Pattern Matching, Part 2: tuples, ranges & types

Mar 30, 2016
Updated: Oct 17, 2016
6 minute read — Swift 3.0
Also available in: Chinese

In the previous article, we saw the basics of pattern matching using `switch` on `enums`. But what about using `switch` with anything other than `enum` types?

> *This post is part of an article series. You can read all the parts here: part 1, part 2, part 3, part 4*

Pattern matching with tuples

In Swift, you're not restricted to use `switch` on integer values or enums like in ObjC.

You can actually `switch` on a lot of stuff, including (but not restricting to) tuples.

This means that you can check multiple data at once by grouping them in a tuple. For example, imagine you have a `CGPoint` and want to check if it's on a special axis, you can `switch` on its `.x` and `.y` properties!

```
let point = CGPoint(x: 7, y: 0)
switch (point.x, point.y) {
  case (0,0): print("On the origin!")
  case (0,_): print("x=0: on Y-axis!")
  case (_,0): print("y=0: on X-axis!")
  case (let x, let y) where x == y: print("On y=x")
  default: print("Quite a random point here.")
}
```

Notice the use of the wildcard _ we saw in the previous article, as well as the `(let x, let y)` in the fourth `case` which *binds* the variables to then be able to use a `where` to check if both are equal, also as we saw in the previous article.

Cases are evaluated in order

Also notice that the `switch` statement will evaluate its `case` patterns in the order they are defined, and will stop at the first one which matches. Contrary to C and Objective-C, there is no need for a `break` keyword[1].

This means that in the code above, if the point is `(0,0)`, then it will match the first `case` and print `"On the origin!"`, but it will stop there, without matching with `(0,_)` nor with `(_,0)`, even if those `case` patterns would otherwise match. Because it stops at the first match.

Strings and Characters

But why stop at tuples? In Swift you can also `switch` to a lot of native types, including `String` and `Character` for example:

```
let car: Character = "J"
switch car {
  case "A", "E", "I", "O", "U", "Y": print("Vowel")
  default: print("Consonant")
}
```

Here you can also notice that you can use multiple patterns separated by a comma to execute the same code for mutliple matches (namely all the vowels here). This allows you to avoid code repetition easily.

Ranges

Ranges are also usable for pattern matching. As a reminder, a `Range<T>` is a generic type which contains a `start` and `end` both of type `T`, where `T` must be a `ForwardIndexType`. This includes types like `Int` and `Character`.

In Swift 2, you can declare a range either explicitly using `Range(start: 1900, end: 2000)`, or using the syntactic sugar operators `..<` (range excluding its `end`) and `...` (range including its `end`); e.g. you can declare the same range as above using `1900..<2000`. The latter form (using the `..<` operator) is prefered though, both because it's more readable and because `Range(start:end:)` will be removed in Swift 3.0 anyway.

So how do we use that with `switch`? Well easy, use ranges in your `case` patterns to check if a value is within that range!

```
let count = 7
switch count {
  case Int.min..<0: print("Negative count, really?")
  case 0: print("Nothing")
  case 1: print("One")
  case 2..<5: print("A few")
  case 5..<10: print("Some")
  default: print("Many")
}
```

Here you see that we mixed `cases` with a single `Int` value and `cases` with `Range<Int>` values. That's not a problem as long as all possible cases are covered by your `switch`.

Even if using `Int` for ranges is the most common case, we can also do that with other `ForwardIndexType`, including… `Character`! Remember the code above? The problem is that it printed "Consonant" even for punctuation characters and anything other than `A-Z`. So let's solve that[2] (and also include lowercase vowels and consonants):

```
func charType(_ car: Character) -> String {
  switch car {
    case "A", "E", "I", "O", "U", "Y", "a", "e", "i", "o", "u", "y":
      return "Vowel"
    case "A"..."Z", "a"..."z":
      return "Consonant"
    default:
      return "Other"
  }
}
print("Jules Verne".characters.map(charType))
// ["Consonant", "Vowel", "Consonant", "Vowel", "Consonant", "Other", "Con
```

Types

Ok but can we go further? Well of course we can: let's also use pattern matching… on types!

For this example, let's define 3 structs all conforming to the same protocol:

```
protocol Medium {
  var title: String { get }
}
struct Book: Medium {
  let title: String
  let author: String
  let year: Int
}
struct Movie: Medium {
  let title: String
  let director: String
  let year: Int
}
struct WebSite: Medium {
  let url: NSURL
```

```
    let title: String
  }

  // And an array of Media to switch onto
  let media: [Medium] = [
    Book(title: "20,000 leagues under the sea", author: "Jules Verne", year:
    Movie(title: "20,000 leagues under the sea", director: "Richard Fleische
  ]
```

Then how do we `switch` according to the type of `Medium` and do a different thing for `Book` than for `Movie`? Easy, use `as` and `is` for pattern matching!

```
  for medium in media {
    // The title part of the protocol, so no need for a switch there
    print(medium.title)
    // But for the other properties, it depends on the type
    switch medium {
    case let b as Book:
      print("Book published in \(b.year)")
    case let m as Movie:
      print("Movie released in \(m.year)")
    case is WebSite:
      print("A WebSite with no date")
    default:
      print("No year info for \(medium)")
    }
  }
```

Notice we use `as` for `Book` and `Movie` here, because we want both to see if they match the type, and if they do, store the casted type in a constant (`let b` or `let m`), because we then want to use it[3].

On the other hand, we only used `is` for `WebSite` because we only want to check if `medium` can match the pattern of "being a `WebSite`". But if it does, we don't need to type-cast it and use the type-casted value (we don't use it in the `print` statement). That's a bit like if we did write `case let _ as WebSite`, as we don't care about the `WebSite` object as long as it's of that type.

Note: having to use `as` and `is` like this in a `switch` might sometimes reveal a code-smell, e.g. in that particular use case it'd probably have been better to add a `var releaseInfo: String { get }` property to the `protocol Medium` instead of `switch`-ing over the various types.

What's next?

In the upcoming parts we'll look at how to make your own types be directly usable for pattern matching, explore some more syntactic sugar, then look at some pattern matching usages outside of the `switch` statement and some even more complex pattern expressions… can't wait!

Note: I'm gonna be traveling ✈ to Japan 🇯🇵 in the next two weeks 😓, so might not be able to publish the next parts of this article series as quickly as the previous ones, but I won't forget you!

> ⏩ *Read next part of this article series here: part 3*

> *Thanks to Frank Manno for updating the code samples of this article to Swift 3!*

---

1. You can override this behavior using the `fallthrough` keyword, to let the evaluation continue through the next `case`. But in practice this is very rarely useful and not often encountered. ↩

2. Of course, that's not the best and recommended way to implement such a string analysis feature — as Unicode characters and locales are way more complex than that. So instead for such a feature we should probably use `NSCharacterSet`, consider what are the letters that the current `NSLocale` consider as vowels (is "y" a vowel? What about "õ" or "ø"?), etc. So don't take that example too seriously as it's only a sample to illustrate the power of `switch + Range`. ↩

3. despite the similarity with expressions like `if let b = medium as? Book` — where you also bind a variable if `medium` can be casted to the expected type — be careful to use `as` and not `as?` in a pattern matching expression. Even if the mechanics might seem similar, the semantics are different here ("try type-casting and `nil` if it fails" vs. "check if the pattern of considering it of this type does match or not"). ↩

Share this post: f 🐦 g+ t