PRO SWIFT

BOOK AND VIDEOS

Break out of beginner's S
with this hands-on g

FREE SAMPLE

Paul Hudson

Chapter 1
Syntax

Wendy Lu (@wendyluwho), iOS engineer at Pinterest

Use `final` on properties and methods when you know that a declaration does not need to be overridden. This allows the compiler to replace these dynamically dispatched calls with direct calls. You can even mark an entire class as final by attaching the attribute to the class itself.

Pattern matching

`Switch/case` is not a new concept: insert a value, then take one of several courses of action. Swift's focus on safety adds to the mix a requirement that all possible cases be catered for – something you don't get in C without specific warnings enabled – but that's fairly trivial.

What makes Swift's **`switch`** syntax interesting is its flexible, expressive pattern matching. What makes it *doubly* interesting is that since Swift's launch most of this pattern matching has been extended elsewhere, so that same flexible, expressive syntax is available in **`if`** conditions and **`for`** loops.

Admittedly, if you jump in at the deep end you're more likely to sink rather than swim, so I want to work up from basic examples. To refresh your memory, here's a basic **`switch`** statement:

```
let name = "twostraws"

switch name {
case "bilbo":
   print("Hello, Bilbo Baggins!")
case "twostraws":
   print("Hello, Paul Hudson!")
default:
   print("Authentication failed")
}
```

It's easy enough when you're working with a simple string, but things get more complicated when working with two or more values. For example, if we wanted to validate a name as well as a password, we would evaluate them as a tuple:

```
let name = "twostraws"
let password = "fr0st1es"

switch (name, password) {
case ("bilbo", "bagg1n5"):
   print("Hello, Bilbo Baggins!")
```

```
case ("twostraws", "fr0st1es"):
   print("Hello, Paul Hudson!")
default:
   print("Who are you?")
}
```

You can combine the two values into a single tuple if you prefer, like this:

```
let authentication = (name: "twostraws", password: "fr0st1es")

switch authentication {
case ("bilbo", "bagg1n5"):
   print("Hello, Bilbo Baggins!")
case ("twostraws", "fr0st1es"):
   print("Hello, Paul Hudson!")
default:
   print("Who are you?")
}
```

In this instance, both parts of the tuple must match the **`case`** in order for it to be executed.

Partial matches

When working with tuples, there are some occasions when you want a partial match: you care what some values are but don't care about others. In this situation, use underscores to represent "any value is fine", like this:

```
let authentication = (name: "twostraws", password: "fr0st1es",
ipAddress: "127.0.0.1")

switch authentication {
case ("bilbo", "bagg1n5", _):
   print("Hello, Bilbo Baggins!")
case ("twostraws", "fr0st1es", _):
   print("Hello, Paul Hudson!")
```

```
default:
    print("Who are you?")
}
```

Remember: Swift will take the first matching case it finds, so you need to ensure you look for the most specific things first. For example, the code below would print "You could be anybody!" because the first **case** matches immediately, even though later cases are "better" matches because they match more things:

```
switch authentication {
case (_, _, _):
    print("You could be anybody!")
case ("bilbo", "bagg1n5", _):
    print("Hello, Bilbo Baggins!")
case ("twostraws", "fr0st1es", _):
    print("Hello, Paul Hudson!")
default:
    print("Who are you?")
}
```

Finally, if you want to match only part of a tuple, but still want to know what the other part was, you should use **let** syntax.

```
switch authentication {
case ("bilbo", "bagg1n5", _):
    print("Hello, Bilbo Baggins!")
case ("twostraws", let password, _):
    print("Hello, Paul Hudson: your password was \(password)!")
default:
    print("Who are you?")
}
```

Matching calculated tuples

So far we've covered the basic range of pattern-matching syntax that most developers use. From here on I want to give examples of other useful pattern-matching techniques that are less well known.

Tuples are most frequently created using static values, like this:

```
let name = ("Paul", "Hudson")
```

But tuples are like any other data structure in that they can be created using dynamic code. This is particularly useful when you want to narrow the range of values in a tuple down to a smaller subset so that you need only a handful of **`case`** statements.

To give you a practical example, consider the "fizzbuzz" test: write a function that accepts any number, and returns "Fizz" if the number is evenly divisible by 3, "Buzz" if it's evenly divisible by 5, "FizzBuzz" if its evenly divisible by 3 *and* 5, or the original input number in other cases.

We can calculate a tuple to solve this problem, then pass that tuple into a **`switch`** block to create the correct output. Here's the code:

```
func fizzbuzz(number: Int) -> String {
    switch (number % 3 == 0, number % 5 == 0) {
    case (true, false):
        return "Fizz"
    case (false, true):
        return "Buzz"
    case (true, true):
        return "FizzBuzz"
    case (false, false):
        return String(number)
    }
}

print(fizzbuzz(number: 15))
```

This approach breaks down a large input space – any number – into simple combinations of true and false, and we then use tuple pattern matching in the case statements to select the correct output.

Loops

As you've seen, pattern matching using part of a tuple is easy enough: you either tell Swift what should be matched, use **let** to bind a value to a local constant, or use **_** to signal that you don't care what a value is.

We can use this same approach when working with loops, which allows us to loop over items only if they match the criteria we specify. Let's start with a basic example again:

```
let twostraws = (name: "twostraws", password: "fr0st1es")
let bilbo = (name: "bilbo", password: "bagg1n5")
let taylor = (name: "taylor", password: "fr0st1es")

let users = [twostraws, bilbo, taylor]

for user in users {
    print(user.name)
}
```

That creates an array of tuples, then loops over each one and prints its **name** value.

Just like the **switch** blocks we looked at earlier, we can use **case** with a tuple to match specific values inside the tuples. Add this code below the previous loop:

```
for case ("twostraws", "fr0st1es") in users {
    print("User twostraws has the password fr0st1es")
}
```

We also have identical syntax for binding local constants to the values of each tuple, like this:

```
for case (let name, let password) in users {
    print("User \(name) has the password \(password)")
```

```
}
```

Usually, though, it's preferable to re-arrange the **let** to this:

```
for case let (name, password) in users {
    print("User \(name) has the password \(password)")
}
```

The magic comes when you combine these two approaches, and again is syntactically identical to a **switch** example we already saw:

```
for case let (name, "fr0st1es") in users {
    print("User \(name) has the password \"fr0st1es\"")
}
```

That filters the **users** array so that only items with the password "fr0st1es" will be used in the loop, then creates a **name** constant inside the loop for you to work with.

Don't worry if you're staring at **for case let** and seeing three completely different keywords mashed together: it's not obvious what it does until someone explains it to you, and it will take a little time to sink in. But we're only getting started…

Matching optionals

Swift has two ways of matching optionals, and you're likely to meet both. First up is using **.some** and **.none** to match "has a value" and "has no value", and in the code below this is used to check for values and bind them to local constants:

```
let name: String? = "twostraws"
let password: String? = "fr0st1es"

switch (name, password) {
case let (.some(name), .some(password)):
    print("Hello, \(name)")
case let (.some(name), .none):
```

```
    print("Please enter a password.")
default:
    print("Who are you?")
}
```

That code is made more confusing because **`name`** and **`password`** are used for the input constants as well as the locally bound constants. They are different things, though, which is why **`print("Hello, \(name)")`** won't print **`Hello, Optional("twostraws")`** – the **`name`** being used is the locally bound unwrapped optional.

If it's easier to read, here's the same code with different names used for the matched constants:

```
switch (name, password) {
case let (.some(matchedName), .some(matchedPassword)):
    print("Hello, \(matchedName)")
case let (.some(matchedName), .none):
    print("Please enter a password.")
default:
    print("Who are you?")
}
```

The second way Swift matches optionals is using much simpler syntax, although if you have a fear of optionals this might only make it worse:

```
switch (name, password) {
case let (name?, password?):
    print("Hello, \(name)")
case let (username?, nil):
    print("Please enter a password.")
default:
    print("Who are you?")
}
```

This time the question marks work in a similar way as optional chaining: continue only if a value was found.

Both of these methods work equally well in **`for case let`** code. The code below uses them both to filter out **`nil`** values in a loop:

```
import Foundation
let data: [Any?] = ["Bill", nil, 69, "Ted"]

for case let .some(datum) in data {
    print(datum)
}

for case let datum? in data {
    print(datum)
}
```

Matching ranges

You're probably already using pattern matching with ranges, usually with code something like this:

```
let age = 36

switch age {
case 0 ..< 18:
    print("You have the energy and time, but not the money")
case 18 ..< 70:
    print("You have the energy and money, but not the time")
default:
    print("You have the time and money, but not the energy")
}
```

A very similar syntax is also available for regular conditional statements – we could rewrite that code like this:

```
if case 0 ..< 18 = age {
```

```
    print("You have the energy and time, but not the money")
} else if case 18 ..< 70 = age {
    print("You have the energy and money, but not the time")
} else {
    print("You have the time and money, but not the energy")
}
```

That produces identical results to the **`switch`** block while using similar syntax, but I'm not a big fan of this approach. The reason for my dislike is simple readability: I don't think "if case 0 up to 18 equals age" makes sense if you don't already know what it means. A much nicer approach is to use the pattern match operator, **`~=`**, which would look like this:

```
if 0 ..< 18 ~= age {
    print("You have the energy and time, but not the money")
} else if 18 ..< 70 ~= age {
    print("You have the energy and money, but not the time")
} else {
    print("You have the time and money, but not the energy")
}
```

Now the condition reads "if the range 0 up to 18 matches age", which I think makes a lot more sense.

An even cleaner solution becomes clear when you remember that **`0 ..< 18`** creates an instance of a **`Range`** struct, which has its own set of methods. Right now, its **`contains()`** method is particularly useful: it's longer to type than **`~=`** but it's significantly easier to understand:

```
if (0 ..< 18).contains(age) {
    print("You have the energy and time, but not the money")
} else if (18 ..< 70).contains(age) {
    print("You have the energy and money, but not the time")
} else {
    print("You have the time and money, but not the energy")
}
```

You can combine this range matching into our existing tuple matching code, like this:

```
let user = (name: "twostraws", password: "fr0st1es", age: 36)

switch user {
case let (name, _, 0 ..< 18):
   print("\(name) has the energy and time, but no money")
case let (name, _, 18 ..< 70):
   print("\(name) has the money and energy, but no time")
case let (name, _, _):
   print("\(name) has the time and money, but no energy")
}
```

That last case binds the user's name to a local constant called **`name`** irrespective of the two other values in the tuple. This is a catch all, but because Swift looks for the *first* matching case this won't conflict with the other two in the **`switch`** block.

Matching enums and associated values

In my experience, quite a few people don't really understand enums and associated values, and so they struggle to make use of them with pattern matching. There's a whole chapter on enums later in the book, so if you're not already comfortable with enums and associated values you might want to pause here and read that chapter first.

Basic enum matching looks like this:

```
enum WeatherType {
   case cloudy
   case sunny
   case windy
}

let today = WeatherType.cloudy
```

```
switch today {
case .cloudy:
   print("It's cloudy")
case .windy:
   print("It's windy")
default:
   print("It's sunny")
}
```

You'll also have used enums in basic conditional statements, like this:

```
if today == .cloudy {
   print("It's cloudy")
}
```

As soon as you add associated values, things get more complicated because you can use them, filter on them, or ignore them depending on your goal.

First up, the easiest option: creating an associated value but ignoring it:

```
enum WeatherType {
   case cloudy(coverage: Int)
   case sunny
   case windy
}

let today = WeatherType.cloudy(coverage: 100)

switch today {
case .cloudy:
   print("It's cloudy")
case .windy:
   print("It's windy")
default:
   print("It's sunny")
```

```
}
```

Using this approach, the actual **switch** code is unchanged.

Second: creating an associated value and using it. This uses the same local constant bind we've seen several times now:

```
enum WeatherType {
    case cloudy(coverage: Int)
    case sunny
    case windy
}

let today = WeatherType.cloudy(coverage: 100)

switch today {
case .cloudy(let coverage):
    print("It's cloudy with \(coverage)% coverage")
case .windy:
    print("It's windy")
default:
    print("It's sunny")
}
```

Lastly: creating an associated type, binding a local constant to it, but also using that binding to filter for specific values. This uses the **where** keyword to create a requirements clause that clarifies what you're looking for. In our case, the code below prints two different messages depending on the associated value that is used with **cloudy**:

```
enum WeatherType {
    case cloudy(coverage: Int)
    case sunny
    case windy
}
```

```
let today = WeatherType.cloudy(coverage: 100)

switch today {
case .cloudy(let coverage) where coverage < 100:
   print("It's cloudy with \(coverage)% coverage")
case .cloudy(let coverage) where coverage == 100:
   print("You must live in the UK")
case .windy:
   print("It's windy")
default:
   print("It's sunny")
}
```

Now, as promised I'm building up from basic examples, but if you're ready I want to show you how to combine two of these techniques together: associated values and range matching. The code below now prints four different messages: one when coverage is 0, one when it's 100, and two more using ranges from 1 to 50 and 51 to 99:

```
enum WeatherType {
   case cloudy(coverage: Int)
   case sunny
   case windy
}

let today = WeatherType.cloudy(coverage: 100)

switch today {
case .cloudy(let coverage) where coverage == 0:
   print("You must live in Death Valley")
case .cloudy(let coverage) where (1...50).contains(coverage):
   print("It's a bit cloudy, with \(coverage)% coverage")
case .cloudy(let coverage) where (51...99).contains(coverage):
   print("It's very cloudy, with \(coverage)% coverage")
case .cloudy(let coverage) where coverage == 100:
```

```
    print("You must live in the UK")
case .windy:
    print("It's windy")
default:
    print("It's sunny")
}
```

If you want to match associated values in a loop, adding a **`where`** clause is the wrong approach. In fact, this kind of code won't even compile:

```
let forecast: [WeatherType] = [.cloudy(coverage:
40), .sunny, .windy, .cloudy(coverage: 100), .sunny]

for day in forecast where day == .cloudy {
    print(day)
}
```

That code would be fine without associated values, but because the associated value has meaning the **`where`** clause isn't up to the job – it has no way to say "and bind the associated value to a local constant." Instead, you're back to **`case let`** syntax, like this:

```
let forecast: [WeatherType] = [.cloudy(coverage:
40), .sunny, .windy, .cloudy(coverage: 100), .sunny]

for case let .cloudy(coverage) in forecast {
    print("It's cloudy with \(coverage)% coverage")
}
```

If you know the associated value and want to use it as a filter, the syntax is almost the same:

```
let forecast: [WeatherType] = [.cloudy(coverage:
40), .sunny, .windy, .cloudy(coverage: 100), .sunny]

for case .cloudy(40) in forecast {
    print("It's cloudy with 40% coverage")
```

```
}
```

Matching types

You should already know the **`is`** keyword for matching, but you might not know that it can be used as pattern matching in loops and **`switch`** blocks. I think the syntax is quite pleasing, so I want to demonstrate it just briefly:

```
let view: AnyObject = UIButton()

switch view {
case is UIButton:
   print("Found a button")
case is UILabel:
   print("Found a label")
case is UISwitch:
   print("Found a switch")
case is UIView:
   print("Found a view")
default:
   print("Found something else")
}
```

I've used UIKit as an example because you should already know that **`UIButton`** inherits from **`UIView`**, and I need to give you a big warning…

Remember: Swift will take the first matching case it finds, and **`is`** returns true if an object is a specific type or one of its parent classes. So, the above code will print "Found a button", whereas the below code will print "Found a view":

```
let view: AnyObject = UIButton()

switch view {
case is UIView:
   print("Found a view")
```

```
case is UIButton:
   print("Found a button")
case is UILabel:
   print("Found a label")
case is UISwitch:
   print("Found a switch")
default:
   print("Found something else")
}
```

To give you a more useful example, you can use this approach to loop over all subviews in an array and filter for labels:

```
for label in view.subviews where label is UILabel {
   print("Found a label with frame \(label.frame)")
}
```

Even though **where** ensures only **UILabel** objects are processed in the loop, it doesn't actually do any typecasting. This means if you wanted to access a label-specific property of **label**, such as its **text** property, you need to typecast it yourself. In this situation, using **for case let** instead is easier, as this filters and typecasts in one:

```
for case let label as UILabel in view.subviews {
   print("Found a label with text \(label.text)")
}
```

Using the where keyword

To wrap up pattern matching, I want to demonstrate a couple of interesting ways to use **where** clauses so that you can get an idea of what it's capable of.

First, an easy one: loop over an array of numbers and print only the odd ones. This is trivial using **where** and modulus, but it demonstrates that your **where** clause can contain calculations:

```
for number in numbers where number % 2 == 1 {
    print(number)
}
```

You can also call methods, like this:

```
let celebrities = ["Michael Jackson", "Taylor Swift", "Michael Caine", "Adele Adkins", "Michael Jordan"]

for name in celebrities where !name.hasPrefix("Michael") {
    print(name)
}
```

That will print "Taylor Swift" and "Adele Adkins". If you want to make your **where** clause more complicated, just add operators such as **&&**:

```
let celebrities = ["Michael Jackson", "Taylor Swift", "Michael Caine", "Adele Adkins", "Michael Jordan"]

for name in celebrities where name.hasPrefix("Michael") && name.characters.count == 13 {
    print(name)
}
```

That will print "Michael Caine".

While it's possible to use **where** to strip out optionals, I wouldn't recommend it. Consider the example below:

```
let celebrities: [String?] = ["Michael Jackson", nil, "Michael Caine", nil, "Michael Jordan"]

for name in celebrities where name != nil {
    print(name)
}
```

That certainly works, but it does nothing about the optionality of the strings in the loop so it prints out this:

```
Optional("Michael Jackson")
Optional("Michael Caine")
Optional("Michael Jordan")
```

Instead, use **`for case let`** to handle optionality, and use **`where`** to focus on filtering values. Here's the preferred way of writing that loop:

```
for case let name? in celebrities {
    print(name)
}
```

When that runs, **`name`** will only contain the strings that had values, so its output will be:

```
Michael Jackson
Michael Caine
Michael Jordan
```

Nil coalescing

Swift optionals are one of the fundamental ways it guarantees program safety: a variable can only be used if it definitely has a value. The problem is that optionals make your code a bit harder to read and write, because you need to unwrap them safely.

One alternative is to explicitly unwrap optionals using `!`. This is also known as the "crash operator" because if you use `!` with an optional that is nil, your program will die immediately and your users will be baying for blood.

A smarter alternative is the nil coalescing operator, `??`, which allows you to access an optional and provide a default value if the optional is nil.

Consider this optional:

```
let name: String? = "Taylor"
```

That's a constant called `name` that contains either a string or nil. If you try to print that using `print(name)` you'll see `Optional("Taylor")` rather than just "Taylor", which isn't really what you want.

Using nil coalescing allows us to use an optional's value or provide a default value if it's nil. So, you could write this:

```
let name: String? = "Taylor"
let unwrappedName = name ?? "Anonymous"
print(unwrappedName)
```

That will print "Taylor": `name` was a `String?`, but `unwrappedName` is guaranteed to be a regular `String` – not optional – because of the nil coalescing operator. To see the default value in action, try this instead:

```
let name: String? = nil
let unwrappedName = name ?? "Anonymous"
print(unwrappedName)
```

That will now print "Anonymous", because the default value is used instead.

Of course, you don't need a separate constant when using nil coalescing – you can write it inline, like this:

```
let name: String? = "Taylor"
print(name ?? "Anonymous")
```

As you can imagine, nil coalescing is great for ensuring sensible values are in place before you use them, but it's particularly useful in removing some optionality from your code. For example:

```
func returnsOptionalName() -> String? {
    return nil
}

let returnedName = returnsOptionalName() ?? "Anonymous"
print(returnedName)
```

Using this approach, **`returnedName`** is a **`String`** rather than a **`String?`** because it's guaranteed to have a value.

So far, so straightforward. However, nil coalescing gets more interesting when you combine it with the **`try?`** keyword.

Consider a simple app that lets a user type and save text. When the app runs, it wants to load whatever the user typed previously, so it probably uses code like this:

```
do {
    let savedText = try String(contentsOfFile: "saved.txt")
    print(savedText)
} catch {
    print("Failed to load saved text.")
}
```

If the file exists, it will be loaded into the **`savedText`** constant. If not, the **`contentsOfFile`** initializer will throw an exception, and “Failed to load saved text” will be

printed. In practice, you'd want to extend this so that **savedText** always has a value, so you end up with something like this:

```
let savedText: String

do {
    savedText = try String(contentsOfFile: "saved.txt")
} catch {
    print("Failed to load saved text.")
    savedText = "Hello, world!"
}

print(savedText)
```

That's a lot of code and it doesn't really accomplish very much. Fortunately, there's a better way: nil coalescing. Remember, **try** has three variants: **try** attempts some code and might throw an exception, **try!** attempts some code and crashes your app if it fails, and **try?** attempts some code and returns nil if the call failed.

That last one is where nil coalescing steps up to the plate, because this exactly matches our previous examples: we want to work with an optional value, and provide a sensible default if the optional is nil. In fact, using nil coalescing we can rewrite all that into just two lines of code:

```
let savedText = (try? String(contentsOfFile: "saved.txt")) ??
"Hello, world!"
print(savedText)
```

That means "try loading the file, but if loading fails use this default text instead" – a neater solution and much more readable.

Combining **try?** with nil coalescing is perfect for situations when a failed **try** isn't an error, and I think you'll find this pattern useful in your own code.

Guard

The **guard** keyword has been with us since Swift 2.0, but because it does four things in one you'd be forgiven for not using it fully.

The first use is the most obvious: **guard** is used for early returns, which means you exit a function if some preconditions are not satisfied. For example, we could write a rather biased function to give an award to a named person:

```
func giveAward(to name: String) {
    guard name == "Taylor Swift" else {
        print("No way!")
        return
    }

    print("Congratulations, \(name)!")
}

giveAward(to: "Taylor Swift")
```

Using that **guard** statement in the **giveAward(to:)** method ensures that only Taylor Swift wins awards. It's biased like I said, but the precondition is clear and this code will only run when requirements I have put in place are satisfied.

This initial example looks almost identical to using **if**, but **guard** has one massive advantage: it makes your intention clear, not only to people but also to the compiler. This is an early return, meaning that you want to exit the method if your preconditions aren't satisfied. Using **guard** makes that clear: this condition isn't part of the functionality of the method, it's just there to ensure the actual code is safe to run. It's also clear to the compiler, meaning that if you remove the **return** statement your code will no longer build – Swift knows this is an early return, so it will not let you forget to exit.

The second use of guard is a happy side effect of the first: using **guard** and early returns allows you to reduce your indent level. Some developers believe very strongly that early returns must never be used, and instead each function should return from only one place. This forces extra indents in the main body of your function code, something like this: