this value will not be **nil** that I want my app to crash if it is not". There are some cases where this might be true, as we will see, but most of the time you will not be so sure. So we need a different approach.

OPTIONAL BINDING

Since forced unwrapping on 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 just make sure that the value is not **nil** before the forced unwrapping. This works, but it is a bit too verbose since it is something we need to do quite often. Moreover, if we have to use the **result** contant more than once in the body of the **if**, we have to place a **!** after each use, making it less convenient.

For this reason **if** statements support *optional binding*, which allows to check an optional and extract its value as part of a single instruction. Moreover, in this way the forced unwrapping of the optional inside of the body of the statement is not necessary anymore, relieving us from using **!** every time.

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

This also works with multiple bindings, in case we need to check multiple optionals:

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

Sometimes, instead of checking the results of some functions, you need to check optionals that are already stored in a constant or variable. A problem here is to find a new name for the binding. Finding good names for variables is always a problem, and it becomes more so if you have to do it twice for the same value. Luckily you don't have to. The optional binding creates new constants that shadow the previous ones in the **if** body, so you can just use 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
```

As you see, here I didn't have to come up with new names. I just named the bindings the same as the original constants.

We can add also a **where** clause to each binding, if we want to add additional checks to values of the bindings themselves before we use them. For example, lets assume that we want to use only the factorials from the previous example only if they are all even. I wrote a little **isEven** function to help, since we have to repeat the check three times:

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

On a tangential note, the above example will almost always execute the body of the **if** because factorials are always even. With the exception of 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 with **while** statements. A **while** loop will continue while the binding produces a valid value and will stop when a **nil** is found. 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. We would write it like this:

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

NIL COALESCING

It happens often that we want to use a default value when an optional does not have a valid one. This can be of course written in the form of an optional binding, in which the else branch assigns the default value we want to use:

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

There is a more succinct and practical way of writing this, though: the *nil coalescing operator* **??**, that accomplishes the same in a single line of code.

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

This simply means "if the factorial is **nil**, use **0**". If an optional has a value, **??** returns it, otherwise it returns the second value. One nice feature of this operator is that it can be chained multiple times. So if we have more than one optional and we want to take the first valid value we find in a certain order, we can chain multiple nil coalescing operators in the same instruction:

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

It is also possible to omit the default value at the end, in which case the result will be **nil** in case no instruction has 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 your name, your age and your city and creates a greeting message from them. The function only works if you provide all three of them. If any is missing, it produces an error message instead.

```
func messageForName(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 didn't give me all information I need!"
   }
}
print(messageForName("Matteo", age: 35, city: "Amsterdam"))
// Hi Matteo, I see you are 35 years old. Nice age. And how is it living in
   Amsterdam?
```

This function though does not tell us which information is missing if we forget to pass it, so let's write a better version that does. We want to generate the message using the information that is available up to that point and return in the message for the first parameter that is missing. In this case we cannot use multiple bindings together anymore, since we have to check different conditions to generate different messages. We have to use separate bindings. Let's have a first shot at it.

```
func messageForName(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 "\(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 has the problem that is not very readable, because of the multiple nesting of the **if** statements. This gets worse when we add more checks, to the point that many developers call this pattern *the pyramid of doom*.

The problem here is that the part that actually does the interesting work (assembling the actual full message) is lost inside the pyramid. All the rest of the code is just error checking. This inside part is called the *happy path* or *golden path*, because it is the path where no errors happened that leads to the actual business logic. This is what we usually care about while the other code is there only for error handling. In our example the actual business logic is just one line long, but usually it is more complicated and we don't want it to be lost within other code.

This is a very common pattern that occurs again and again in real code and is often made worse by the fact that in the intermediate steps we actually have to do something with the partial result. This means that the business logic gets split across the statements, making it interspersed with the error checking code, so it becomes even more lost.

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
}
```

Those "do something" (in bold) are our happy path, which we want to be able to follow easily. You can also notice that the error handling for each condition is separated from the actual condition that checks it. They even happen in reverse order, so the last error handling actually refers to the first check, putting them quite away from each other.

We want all the actual business logic to be easy to spot and the error handling to be kept close to the actual checks. We do so by checking for errors in our if statements instead of checking for success and returning from the function as soon as possible. This keeps the errors close to the checks and keeps the logic outside of the if statements, pushing it to the left where it is 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 following what we have just seen:

```
func messageForName(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 “\(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 has still a little annoyance. We lost the optional bindings and we actually have to resort to forced unwrapping every time. If we have to use these values more than once this gets even worse. This is why Swift 2.0 introduced *guard* statements. Guard statements allow us to check for **nil** and return early while at the same time bind the eventual result to a constant.

```
func messageForName(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 “\(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)?"
}
```

As you can see, we don’t need to force unwrap our constants anymore after the guard statements.

Guard statements are intended for early exit only, so they come with a restriction: they must contain an exit statement, either **return** when they are used to exit from a function, or **continue** or **break** when they are used inside loops.

4. OPTIONALS IN STRUCTURES AND CLASSES

OPTIONAL CHAINING ON PROPERTIES

Optionals are not only used as return values in functions, but can also be used for variables and properties of structures and classes. We use them to indicate that a variable or property might not hold a value at a given time. As an example, let's create a structure to hold a people information for a city library. Every person has an associated library card that holds the list of books the person borrows.

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

class LibraryCard {
   var numberOfBorrowedBooks = 0
}
```

The **libraryCard** property of **Person** is an optional, because not everybody has a subscription to the library and they might be in the database for different reasons (maybe they are employees who don't read books). We now want to print how many books a person has borrowed in the past. To get to the **numberOfBorrowedBooks** property, though, we have to traverse the optional **libraryCard** property of **Person**, which might be **nil**. We can of course use an optional binding on **libraryCard** like we saw in the previous chapter, and if the property is not **nil**, access the the **numberOfBorrowedBooks** property of the unwrapped value. This becomes verbose and tedious pretty soon, though. For this reason, Swift has an alternative mechanism that allows us to directly go through the optional to the value we want, without the use of an optional binding. This mechanism is called *optional chaining* and is done by using a **?** after the 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 **?** has a different meaning that it had when we put it after a type. While the latter meant “this value might not exist”, used at the end of a variable or a property, a **?** means instead “If it has a value, access the specified property, otherwise return **nil**”. This is one of the things that make optionals confusing: the **?** has two different meanings, depending where you use it.

So if **libraryCard** is not **nil** we access its **numberOfBorrowedBooks** property directly. This is different from using a **!**, which would force the optional to unwrap and cause a crash in case it finds a **nil**.

Pay attention that since accessing the **numberOfBorrowedBooks** property might fail, the value returned by the optional chaining is still an optional, even if the final property we are accessing is not. In this case, the **numberOfBorrowedBooks** property has type **Int**, but the value returned by **me.libraryCard?.numberOfBorrowedBooks** has type **Int?** because we are traversing an optional that might be **nil**. This is why in the example we still used an optional binding on the result of the chaining.

It is also possible to try to set a property through optional chaining. While an assignment normally would not have a return type, attempting to set a property through an 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 now expand our classes to include a list of borrowed books:

```
class LibraryCard {
   var borrowedBooks = [Book]()
   var numberOfBorrowedBooks: Int {
      get {
         return borrowedBooks.count
      }
   }

   func addBook(book: Book) {
      borrowedBooks.append(book)
   }
}

class Book {
   let title: String

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

To get to the **addBook**() method, we still have to traverse the **libraryCard** property of **Person**, which is optional. Like in the case of the assignment, the **addBook**() method has a **Void** return type, but through the optional chaining this will become **Void?**, allowing us to check for the success of the method call.

```
let john = Person()
let book = Book(title: "Moby Dick")
if (john.libraryCard?.addBook(book)) != nil {
   print("John has borrowed \(book.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 multiple levels of optionals. Let's add a property to **LibraryCard** to retrieve the last book a person has borrowed:

```
class LibraryCard {
   var borrowedBooks = [Book]()
   var numberOfBorrowedBooks: Int {
      get {
         return borrowedBooks.count
      }
   }

   var lastBorrowedBook: Book? {
      return borrowedBooks.last
   }

   func addBook(book: Book) {
      borrowedBooks.append(book)
   }
}
```

We can now use multiple levels of chaining to fetch the title of the last book John borrowed:

```
var john = Person()
john.libraryCard = LibraryCard()
let book = Book(title: "Moby Dick")
john.libraryCard?.addBook(book)
if let bookTitle = john.libraryCard?.lastBorrowedBook?.title {
   print("The last book John has borrowed is \(book.title)")
} else {
   print("John has not borrowed any books")
}
// prints "The last book John has borrowed is Moby Dick"
```

As we have already seen, even if the type returned by a property or a method is not optional, optional chaining will make it so. If the 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, optional chaining can be used on subscripts of arrays or dictionaries as well. Everything that has been said for properties and methods applies to subscripts too.

OPTIONAL CHAINING ON OPTIONAL PROTOCOL REQUIREMENTS

Until now we have seen the use of optionals for things that might not return a value. Optional chaining can be also used for properties and methods in protocols that are not a requirement and thus might not exist at all. This is the case with *optional protocol requirements*. When defining a protocol, it is possible to specify some methods or properties as optional. This means that the classes that conform to the protocol are not required to implement them and might ignore them. Thus, when you are accessing these optional properties or methods of an instance that conforms to such protocol, you cannot be sure that this instance has an implementation for them. For this reason the use of optional chaining is required.

Let's assume we are making a role playing game game which includes wizards. A wizard can sometimes have a companion creature, which can be a simple animal or a more powerful magical creature. Some creatures can give the wizard an extra spell to cast. We can express this with a protocol:

```
class Spell: NSObject {

}

@objc protocol Familiar {
   optional func familiarSpell() -> Spell
}
```

(Note: the protocol is marked with the **@objc** keyword because this is the only way to declare optional methods or properties in protocols in the current implementation of Swift. Also, the **Spell** class inherits from **NSObject** instead of being a pure Swift class because the return value of optional protocol requirements needs to be Objective-C compatible

as well. Hopefully in a future version Swift will have a native way to express this instead of relying on Objective-C bridging).

We can now implement a **Wizard** class, with a method returning the available spells to cast:

```
class Wizard {
   var familiar: Familiar?
   var spells = [Spell]()
   func availableSpells() -> [Spell] {
      guard let familiarSpell = familiar?.familiarSpell?() else {
         return spells
      }
      return spells + [familiarSpell]
   }
}
```

Pay attention to this example because here optional chaining is used in two different ways. The first one we have already seen. A **Wizard** might not have a **familiar**, so the **familiar** property is an optional. We traverse this property through optional chaining. The second one instead checks if the **familiarSpell**() method is present. Notice that the **?** here is placed before the parentheses in the method call and not after. The latter would check if the returned value existed, while in this case we are checking for the existence of the method itself.

5. AVOIDING THE CHECKS FOR NIL

IMPLICITLY UNWRAPPED OPTIONALS

As we have seen, optionals add to our code a lot of question and exclamation marks, if statements, guards and optional bindings, which sometimes make code harder to read. This is one of the trade offs of optionals we have to live with. There are some specific cases though where we know that after an optional has a value, it will never be **nil** again. In these cases all the checks we normally need are not required. We know that this specific optional will always have a value from some point on, which makes it not really an optional anymore.

For these cases, we can declare an optional as *implicitly unwrapped*. We do so by placing a **!** after the type of the optional instead of **?**. This allows the optional to start with a **nil** value at a moment in which a valid value cannot be provided yet, while allowing us to use it in the rest of our code as if it was not an optional. This means no unwrapping, bindings, ifs or guards are necessary even if this is an optional. This means that the optional is used as if it was not an optional. The compiler won't make any checks on it and won't bother you.

Pay attention to two things here: first, this is another case where an operator gets another meaning. Using a **!** after an optional means forced unwrapping. Using it after a type means implicit unwrapping.

The second thing is that implicitly unwrapped optionals are also dangerous. This is because they are still optionals, and if we want we can still assign a **nil** to them, without the compiler warning us. Since we use an implicitly unwrapped optional without any checks, if we find a **nil** without expecting it, this has the same effect as forced unwrapping: a crash. So pay attention.

There are a few cases though where they are still useful and sometimes even needed.

UNOWNED REFERENCES IN REFERENCE CYCLES

When two objects hold a reference to each other, under ARC we have to avoid creating strong reference cycles, or the two objects will never be removed from memory. There are three specific cases in which these cycles can exist.

1. In the first case the two classes can both have a **nil** reference to the other one. This case is the easy one and is solved by declaring both the references as optionals and marking one of them as **weak** to avoid a strong reference cycle.

   Let's see an example to make this clearer. A **Person** and an **Apartment** objects can have a reference cycle between themselves, since a person can own an apartment and an apartment can have a tenant. But they can also exist independently from each other, since a person might be homeless and an apartment might be empty. Here we can mark the **tenant** property of **Apartment** as weak to avoid the strong reference cycle.

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

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

2. In the second case one of the two references always needs to have a value and cannot be **nil**, because the existence of the referencing object depends on the existence of the referenced one. Since this property is not an optional, it cannot be marked as **weak**, because **weak** references need to be set to **nil** by ARC when the referenced object is removed from memory. On the other hand, the reference in the other object cannot be **weak** either, because it is needed to keep the dependent object in memory. This case is solved in Swift by making the first reference optional and marking the second reference as **unowned**. Unowned references behave like **weak** references from the point of view of ARC, but are not op-

tional, so they are not allowed it to be set to **nil**.

For example, a customer of a bank might have or not have a credit card, so this property can be an optional. It cannot be weak though, because we don't want a credit card object, if it exists, to disappear from memory. A credit card, though, needs to always have an associated customer, so it cannot be optional. As such, it cannot be marked as **weak**, so it needs to be **unowned**, otherwise it will create a strong reference cycle.

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

class CreditCard {
   unowned let owner: Customer

   init(owner: Customer) {
      self.owner = owner
   }
}
```

3. The remaining case is the trickiest one and is the case in which both references always need to have a value. This means that one of the two objects needs to be created in the initializer of the other one. Here the first object passes itself as a parameter to the initializer of the second one. This guarantees that both references are initialized properly. Here comes a problem, though: Swift does not allow you to reference **self** in an initializer until the object 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, and as I just said, this is forbidden.

   This is where implicitly unwrapped optionals come to the rescue. The first reference is made an implicitly unwrapped optional, which allows it to be temporarily initialized to **nil**, while the second instance gets created.

   Let's see an example: a **Country** must have a **capital** and a **City** must have a **Country** to which it belongs. So none of these two

classes is allowed to have a **nil** reference to the other one. In the initializer of **Country** we create an instance of **City** for the capital. This instance of **City** needs a **country** passed as a parameter to its initializer, so **Country** needs to pass **self** when creating it. But to be able to pass **self** it needs to be completely initialized, which is still not because it still does not have a **capital**. So **capital** must be an implicitly unwrapped optional to allow **nil** to be temporarily assigned to it, while an instance of **City** for the **capital** gets created.

```
class Country {
   var capitalCity: City!

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

class City {
   unowned let country: Country

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

INITIALIZATION OF IBOUTLETS

Another common scenario for implicitly unwrapped optionals are outlets to objects coming from a storyboard or a xib file. When views or view controllers are initialized from an interface builder file, their outlets cannot be connected yet. They will only be connected after initialization is completed, so they need to be nil in the meanwhile. When any other code in the class is called after initialization, though, these outlets are guaranteed to be connected. This is why **IBOutlets** are usually declared as implicitly unwrapped optionals.

FAILABLE INITIALIZERS

When the initialization of an object fails, it should return **nil** as soon as possible. By definition, though, at that point not all the properties will have been initialized and Swift does not allow an **init** method to return until all properties are initialized. Implicitly unwrapped optionals solve

this problem too. 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 only when you are sure of what you are doing. Keep in mind that they are still optionals even if you don't use them as such in your code, so they might be assigned a **nil** at any point without the compiler checking it. This **nil** will get unwrapped when you use the optional and cause a crash. As said here the danger lies in the fact that implicitly unwrapped optionals bypass all the compiler checks. This means you are not forced to check their content before using them. They are still optionals though and the compiler will allow you to assign **nil** to them without any complaint. For this reason they have a high potential of generating runtime exceptions and crashes.

As I said, they are intended for specific cases where you are sure that the value will always exist after initialization. When you are sure of this, implicitly unwrapped optionals are very convenient since you don't have to check them every time you use them. This is not always so easy to know, though. It is easy to fall into the temptation to use implicitly unwrapped optionals for cases that do not fall under this rule, since they might seem a very convenient for stopping the compiler from complaining. Remember that the compiler checks are there for a reason. It should not be your goal to simply silence them. Doing this defeats the whole point of optionals and puts your code in danger. Even when you think you will remember about some implicitly unwrapped optional you declared, you or your teammates will probably forget without the compiler's help. So, when in doubt, use a normal optional instead.

6. CHECKING THE TYPE OF AN OBJECT: DOWNCASTING

When we work with classes that inherit from a superclass, we might happen to have an instance of which we know only the most generic type. In some cases though we might want to access properties or methods of a specific subclass, if that object is actually an instance of such subclass. To be able to do this, we have to *downcast* an instance from its generic superclass to the more specific subclass. Swift offers two type cast operators for this: **as!** and its optional version **as?**. The first one is used when we are already sure of the type of an object and we want to just cast it to a different type. Many times, though, we do not have this information. Using the **as!** operator is the same as forcing the unwrapping of an optional, if it fails, it causes a runtime exception. For those cases we use **as?**, which returns an optional, which will be **nil** in case the cast fails.

As an example, let's go back to our library. Libraries nowadays rent not only books but also movies, so we have to make a distinction between the two. We can have a generic **LibraryItem** superclass that contains the common attributes for what our library contains and two more specific subclasses for each item type: **Book** and **Movie**.

```
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 then create an array that contains the catalog of our library. This will be an array of **LibraryItems**. When we want to print this catalog though, we have want to know if each item is either a book or a movie, to be able to access the correct properties of the object. Since we don't know the type of each object in advance, we check it with the **as?** operator, which returns an optional. We need to check this optional to see if the cast 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. WHEN AN INSTANCE CANNOT BE INITIALIZED

FAILABLE INITIALIZERS

Sometimes it is useful to have the initialization of a structure, class or enumeration fail to prevent the creation of instances with the wrong parameters. Swift offers the opportunity to mark an initializer as *failable*, meaning that initialization of a new instance might fail and return **nil** instead of a valid value. We declare that an intializer can fail by placing a **?** after the **init** keyword for the initializer, before the method parentheses.

As an example, let's assume that we have a structure that represent animals, and we want to prevent that its instances are initialized without a valid species. A non optional parameter in the initializer already does part of this check, not allowing **nil** to be passed at initialization time, but we also want to make sure that the string passed to the initializer is not an empty string (we could of course add more checks for this):

```
struct Animal {
   let species: String

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

Since this initializer could fail, we will need a check when we create a new instance of **Animal**:

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

If we want to define **Animal** as a class instead of a structure, this code does not work anymore and we have to make one little change. 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
   }
}
```

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

8. ADVANCED OPTIONAL TECHNIQUES

ITERATING OVER OPTIONALS

It happens from time to time that we need to work with arrays of optionals. This means that some of the values in an array might be **nil**. Most of the times though, we don't care about those and only want to use the valid values. For example, if we try to convert an array of strings into integers, some of those strings might not be convertible into integers. For this reason the **Int.init**(**String**) initializer is failable and might return **nil** if the string does not represent an integer. So our array will contain **nil** values where this conversion fails.

To then take only the converted integers, we can, for example, use the already well known optional binding:

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

for integer in integers {
   if let integer = integer {
      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. So here we are applying to **strings** a little function that converts a string to an integer. What we get back is an array of integers, or **nil**s where the conversion fails.

There are a couple of ways to remove the binding from inside of the loop and make this code more succint 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 also a way to filter out all the **nil**s at once, which we will see in a moment.

DOUBLY NESTED OPTIONALS

At this point, you might think that you have a solid grasp of optionals and how to use them. You understand what optionals mean 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 the last value of the array is a **nil**. Other values though are printed as **Optional**(**2**) instead of just their value. This is not really a surprise after all, we know already 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(numbers.last)
// Prints "Optional(nil)"
```

Wait, what is going on here? We expected **nil**, but we actually got an **Optional**(**nil**). What does that even mean? A **nil** that could also be **nil**? It does not make much sense, does it? To figure out what is going on here let's look at how the **last** property of **Array** is declared in the Swift standard library:

```
var last: T? { get }
```

The type of **last** is the optional of the type **T** contained in the array. Since our array contains values of type **Int?**, the **last** property returns a value of type **Int??**.

So it seems that the result of **last** on our **numbers** array produces a value that is optional twice. These kind of optionals are called *doubly nested optionals*, and as you can see they can easily pop up in our code, so we need to be prepared to handle them. Let's try and see if we can also produce them ourselves directly:

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

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

Indeed it seems that 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 finally look at what optionals really are under the hood.

HOW OPTIONALS WORK UNDER THE HOOD

At a first sight, optionals might seem like some magic the compiler performs to allow us to use **nil** as a sentinel value with any type, even where they would not normally be supported in other languages, for example for simple types like **Int**. Many languages have **nil** values as well and get away with using **0** as a value for **nil**. They can do so only for reference types though, since references are usually numeric memo-

ry addresses and **0** is not a valid one. This of course leaves out simple types like integers, floating point numbers, characters and booleans, where the value **0** is an acceptable value (also for booleans, where **0** usually means **false**). For this reason Swift has a very different approach to optionals.

It turns out that optionals are not so obscure as you might think. They are implemented using standard language features we can also use, which means we can understand how they work under the hood.

At the end of the Swift Programming Language guide from Apple, there is a chapter on the language reference, which people usually skip (me included). In that chapter, the section dedicated to the **Optional** Type states (emphasis mine):

"The type Optional<T> ***is an enumeration with two cases****, None and Some(T), 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 are in reality just an enumeration that has two cases, one for when the value exists and one for the absence of the value. This explains why we cannot, for example, add an **Int** and an **Int?**: the first one is an integer, while the second one is an enumeration, which is a clear type mismatch. More in general, this makes any optional a completely different type compared to its non optional counterpart. This allows the compiler to check if we are putting an optional where the type is not. Again, these are two different types. Assigning an **Int?** to an **Int** variable is exactly the same as assigning a **String** to an **Int** variable. The types don't match. This is also why we talk about *unwrapping* an optional: the value we need is wrapped inside an enumeration, so we have to extract it before using it.

The various operators we have seen in this guide are just syntactic sugar Swift adds around this enumeration to make our life easier. Without it we could still use optionals, albeit in a more clunky way. For example, since optionals are instances of an enumeration, we can match them using a **switch** statement like 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 equivalent to optional binding. So we could use optionals even if Swift didn't have all the syntactic sugar built around the **?**, **??** and **!** operators. But, as you can see, that would be more verbose and become really annoying (as it indeed happens in other languages like Haskel, from which Swift took the idea of optionals as enumerations). So, thankfully, the Swift creators took this into account when designing the language.

MAPPING OPTIONALS

We have seen that **Optional<T>** under the hood is just another type, so it is natural that this type comes with some functions attached. One of these is the **map** function. You might be familiar with this function from arrays and we have seen an example of it at the beginning of this chapter. If you are not, in short **map** applies a transformation function to all the elements of an array and returns an array with the results.

What does it mean to map an optional though? We can clearly see what it means for arrays, but for optionals it is less clear. Let's look at the definition of **map** for both types (for **Array<T>** the **map** function is defined in the **CollectionType** protocol).

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

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

In these definition, **T** is the type contained in the array or the optional, while **U** is another type that might or might not be the same as **T**. You might know already that in Swift a method of a type is nothing else than a closure that takes the instance it is called on as the first parameter. So, let's rewrite these definitions according to this, to make them more understandable.

```
func map<T, U> (array: [T], transform: T -> U) -> [U] // Arrays

func map<T, U> (optional: T?, transform: T -> U) -> U? // Optionals
```

You probably already noticed yourself that these two function definitions look almost the same. In the first definition, **map** takes an array containing elements of type **T** and a function that transforms elements of type **T** into elements of type **U**, and returns and array of **U** elements. In the second definition, **map** takes an optional containing an element of type **T** and a function that transforms elements of type **T** into elements of type **U**, and returns an optional of **U**.

So, in both cases, **map** takes a container with **T** inside and a function to transform **T** into **U**, and returns that container with **U** inside. We have seen in the previous section how optionals are containers themselves, being just an enumeration. The only difference is that arrays contain multiple elements, while optionals contain only one.

Ok, enough theory. What are the practical implications for us here? What can we use the **map** function on optionals for?

The **map** function on an optional applies the function it receives as parameter to the optional when the optional has a value, and returns the result of that function. If the optional is **nil**, then **map** returns **nil**.

Notice that the function that **map** takes as a parameter is a normal function that does not need to care about optionals. The **transform** parameter of **map** is of type **T -> U**, where neither **T** nor **U** are optional types. The advantage of using **map** then is that we don't need to check the optional for a **nil** before applying the transform and we don't need unwrapping: **map** does everything itself.

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

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

Our **square** function is not concerned with **nil** values, because the square is defined only for numbers and **nil** is not a number. It might happen though that we have an optional integer to which we want to apply this **square** function. Normally, we would have to use optional binding:

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

This patter of "take an optional and transform it if it is not **nil**" is actually quite common in Swift code. With **map** we can avoid the binding:

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

So, practically speaking, the **map** function allows us to apply to optionals any function that would not accept optionals in its parameters.

FLATMAP

Optionals have another mapping function called **flatMap**. Its definition is slightly different from the definition of **map**.

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

Or, again, including the first parameter of the closure to make it easier to interpret:

```
func flatMap<T, U> (optional: T?, transform: T -> U?) -> U?
```

The only difference here is that the **transform** function returns an optional, while for **map** this was not the case. So, for example, we can

now apply to an optional our **factorial** function, which as you remember that returns an **Int?**:

```
let optionalInt: Int? = 3
let result = optionalInt.flatMap(factorial)
```

We don't have to stop here: we can actually concatenate as many **map** and **flatMap** calls as we want, since they all return an optional. For example, we can first calculate the factorial of an optional integer and then its square:

```
let optionalInt: Int? = 3
let result = optionalInt
   .flatMap(factorial)
   .map(square)
// result is 36
```

This opens the door to concepts of functional programming like functors, applicative functors and monads. These are complex topics (with horrible names) that would require their own guide to explain, so I will leave that out. Still they introduce interesting concepts that might be worth exploring. To just give you an idea, let's say that we want to make a function for what we did in the last example: this function takes an **Int?** and first calculates its factorial. Then calculates the square of the result, if this is not **nil** and returns it.

Let's write this function as we would normally, without using **map** or **flatMap**. It would probably look like this:

```
func factorialAndSquare(optionalInt: Int?) -> Int? {
   guard let number = optionalInt else {
      return nil
   }
   guard let result = factorial(number) else {
      return nil
   }
   return square(result)
}
```

This is much more verbose, but as we have seen in the section about the happy path, a very common pattern that happens all the time. What combining **map** and **flatMap** allows us to do is to actually replace this pattern with a more readable chaining of functions.

As I said it is a bit more complicated than this. Usually what happens is that the **guard** statements are interleaved with other instructions that don't really produce an optional or a result at all. To make this style of programming possible in those cases too we would need to define more functions or operators, which as I said are outside the scope of this guide. But you can already use **map** and **flatMap** in the way I just showed you, if you have the chance.

As a last tip, arrays also have their version of **flatMap**. To give you a very practical use, this function can be useful when simply applying **map** to an array might produce an array containing **nil** values, but we are only interested in the non **nil** ones (which is usually the case). We have seen some examples of it already. Using **flatMap** instead of **map** we can directly get back an array of non optional values, where the **nil**s have been filtered out.

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

9. CLOSING NOTES

Although optionals are a new concept for many developers, they do not solve a new problem. Representing non existing values is a problem that has existed for long time. Many languages, like Objective-C, solve this problem by using **nil** values for references only. Although we still use **nil** values in Swift, optionals approach the problem in a different way. Apart from the fact that Swift makes it possible to use **nil** values with simple types, what benefits do optionals bring to the table?

One of the first advantages of optionals is that in Swift it is always clear from the type of a value if it can be **nil** or not.

In many other languages there is no such a guarantee. Some functions might return **nil** values, others never will, some others might use some other sentinel value (for example **-1**). Some function might accept **nil** as a parameter, some others might not. This is never clear from just the declaration, though. To know for sure you have to check the code or the documentation, or simply guess and hope that it will not cause unexpected problems (not the best strategy, I would say). Many times unfortunately you don't have access to the source and the documentation might not mention it or might even not exist at all. I have found myself many times reading some documentation, even from Apple, wondering whether **nil** values were acceptable or not, with no clear answer.

In Swift there is no such a problem. If the type of a return value or a parameter is not declared as optional, we are sure that a **nil** value will never be present. To be able to use **nil** we have to explicitly allow it with an optional. This saves us from a lot of mental baggage that might accumulate in our mind while writing code. When we write in other languages, we have to keep in mind all the time which values could be **nil** or not. In Objective-C for example, **nil** values propagate throughout our code base, since sending messages to **nil** does not cause any problem (but generates another **nil**). In this way though, **nil**s end up in unexpected places and sometimes cause obscure bugs. This is why

Apple added optionals to Swift. Swift forces us to deal with **nil** values as soon as they surface in our code, making us think about them every time they appear. This makes sure they will not make their way into other parts of our code.

A second advantage of optionals is that the compiler can do many of the needed checks for us. Many times, even if we try to think about all the possible edge cases for our code, 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 guard ourselves with assertions, but they go only that far. The compiler, though, can stop us as soon as we make a mistake. I know it can feel annoying at the beginning to have the compiler complaining about some optional that is assigned to a non optional value. But if you think about it, this could have caused a very hard to track bug later that would have required, in the end, more of your time and frustration to fix. Hopefully after reading this guide you will be able to quickly fix the compiler complaints with some more robust code.

Swift is a highly opinionated language and some people will disagree with those opinions. Optionals are one of those. Coming from Objective-C, I myself thought at first that it would have been annoying to check for **nil** values all the time, like in Java. In the end, though, I find myself appreciating the value that optionals offer. They need a bit more though beforehand on the part of the developer, but you can get used to them pretty quickly. In my opinion the benefits outweigh the initial annoyance.

Reading comments and blog posts, many people seem to hate optionals, but I think most of the times this hate comes just from poor understanding. When you know exactly 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, by explicitly unwrapping optionals without checking the value. 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 take decisions. As in many things, this requires understanding the benefits and the pit-

falls of such decisions. This understanding allows you to take routes that might seem dangerous at first, but are much safer than they look.

I hope you enjoyed this guide and found it useful to improve your understanding of Swift optionals. It took me many hours of work to put this material together.

If you know someone that can benefit from this free guide, please feel free to send them this link:

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