Pattern Matching, Part 1: switch, enums & where clauses

Mar 27, 2016
Updated: Oct 17, 2016
7 minute read — Swift 3.0
Also available in: cn Chinese

From a simple `switch` to complex expressions, pattern matching in Swift can be quite powerful. Today we're gonna start exploring it by seing some cool usages of `switch`, before going further in later articles with even more advanced pattern matching techniques.

This article serves as an introduction to the incomming articles about Pattern Matching.

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

Switch Basics

The simplest and more common usage of pattern matching in Swift is via a `switch` statement. You hopfully already know their simplest form, e.g.:

```
enum Direction {
  case north, south, east, west
}

// We can easily switch between simple enum values
extension Direction: CustomStringConvertible {
  var description: String {
    switch self {
    case .north: return "⬆"
    case .south: return "⬇"
    case .east: return "➡"
    case .west: return "⬅"
    }
  }
}
```

But `switch` can go even further, allowing you to match against **patterns** containing variables, and binding to those variables when it matches. This applies typically to `enum` with associated values:

```
enum Media {
  case book(title: String, author: String, year: Int)
  case movie(title: String, director: String, year: Int)
  case website(url: NSURL, title: String)
}

extension Media {
  var mediaTitle: String {
    switch self {
    case .book(title: let aTitle, author: _, year: _):
      return aTitle
    case .movie(title: let aTitle, director: _, year: _):
      return aTitle
    case .website(url: _, title: let aTitle):
      return aTitle
    }
  }
}

let book = Media.book(title: "20,000 leagues under the sea", author: "Jule
book.mediaTitle
```

The generic syntax in this case is `case MyEnum.enumValue(let variable)` to tell "if the value is a `MyEnum.enumValue` — which has an associated value — then bind the variable `variable` to that associated value".

When the `media` instance match one of the `case` — like with `book` which is a `Media.book` and matches the first `case` of the `switch` — then **a new variable `let aTitle` is created** and the associated value `title` is *bound* to this given variable. That's why the `let` is needed there, because that's gonna create a new variable (well, constant) if it matches.

Note that you can write the `let` in front of the whole expression, instead of using it in front of each variable, e.g. these two lines are equivalent:

```
case .book(title: let aTitle, author: let anAuthor, year: let aYear): …
case let .book(title: aTitle, author: anAuthor, year: aYear): …
```

Notice also the use of the wildcard pattern _ in the above code, which basically means "I expect to be something there, but I don't care about it, so don't bother binding it to a variable as I won't use it anyway". So that's somehow like a placeholder for a value we won't use.

Using fixed values

Remember that `case` is still about **pattern matching**, so it tries to **match** what you try to compare it with. This means that you can also use constant values to check if it matches. For example:

```
extension Media {
  var isFromJulesVerne: Bool {
    switch self {
    case .book(title: _, author: "Jules Verne", year: _): return true
    case .movie(title: _, director: "Jules Verne", year: _): return true
    default: return false
    }
  }
}
book.isFromJulesVerne
```

Granted, this example is not really useful *per se*, but it's just to show you that you can bind to constant values too. I mentionned it because I've already seen code that binds a value to a variable then check if that variable is equal to a constant... instead of pattern-match directly with the constant!

A more useful and generic example could be something like:

```
extension Media {
  func checkAuthor(_ author: String) -> Bool {
    switch self {
    case .book(title: _, author: author, year: _): return true
    case .movie(title: _, director: author, year: _): return true
    default: return false
    }
  }
}
book.checkAuthor("Jules Verne")
```

Notice here that althrough we use `author` in the `case` patterns, we don't need to use `let` here (unlike in the preview paragraph). That's because in this case, we won't create a variable to bind a value to it. Instead, we use a constant which already has a value (provided by the function parameter) — not creating a new one that is gonna be bound to the value matched with `self`.

> *[EDIT] Another great example of matching with constants have been suggested by @ashfurrow on Twitter when dealing with HTTP status codes:*

```
enum Response {
  case httpResponse(statusCode: Int)
  case networkError(Error)
```

```
  …
}

let response: Response = …
switch response {
  case .httpResponse(200): …
  case .httpResponse(404): …
  …
}

// cleaner than using stuff like `case .httpResponse(let code) where code
```

Binding multiple patterns at once

As per Swift 2.2, we can't bind multple patterns at once. So for example this isn't possible yet, because here we try to declare variables both if `self` matches `.book` or `.movie` , and bind a variable in both cases:

```
extension Media {
  var mediaTitle2: String {
    switch self {
      // Error: 'case' labels with multiple patterns cannot declare variab
    case let .book(title: aTitle, author: _, year: _), let .movie(title:
      return aTitle
    case let .website(url: _, title: aTitle):
      return aTitle
    }
  }
}
```

This is understandable in most cases; like what would you expect the code to do if you tried to write `case let .book(title: aTitle, author: _, year: _), let .movie(title: _, director: _, year: aYear)`? How would you be able to use the bound variables `aTitle` or `aYear` in your `case` code then? If it's a `.book` then only `aTitle` would have been bound, so what about `aYear`? What if you tried to use that `aYear` variable in the code of that `case`? Wouldn't probably make sense.

But one might think that in the specific case when you try to bind variables of the **same type** and with the **same name**, that would still make sense to work, like in the example above where we try to bind with `aTitle` in both cases (`.book` and `.movie`). And that would be quite useful to avoid repeating code, right? So why is this not possible in that specific case? Well fear not, this Swift-Evolution Proposal SE-0043 has been accepted and allow this in Swift 3.

Using tuples without argument labels

Note that when dealing with `enum` with associated values, we can consider the associated values to be one single associated value represented as a tuple containing all the real associated values. This has two consequences:

- First you can omit the argument labels, and the `case` will still work:

```
extension Media {
  var mediaTitle2: String {
    switch self {
    case let .book(title, _, _): return title
    case let .movie(title, _, _): return title
    case let .website(_, title): return title
    }
  }
}
```

- Second, you can also treat the associated value like a unique, big tuple, then access its individual elements:

```
extension Media {
  var mediaTitle3: String {
    switch self {
    case let .book(tuple): return tuple.title
    case let .movie(tuple): return tuple.title
    case let .website(tuple): return tuple.title
    }
  }
}
```

As an added bonus, not specifying the tuple at all is syntactic sugar for matching any associated values, so those 3 expressions are equivalent:

```
case .website // not specifying the tuple at all
case .website(_) // matching a single tuple of associated values that we d
case .website(_, _) // matching individual associated values that we don't
```

Using Where

Pattern matching allows way more powerful stuff than just comparing two enums. You can add conditions to the comparisons, using a `where` clause, like this:

```
extension Media {
  var publishedAfter1930: Bool {
    switch self {
```

```
        case let .book(_, _, year) where year > 1930: return true
        case let .movie(_, _, year) where year > 1930: return true
        case .website: return true // same as "case .website(_)" but we ignore
        default: return false
        }
    }
}
```

This will only match if both the left side of the pattern (like `let .book(_, _, year)`) successfully matches, **and** the `where` condition is evaluated to `true`. This allows some powerful patterns that we'll dig into in later parts of this article series.

What's next?

This article was pretty simple to remind you of the basics of pattern matching in `switch`. The next parts are gonna talk about more advanced usages, including:

- using `switch` with anything other than `enum` (especially pattern matching with `tuples`, `structs`, `is` and `as`).
- using pattern matching with other statements, including `if case`, `guard case`, `for case`, `=~`, ...
- Nested patterns, including ones containing `Optional` values
- Combining them all to create some magic.

> ⏩ *Read next part of this article series here: part 2*

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

Share this post: