iOS TRAINING

PREPARED BY FARHAJ AHMED

OPTIONALS

All the variables and constants you've dealt with so far have had concrete values. When you had a string variable, like `var name`, it had a string value associated with it, like `"Matt Galloway"`. It could have been an empty string, like `""`, but nevertheless, there was a value to which you could refer.

That's one of the built-in safety features of Swift: If the type says `Int` or `String`, then there's an actual integer or string there, guaranteed.

This chapter will introduce you to the concept of **optionals**, a special Swift type that can represent not just a value, but also the absence of a value. By the end of this chapter, you'll know why you need optionals and how to use them safely.

Introducing nil

Sometimes, it's useful to have a value that represents nothing at all. Imagine a scenario where you need to refer to a person's identifying information; you want to store the person's name, age and occupation. Name and age are both things that must have a value — everyone has them. But not everyone is employed, so the absence of a value for occupation is something you need to be able to handle.

Without knowing about optionals, this is how you might represent the person's name, age and occupation:

```
var name = "Matt Galloway"
var age = 30
var occupation = "Software Developer & Author"
```

OPTIONALS

Sentinel values

A valid value that represents a special condition such as the absence of a value is known as a **sentinel value**. That's what your empty string would be in the previous example.

Let's look at another example. Say your code requests something from a server, and you use a variable to store any returned error code:

```
var errorCode = 0
```

In the success case, you represent the lack of an error with a zero. That means `0` is a sentinel value.

Just like the empty string for occupation, this works, but it's potentially confusing for the programmer. `0` might actually be a valid error code — or could be in the future, if the server changed how it responded. Either way, you can't be completely confident that the server didn't return an error.

In these two examples, it would be much better if there were a special type that could represent the absence of a value. It would then be explicit when a value exists and when one doesn't.

Nil is the name given to the absence of a value, and you're about to see how Swift incorporates this concept directly into the language in a rather elegant way.

Some other programming languages simply use sentinel values. Some, like Objective-C, have the concept of `nil`, but it is merely a synonym for zero. It is just another sentinel value.

Swift introduces a whole new type, **optional**, that handles the possibility a value could be `nil`. If you're handling a non-optional type, then you're guaranteed to have a value and don't need to worry about the existence of a valid value. Similarly, if you are using an optional type then you know you must handle the `nil` case. It removes the ambiguity introduced by using sentinel values.

OPTIONALS

Introducing optionals

Optionals are Swift's solution to the problem of representing both a value and the absence of a value. An optional type is allowed to reference either a value *or* `nil`.

Think of an optional as a box: it either contains a value, or it doesn't. When it doesn't contain a value, it's said to contain `nil`. The box itself always exists; it's always there for you to open and look inside.

Optional box containing a value

Optional box containing no value

A string or an integer, on the other hand, doesn't have this box around it. Instead there's always a value, such as `"hello"` or 42. Remember, non-optional types are guaranteed to have an actual value.

OPTIONALS

You declare a variable of an optional type by using the following syntax:

```
var errorCode: Int?
```

The only difference between this and a standard declaration is the question mark at the end of the type. In this case, `errorCode` is an "optional `Int`". This means the variable itself is like a box containing either an `Int` or `nil`.

> **Note:** You can add a question mark after any type to create an optional type. This optional type is said to *wrap* the regular non-optional type. For example, optional type `String?` wraps type `String`. In other words: an optional box of type `String?` holds either a `String` or `nil`.
>
> Also, note how an optional type must be made explicit using a type annotation (here `: Int?`). Optional types can never be inferred from initialization values, as those values are of a regular, non-optional type, or `nil`, which can be used with any optional type.

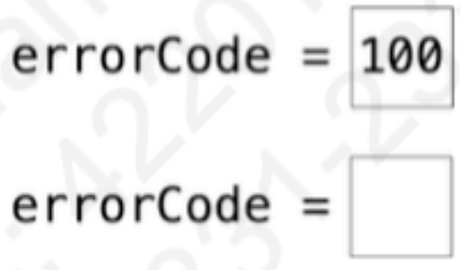
Setting the value is simple. You can either set it to an `Int`, like so:

```
errorCode = 100
```

Or you can set it to `nil`, like so:

```
errorCode = nil
```

This diagram may help you visualize what's happening:



The optional box always exists. When you assign `100` to the variable, you're filling the box with the value. When you assign `nil` to the variable, you're emptying the box.

Take a few minutes to think about this concept. The box analogy will be a big help as you go through the rest of the chapter and begin to use optionals.

OPTIONALS

MINI EXERCISES

1. Make an optional `String` called `myFavoriteSong`. If you have a favorite song, set it to a string representing that song. If you have more than one favorite song or no favorite, set the optional to `nil`.

2. Create a constant called `parsedInt` and set it equal to `Int("10")` which tries to parse the string `10` and convert it to an `Int`. Check the type of `parsedInt` using Option-Click. Why is it an optional?

OPTIONALS

Unwrapping optionals

It's all well and good that optionals exist, but you may be wondering how you can look inside the box and manipulate the value it contains.

Take a look at what happens when you print out the value of an optional:

```
var result: Int? = 30
print(result)
```

This prints the following:

```
Optional(30)
```

That isn't really what you wanted — although if you think about it, it makes sense. Your code has printed the box. The result says, "`result` is an optional that contains the value `30`".

To see how an optional type is different from a non-optional type, see what happens if you try to use `result` as if it were a normal integer:

```
print(result + 1)
```

This code triggers an error:

```
Value of optional type 'Int?' not unwrapped; did you mean to use '!' or
'?'?
```

It doesn't work because you're trying to add an integer to a box — not to the value inside the box, but to the box itself. That doesn't make sense!

OPTIONALS

Force unwrapping

The error message gives an indication of the solution: It tells you that the optional is `not unwrapped`. You need to unwrap the value from its box. It's like Christmas!

Let's see how that works. Consider the following declarations:

```
var authorName: String? = "Matt Galloway"
```

```
var authorAge: Int? = 30
```

There are two different methods you can use to unwrap these optionals. The first is known as **force unwrapping**, and you perform it like so:

```
var unwrappedAuthorName = authorName!
print("Author is \(unwrappedAuthorName)")
```

This code prints:

```
Author is Matt Galloway
```

Great! That's what you'd expect.

The exclamation mark after the variable name tells the compiler that you want to look inside the box and take out the value. The result is a value of the wrapped type. This means `unwrappedAuthorName` is of type `String`, not `String?`.

The use of the word "force" and the exclamation mark `!` probably conveys a sense of danger to you, and it should. You should use force unwrapping sparingly. To see why, consider what happens when the optional doesn't contain a value:

```
authorName = nil
print("Author is \(authorName!)")
```

This code produces the following runtime error:

```
fatal error: unexpectedly found nil while unwrapping an Optional value
```

The error occurs because the variable contains no value when you try to unwrap it. What's worse is that you get this error at runtime rather than compile time – which means you'd only notice the error if you happened to execute this code with some invalid input. Worse yet, if this code were inside an app, the runtime error would cause the app to crash!

How can you play it safe?

To stop the runtime error here, you could wrap the code that unwraps the optional in a check, like so:

```
if authorName != nil {
  print("Author is \(authorName!)")
} else {
  print("No author.")
}
```

The `if` statement checks if the optional contains `nil`. If it doesn't, that means it contains a value you can unwrap.

The code is now safe, but it's still not perfect. If you rely on this technique, you'll have to remember to check for `nil` every time you want to unwrap an optional. That will start to become tedious, and one day you'll forget and once again end up with the possibility of a runtime error.

OPTIONALS

Optional binding

Swift includes a feature known as **optional binding**, which lets you safely access the value inside an optional. You use it like so:

```
if let unwrappedAuthorName = authorName {
  print("Author is \(unwrappedAuthorName)")
} else {
  print("No author.")
}
```

You'll immediately notice that there are no exclamation marks here. This optional binding gets rid of the optional type. If the optional contains a value, this value is unwrapped and stored in, or *bound to*, the constant `unwrappedAuthorName`. The `if` statement then executes the first block of code, within which you can safely use `unwrappedAuthorName`, as it's a regular non-optional `String`.

If the optional doesn't contain a value, then the `if` statement executes the `else` block. In that case, the `unwrappedAuthorName` variable doesn't even exist.

You can see how optional binding is much safer than force unwrapping, and you should use it whenever an optional might be `nil`. Force unwrapping is only appropriate when an optional *is guaranteed* contain a value.

Because naming things is so hard, it's common practice to give the unwrapped constant the same name as the optional (thereby *shadowing* that optional):

```
if let authorName = authorName {
  print("Author is \(authorName)")
} else {
  print("No author.")
}
```

You can even unwrap multiple values at the same time, like so:

```
if let authorName = authorName,
   let authorAge = authorAge {
  print("The author is \(authorName) who is \(authorAge) years old.")
} else {
  print("No author or no age.")
}
```

This code unwraps two values. It will only execute the `if` part of the statement when both optionals contain a value.

You can combine unwrapping multiple optionals with additional boolean checks. For example:

```
if let authorName = authorName,
   let authorAge = authorAge,
   authorAge >= 40 {
  print("The author is \(authorName) who is \(authorAge) years old.")
} else {
  print("No author or no age or age less than 40.")
}
```

Here, you unwrap `name` and `age`, and check that `age` is greater than or equal to 40. The expression in the `if` statement will only be `true` if `name` is non-nil, *and* `age` is non-nil, *and* `age` is greater than or equal to 40.

Now you know how to safely look inside an optional and extract its value, if one exists.

OPTIONALS

Mini-exercises

1. Using your `myFavoriteSong` variable from earlier, use optional binding to check if it contains a value. If it does, print out the value. If it doesn't, print `"I don't have a favorite song."`

2. Change `myFavoriteSong` to the opposite of what it is now. If it's `nil`, set it to a string; if it's a string, set it to `nil`. Observe how your printed result changes.

OPTIONALS

Introducing guard

Sometimes you want to check a condition and only continue executing a function if the condition is true, such as when you use optionals. Imagine a function that fetches some data from the network; that fetch might fail if the network is down. The usual way to encapsulate this behavior is using an optional, which has a value if the fetch succeeds, and `nil` otherwise.

Swift has a useful and powerful feature to help in situations like this: the **guard statement**.

Consider the following function:

```
func calculateNumberOfSides(shape: String) -> Int? {
  switch shape {
  case "Triangle":
    return 3
  case "Square":
    return 4
  case "Rectangle":
    return 4
  case "Pentagon":
    return 5
  case "Hexagon":
    return 6
  default:
    return nil
  }
}
```

This function takes a shape name and returns the number of sides that shape has. If the shape isn't known, or you pass something that isn't a shape, then it returns `nil`.

OPTIONALS

You could use this function like so:

```
func maybePrintSides(shape: String) {
  let sides = calculateNumberOfSides(shape: shape)

  if let sides = sides {
    print("A \(shape) has \(sides) sides.")
  } else {
    print("I don't know the number of sides for \(shape).")
  }
}
```

There's nothing wrong with this, and it would work. However the same logic could be written with a guard statement like so:

```
func maybePrintSides(shape: String) {
  guard let sides = calculateNumberOfSides(shape: shape) else {
    print("I don't know the number of sides for \(shape).")
    return
  }

  print("A \(shape) has \(sides) sides.")
}
```

The guard statement comprises `guard` followed by a condition that can include both Boolean expressions and optional bindings, followed by `else`, followed by a block of code. The block of code covered by the `else` will execute if the condition is *false*.

The block of code that executes in the case of the condition being false *must* return — this is the true beauty of the guard statement.

You may hear programmers talking about the "happy path" through a function; this is the path you'd expect to happen most of the time. Any other path followed would be due to an error, or another reason why the function should return earlier than expected.

Guard statements ensure the happy path remains on the left hand side of the code; this is usually regarded as a good thing as it makes code more readable and understandable.

Also, because the guard statement must return in the false case, the Swift compiler knows that if the condition was true, anything checked in the guard statement's condition *must* be true for the remainder of the function. This means the compiler

can make certain optimizations. You don't need to understand how these optimizations work, or even what they are, since Swift is designed to be user-friendly and fast.

OPTIONALS

Nil coalescing

There's one final and rather handy way to unwrap an optional. You use it when you want to get a value out of the optional *no matter what* — and in the case of `nil`, you'll use a default value. This is called **nil coalescing**.

Here's how it works:

```
var optionalInt: Int? = 10
var mustHaveResult = optionalInt ?? 0
```

The nil coalescing happens on the second line, with the double question mark (`??`), known as the **nil coalescing operator**. This line means `mustHaveResult` will equal either the value inside `optionalInt`, or `0` if `optionalInt` contains `nil`. In this example, `mustHaveResult` contains the concrete `Int` value of `10`.

The code above is equivalent to the following:

```
var optionalInt: Int? = 10
var mustHaveResult: Int
if let unwrapped = optionalInt {
  mustHaveResult = unwrapped
} else {
  mustHaveResult = 0
}
```

Set the `optionalInt` to `nil`, like so:

```
optionalInt = nil
mustHaveResult = optionalInt ?? 0
```

Now `mustHaveResult` equals `0`.

OPTIONALS

Key points

- `nil` represents the absence of a value.
- Non-optional variables and constants must always have a non-`nil` value.
- **Optional** variables and constants are like boxes that can contain a value *or* be empty (`nil`).
- To work with the value inside an optional, you must first unwrap it from the optional.

- The safest ways to unwrap an optional's value is by using **optional binding** or **nil coalescing**. Use **forced unwrapping** only when appropriate, as it could produce a runtime error.

FEEDBACK TIME

http://bit.ly/iOSFeedback

THANK YOU