Pattern Matching, Part 3: Custom pattern matching & syntactic sugar

Apr 24, 2016
Updated: Oct 17, 2016
6 minute read — Swift 3.0
Also available in: Chinese

In parts 1 and 2 of this article series, we saw some usages of `switch` on a lot of things, including `tuples`, `Range`, `String`, `Character` and even type. But what if we can use pattern matching even with our own custom types?

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

Switch and the pattern matching operator

When you write `case 1900..<2000` for example in a `switch`, how does Swift know how to compare that `Range` with the single value you're `switch`-ing onto?

Well the answer is simple: Swift uses the `~=` operator. You can use a `Range<I>` in a `case` when switching over a `I` (e.g. an `Int`) simply because the `~=` operator is declared between a `Range<I>` and a `I`:

```
func ~=<I : ForwardIndexType where I : Comparable>(pattern: Range<I>, valu
```

And in fact when you write `switch someI` with a `case aRangeOfI` then Swift will try to match it by calling `aRangeOfI ~= someI` (which returns a `Bool` telling if it matched).

This means that you can actually define the same operator `~=` on your own custom types to make them usable in a `switch`/`case` statement, the same way you can use `Range`!

Making your types respond to pattern matching

So let's do that with a custom struct:

```
struct Affine {
  var a: Int
  var b: Int
}

func ~= (lhs: Affine, rhs: Int) -> Bool {
  return rhs % lhs.a == lhs.b
}

switch 5 {
case Affine(a: 2, b: 0): print("Even number")
case Affine(a: 3, b: 1): print("3x+1")
case Affine(a: 3, b: 2): print("3x+2")
default: print("Other")
}
```

And this works and prints `3x+2`!

Note however that Swift can't know if the `switch` is exhaustive with custom pattern matching. For example, even if we add a `case Affine(a: 2, b: 1)` and a `case Affine(a: 2, b: -1)` which would cover every positive and negative integer case, Swift wouldn't know and would still force us to use a `default:` statement.

Also, don't mix up the parameters order: the first parameter of the infix `~=` operator (commonly named `lhs`, for *left-hand side*) is the object you're gonna use in your `case` statements. The second parameter (commonly named `rhs` for *right-hand side*) is the object you're `switch`-ing over.

Other uses of ~=

There can be plenty of other uses of `~=`

For example, we can add support for pattern matching on our `Book` struct from part 2 article:

```
func ~= (lhs: Range<Int>, rhs: Book) -> Bool {
  return lhs ~= rhs.year
}
```

Now let's test it:

```
let aBook = Book(title: "20,000 leagues under the sea", author: "Jules Ver
switch aBook {
```

```
case 1800..<1900: print("19th century book")
case 1900..<2000: print("20th century book")
default: print("Other century")
}
```

Of course, I discourage this kind of usage, as comparing a book directly with a range of integers does not make it obvious that it compares the book's year. Better switch over the `aBook.year` directly. But that's just to show the power of the `~=` operator (and because I didn't have a better example in mind 🙃).

Another usage example of `~=` could be to check if a `String` is "close enough" to another one. For example, if you're creating a quizz game and expect the player to type the answer from the keyboard but want to be case insensitive, diacritics insensitive, and even tolerate small typos, you could imagine this usage:

```
struct Answer {
  let text: String
  let compareOptions: NSStringCompareOptions = [.CaseInsensitiveSearch, .D
}

func ~= (lhs: Answer, rhs: String) -> Bool {
  return lhs.text.compare(rhs, options: lhs.compareOptions, range: nil, lc
}

let question = "What's the French word for a face-to-face meeting?"
let userAnswer = "Tete a Tete"

switch userAnswer {
case Answer(text: "tête-à-tête"): print("Good answer!")
case Answer(text: "tête à tête"): print("Almost… don't forget dashes!")
default: print("Sorry, wrong answer!")
}
// prints "Almost… don't forget dashes!"
```

See how the comparison uses a case-sensitive, diacritics-insensitive and width-insensitive comparison to be lenient about the answer?

Syntactic sugar on Optionals

But the syntactic sugar of `switch` and pattern matching doesn't stop at the transparent use of `~=` by the `switch`/`case` statement.

Another useful syntactic sugar to know when dealing with `switch` is simply `x?`. You recognize with the question mark the relation with `Optionals` of course.

In this specific context, using `x?` is syntactic sugar for `.Some(x)`. This means that you can write stuff like this:

```
let anOptional: Int? = 2
switch anOptional {
case 0?: print("Zero")
case 1?: print("One")
case 2?: print("Two")
case nil: print("None")
default: print("Other")
}
```

In fact, if you don't use `?` but write `case 2:` instead of `case 2?:` then the compiler will complain with an error like: `expression pattern of type 'Int' cannot match values of type 'Int?'` because it would be trying to match an `Int?` (`anOptional`) with an `Int` (`2`).

But using `case 2?:` is the exact equivalent of writing `case Optional.Some(2)`, which produces an `Int?` containing `2`, which can be matched against another `Int?` like `anOptional`. `case 2?:` is just a more compact form of `.Some(2)`.

Switch on enums from rawValue

Talking about this, I recently stumbled upon some code which used `enum` (with an `Int` rawValue) to organize a `UITableView` used as a Menu. This is a good idea to manipulate `enum MenuItem` instead of the `indexPath.row`, right?

```
enum MenuItem: Int {
  case home
  case account
  case settings
}
```

But then to implement each tableView row based in the `MenuItem`, the code was looking like this:

```
switch indexPath.row {
case MenuItem.home.rawValue: …
case MenuItem.account.rawValue: …
case MenuItem.settings.rawValue: …
default: ()
```

First of all, notice how the `switch` is done on an `Int` (`indexPath.row`) and then each `case` uses a `rawValue`. This is wrong for multiple reasons.

- the first being that nothing prevents you to use any other value, like a copy/pasting could make you write `case FooBar.baz.rawValue` and the compiler

won't even complain. But you're dealing with `MenuItems`, so you should leverage the compiler to ensure you only deal with `MenuItems`, right?
- the other problem is that this `switch` is not exhaustive by itself, this is why the `default` statement was necessary. I strongly recommand you to not use `default` when possible, and instead make your `switch` exhaustive, this way if you happen to add a new value to your `enum` you'll be forced to think about what to do with it instead of it being ignored or eaten up by the `default` without you realizing.

So instead of switching on `indexPath` and `case ….rawValue`, you should rather build the enum from the `rawValue` first. This way you can then only switch over `cases` that use `MenuItem` enum cases, not anything else like `FooBar.baz` or whatnot.

And to do that, because `MenuItem(rawValue:)` is a failable initializer and will in fact return a `MenuItem?`, you can leverage the syntactic sugar we discovered above!

```
switch MenuItem(rawValue: indexPath.row) {
case .home?: …
case .account?: …
case .settings?: …
case nil: fatalError("Invalid indexPath!")
}
```

Well to be honest, I rather prefer using a `guard let` for that kind of stuff, as I find it way more readable than using `?` in every `case`:

```
guard let menuItem = MenuItem(rawValue: indexPath.row) else { fatalError("
switch menuItem {
case .home: …
case .account: …
case .settings: …
}
```

But hey, it's good to know all the possible alternatives!

Conclusion

That's it for today. Next (and probably last) part of this article series will talk about using pattern matching in contexts other than `switch`, especially `if`, `guard`, but also `for` loops, and using these features in a whole new level. Can't wait!

> ⏩ *Read last part of this article series here: part 4*

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

Share this post: