Joy of Elixir

Part One: Introductions

Part Two: Getting Started

Part Three: Building on the foundations

The Appendixes

But who is this book for, really?

Elixir? Isn't that something you drink?

Elixir *the programming language* is very different from an elixir which Wikipedia defines as:

> a clear, sweet-flavored liquid used for medicinal purposes

And for the programming language variant it says:

> Elixir is a functional, concurrent, general-purpose programming language that runs on the Erlang virtual machine (BEAM).

The first explanation seems like it is written in easy-to-understand perfectly-cromulent English. The second explanation is a stream of words that your nerdy friend would espouse right after pushing up their glasses and they'd probably prefix the sentence with "Well, actually [pause for dramatic effect]".

Sure, *some* of the words you'd understand, but the rest is nerdy-gobbledegook.

What the devil does "functional" even mean? Does it mean that it works? Well I sure would hope it works!

Concurrent? What?

What even is an Erlang?

Furthermore, what is a *virtual machine* and what makes it different from a regular machine?

What is a BEAM? I've seen the Olympics, is it like the balance beam? Or is it like a light beam? Or does it somehow aid in the architecture of buildings? You'll discover later on what it means, but right now it is not as important as the description makes it seem.

It is a sentence very clearly written by one of your nerdy friend's compatriots for all the other nerdy people to understand. They'd all read the sentence and nod sagely and say stuff like "ahhh yes that is correct and succinct and easy to understand". That is because they are nerds and they *live* this stuff. You are new and we should at least make *some* effort not to scare you off with such talk!

Never mind the nerd-compatible explanations of what Elixir-the-programming-language is. Those are written for the nerds. Let's try this one on for size:

> Elixir is a programming language that humans use to ask the computer to perform certain actions.

That's a pretty good explanation. I can say that because I am writing the book. The explanation uses simple English words to describe what you would use Elixir to do. It doesn't explain its fancy features using words like "functional" and "concurrent", and it *certainly* does not mention anything about BEAMs.

The explanation is so good that it could be used to explain any programming language in existence. So it is probably *too* generic. It's like the black-and-white labelled food you find on the bottom shelf at the supermarket for a dollar. It does the job, but it doesn't quite sit well in your stomach soon after.

Ok, so let's re-vamp it! Simple Explanation Of Elixir The Programming Language, Version Two, here we go:

> Elixir is an extremely fun and easy-to-use programming language that humans use to ask the computer to do things. You can write programs that are easy-to-understand for humans and computers alike. *Writing Elixir is a joy*.

Oh yeah! That's so much better. It gets the people *going*. Speaking of which, a small crowd of a few hundred people has gathered around. "Joy? I like joy!", they shout. Then they say "we want to see some Elixir code! Enough talk!". Uh oh, the masses are getting rowdy. I better give the people what they want.

Appeasing the masses with code

OK, here's some code:

```
"Hello, World!"
```

The masses are silent. Their stares harden. Their spokesman -- who they've nominated while I thought of this snippet -- says, systematically tearing down the fourth wall: "That's just a bunch of quoted words! Just like this one that I'm speaking."

The crowd agrees.

Ok, you're right. It is some quoted words, but they're quoted words that both humans *and* computers can understand. That's pretty cool. Well, I thought so, at least.

The crowd of humans arrayed before us know the words and that the words have *meaning* behind them and form an understandable sentence. The computer knows only the words, and cares not that they have a meaning or that they form an understandable sentence. "Those things are only important to the humans", it thinks, without acknowledging the Machine Learning zeitgeist.

When we say the sentence to the masses, they can easily repeat it back to us. To speak to the computer, we need to open up a *prompt* for the particular language we want to use to converse. Diffeent languages have different prompts. In this case, we want to talk Elixir and so we can open a prompt with the `iex` command. `iex` stands for "Interactive EliXir". Let's open up one now by typing `iex` and pressing `Enter`.

When it starts, the prompt says:

```
Erlang/OTP 19 [erts-8.1] [source] ...

Interactive Elixir (1.4.2) - press Ctrl+C to exit (type h() ENTER for help)
```

We can safely ignore this output. It is just giving us some nerdy information. The last line shown here tells us that we're running Interactive Elixir "1.4.2". This tells us that we're running the 1.4.2 version of Elixir. It tells us that we can hit `Ctrl+C` to exit (stop the `iex` command). The next thing the computer says is:

```
iex>
```

This tells us that the computer is now listening for our instructions, eagerly awaiting them. It's *prompting* us for input. Let's give the computer our sentence again, pressing enter at the end of the line:

```
iex> "Hello, World!"
"Hello, World!"
```

On the first line here, we're giving the computer an instruction that looks like a regular sentence. This is because it is a regular sentence. In computer-nerd-terminology, we refer to this double-quoted collection of symbols as a *string*. It's easier to think of it like a string if you think of all of the sentence's parts being connected by an actual string:

TODO: String image goes here.

The computer then takes in this string, interprets it and tells us how it interpreted the string. In this case, the computer is just parroting our sentence/string back to us. The computer can do a lot more than this, believe me. Computers wouldn't be very good if they all they did was parrot back to us.

The masses are now getting fidgety. And the computer is prompting us with this line.

```
iex>
```

The computer hungers for more input. The masses want to see more of what this language can do. Fortunately for me (and you), computers can do much more than parrot.

"We just saw a REPL!" "A what now?"

Nerdier types would call this "prompt" of ours a REPL. If those nerdier types have been using the word "REPL" to describe prompts like iex, you now know what they mean. REPL = prompt. You know that it has nothing to do with descending from a height suspended by a double coiled rope (rappel).

"But hey REPL looks like an acronym!", I hear you say. Gold star for you. It *is* an acronym and it stands for **R**ead, **E**valuate, **P**rint, and **L**oop.

The `iex` prompt prompts us for input. This is the "read" step. Once we give it input, it then *evaluates* what we've given it to determine if the computer is capable of running our program. If it is, then the computer runs our program and "*prints*" the output back to the prompt to show us what the computer has done. And then it prompts us again because it is *looping*. Hence: read, evaluate, print, loop: REPL.

Mathematical!

How about we ask the computer to do some mathematical equations?

```
iex> 2 + 4
6
iex> 3 - 6
-3
iex> 4 * 12345
49380
iex> 1234 / 4 + 2 - 12 * 3
274.5
```

This appeases the masses, slightly. They're a fickle bunch. The computer is now no longer parroting things back to us. It's instead calculating the not-so-advance mathematical equations we're giving it, and then giving us the right numbers.

Why are numbers yellow, but strings green?

You may have noticed that a string shows up as green in the `iex` prompt:

```
iex> "Hello World!"
"Hello World"
```

But numbers show up as yellow:

```
iex> 2 + 4
6
```

And you might've wondered why.

Well, there is no particular reason for strings being green and numbers being yellow; the colours don't mean much else than to serve as useful indicators of the different kinds of data. Strings are one kind of data, and numbers are a different kind. We'll see more kinds as we continue on our journey.

Green doesn't mean that strings are somehow more special than numbers, and yellow doesn't mean that numbers are more important. They're just colours to help differentiate the two at a glance.

They realise that Elixir is now built for more things than simple parroting. Elixir can do calculations too! We've used the symbols `+`, `-`, `*`, and `/` here, asking the computer to add, subtract, multiply and divide, in that order.

Exercises

- Get Elixir to calculate the number of seconds in the day by multiplying the hours in a day by 60 twice. How many seconds are there in a day?
- Calculate the average of these numbers: 4, 8, 15, 16, 23 and 42.

Now, where did I put that value?

We've now seen that Elixir can handle words and numbers easily. But what else can it do? Well, it can remember things.

```
iex> sentence = "A really long and complex sentence we'd rather not repeat."
"A really long and complex sentence we'd rather not repeat."
iex> score = 2 / 5 * 100
40
```

As long as we leave `iex` running, Elixir will remember that `sentence` is `"A really long and complex sentence we'd rather not repeat."` and `score` is `40`. At any point in time, we can ask it what `sentence` or `score` is and it will tell us:

```
iex> sentence
"A really long and complex sentence we'd rather not repeat."
iex> score
40
```

`sentence` and `score` here are *variables*, and they're given that particular name because the thing that the computer remembers with that name of `sentence` or `score` can *vary*. We can tell the computer to forget its previous definition of `sentence` and give it a new one like this:

```
iex> sentence = "An even longer and significantly more complex sentence
   that we might be ok with repeating, if the mood takes us."
"An even longer and significantly more complex sentence
   that we might be ok with repeating, if the mood takes us."
```

Then when we ask the computer what `sentence` is, it will have forgotten the old `sentence` and will only know the new one:

```
iex> sentence
"An even longer and significantly more complex sentence
   that we might be ok with repeating, if the mood takes us."
```

Now the masses are looking chirpier. Some are even smiling! Isn't that wonderful? Let's create another variable called `place`:

```
iex> place = "World"
"World"
```

The computer will now remember that `place` is `"World"`. "But what's the use of just *setting* a variable like this? Why make the computer remember at all?", Roberto (the spokesman) says. By the way, we have since learned the spokesman's name is Roberto, and so we've done some variable creation of our own in our own brain: `spokesman = "Roberto"`. Woah.

Ok, Roberto's right. We should do something meaningful with this variable. Ok, Roberto, how about this? *[cracks knuckles]*

```
iex> "Hello #{place}!"
"Hello, World!"
```

The masses go nuts. It's pandemonium! Then after a few shushing motions with my hands, they're (mostly) quiet again.

"What just happened?", asks Roberto? But *you*, Dear Reader, had asked that question already. Roberto didn't hear because of the crowd and the *sheer, unbounded* pandemonium.

What on earth are those funky characters around `place`? You can think of them as a placeholder. Nerdier-types would call it *interpolation*. It tells the computer that we want to put the `place` variable *riiiiight* here, right after `Hello` and the space, and right before `!`.

The computer takes the sentence that we give it, realises that there's that funky `#{}` placeholder indicator, and it remembers what `place` was and puts it right there. Pretty nifty, eh?

Exercises

- If we store the number of seconds in a day using this code: `seconds = 86400`, calculate using that variable how many seconds there are in 30 days.
- Create a variable called `name`, store a string in it and place the value of that variable in another string.
- The line `5 / "four"` shows an error. Think about why this error might happen.

Making lists

But what *else* can Elixir do? We've seen that it can handle strings and it can handle mathematical equations with ease and it can remember things, but what else? Well, what if we had a *list* of things that we wanted the computer to remember? And we wanted the computer to remember this list of things in a particular order? We could do this by assigning each of the things to a unique variable:

```
iex> first = "fish"
"fish"
iex> second = "ham"
"ham"
iex> third = "eggs"
"eggs"
```

But then we would need to remember that the things we want are stored in the variables `first`, `second` and `third`. And what if we -- as humans -- forget what position we were up to?

```
iex> fifth = "bread"
"bread"
```

Disasterous! This simply won't do.

A better solution to getting the computer to remember a list in Elixir is to actually tell it what we're storing is a list. To do this, we wrap our list in square brackets (`[]`) and separate each item in the list with a comma:

```
iex> shopping_list = ["fish", "ham", "eggs", "bread"]
["fish", "ham", "eggs", "bread"]
```

The computer will now remember our entire shopping list in one variable, rather than remembering it across multiple variables. Now we have a one-stop-shop that we can go to for our shopping list.

Roberto, impressed with Elixir's ability to keep a list of anything his heart desires, has started taking the names down of the people who are in the crowd. His computer materialised from nowhere, or perhaps from somewhere in the throng nearby. It was too quick to really notice. He starts taking down names, fingers flying across the keyboard:

```
iex> those_who_are_assembled = [
"Roberto",
"The Author",
"The Reader",
"Juliet",
"Mary",
"Bobalina",
"Charlie",
"Charlie (no relation)"
]
["Roberto", ...]
```

Do you see what Roberto did there?

He started a list on one line and then continued it on the lines following. When he finished his list, the computer then took *all* the lines, processed them as one big instruction and presented us with the finished list. Elixir will let us write code over multiple lines like this, which is handy because it helps for readability!

And so Roberto continues. He disappears into the crowd directly to our right, and somehow re-appears within mere seconds on our left. This is despite the density of the crowd. Amazing stuff. He then asks: "Hey Mr. Author Guy, I want to also track people's ages so that I can do some statistics on who exactly is assembled here. Oh, and I want to track their gender too. But a list doesn't seem very suitable for this. How can I do this?"

Well Roberto, there's more than one way you could do this. If all you wanted to track about a person was their name, their age and their gender, you could make each item in the list its own list:

```
iex> those_who_are_assembled = [
...> ["Roberto", "30ish", "Male"]
...> ]
```

These lists-inside-of-lists would be called *sub-lists*. You would then have to keep in mind that the first item in each sublist is the name, the second is the age and the third is the gender. What if you lost track of this and started adding in people, but you flipped the age and their gender?

```
iex> those_who_are_assembled = [
...> ["Roberto", "30ish", "Male"],
  ... a long time passes ...
...> ["Young Roberto", "Male", "20ish"],
...> ]
```

Pandemonium again! We need a better way of enforcing what data we're collecting for each person. We need a better data structure than lists within lists. Read on to find out what that data structure is!

Crafting maps

Back in the last chapter, Roberto tried tracking information about the people assembled using some lists within another list:

```
iex> those_who_are_assembled = [
...> ["Roberto", "30ish", "Male"],
  ... a long time passes ...
...> ["Young Roberto", "Male", "20ish"],
...> ]
```

But as we can see here, the order of the items within the list can be input incorrectly due to human error. We need something that helps prevent human error like this, and if that *something* could helpfully indicate what each item in the list was -- by giving it a kind of a name -- then that would be good too.

For that, we can use a *map*. Just like a regular map, maps in Elixir can tell us where to find things, if only we knew where to look.

Let's look at how we could collect a single person's data using a map:

```
iex> person = %{"name" => "Roberto", "age" => "30ish", "gender" => "Male"}
%{"age" => "30ish", "gender" => "Male", "name" => "Roberto"}
```

In a map, we structure data using *keys* and *values*. The thing to the *left* of the `=>` is called a *key* and the thing to the *right* is called a *value*.

You might notice here that the computer has taken in our map and returned the keys in a different order to the one that we specified. The order of keys doesn't matter in a map at all, unlike in lists where order *does* matter. We could specify the keys in any order we wish, and the map would be the same:

```
iex> person = %{"gender" => "Male", "age" => "30ish", "name" => "Roberto"}
%{"age" => "30ish", "gender" => "Male", "name" => "Roberto"}
```

You may now be thinking about how to get the data back out of a map once you've put it in. In order to access a value from a map we need to know the corresponding key. Once we know the corresponding key, then we can use `[]` to pluck that value out of the map. Think of it like the claw from a skilltester, diving in to pick out the value.

TODO: Skilltester image goes here.

For instance, if we wanted to get out the value of `"name"` for `person` we can do:

```
iex> person["name"]
"Roberto"
```

And if we want to get the age, we would use the `"age"` key:

```
iex> person["age"]
"30ish"
```

Roberto lets out a thoughtful "Mmmmmmmmmm" in something very close to an agreement. He likes maps. Now Roberto can know the exact data that we're collecting about the assembled masses, without a concern for how the data ordered, and then use that for later on. We don't know yet what Roberto has in mind for the data, but he's collecting it for a reason. Or at least, it seems that way.

To collect data about all the people assembled here before us, we can create a *list of maps*:

```
iex> [
...> %{age: "30ish", gender: "Male", name: "Roberto"},
...> %{age: "30ish", gender: "Male", name: "The Author"},
...> ]
```

Roberto excitedly disappears into the crowd again with this new knowledge.

Funky functions

After about half an hour, Roberto re-appears from seemingly nowhere. His computer screen glows with the information he has collected from the crowd. We're not quite sure how long he's been back for, but the moment our attention fixes on him, he speaks: "Okay, so you can represent different... *kinds* of things in Elixir -- strings, numbers, lists, maps -- but what is the point of doing that at all? What can you *do* with them? Do I need to write code to work with these different kinds of things all the time? Can the computer remember code too?"

Well Roberto, that was a very clever and completely unintentional segue into this next part of the book. Thank you for doing that.

Yes, in fact you can tell the computer to remember some code too. This saves *a lot* of typing. Crazy amounts of the stuff. You'll get *years* of your life back!

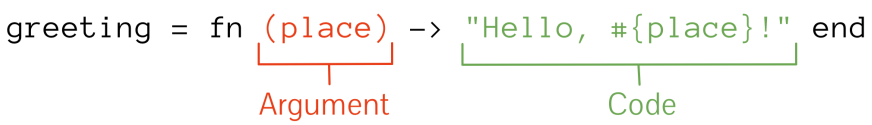
Ok, enough chit-chat. Let's look at one way we can use to make the computer remember some code:

```
iex> greeting = fn (place) -> "Hello, #{place}!" end
#Function<6.52032458/1 in :erl_eval.expr/5>
```

This is called a *function* and we can write whatever code we want the computer to remember for later, just like we could tell the computer to remember a string, a number, a list or a map. Elixir code is all about the functions, and now we understand why Wikipedia said it was a "functional" language. It didn't mean that it was functional in the sense that it is operational, but moreso that it uses functions to get things done. You'll be seeing a lot of functions in Elixir code from here on out.

The `fn` tells the computer we're about to define a function -- because typing `function` each time would just be too much for us to bear -- and the defining-of-the-function doesn't stop until it gets to the `end`. Just like you wouldn't stop until you got to the end of something -- i.e. a book -- right?

The `(place)` here defines an *argument* for the function; think of it like a variable that is only available within this function and not the angry shouting matches that most `normal` arguments can devolve into. The `->` tells the computer that we're done defining the arguments for the function, and anything after this but before the `end` is going to be the code to run.



Once we hit enter at the end of this line, the computer gives us some output to indicate it has accepted our function. It's not the friendliest output, but at least it's something that tells us the computer has done *something*.

```
iex> greeting = fn (place) -> "Hello, #{place}!" end
#Function<6.52032458/1 in :erl_eval.expr/5>
```

We've assigned our function to the `greeting` variable and so that's what the computer will remember the function as. "How do we make the computer run that function?", Roberto asks keenly, suddenly impressed about the computer's ability to remember functions. I'm sure you're asking the same thing, dear reader.

Well, Roberto (and dearest reader), we need to use some new and exciting code that we've not seen yet:

```
iex> greeting.("World")
"Hello, World!"
```

This is how we make the function run. We run this function by putting a dot after its name. The brackets after this dot represent the argument for the function. This time that we call the function, `place` will be `"World"`. But it doesn't always have to be `"World"`. It can be anything you wish:

```
iex> greeting.("Mars")
"Hello, Mars!"
iex> greeting.("Narnia")
"Hello, Narnia!"
iex> greeting.("Motherland")
"Hello, Motherland!"
```

Each time we run the function here we give it a new value for `place`. We don't have to set `place` ourselves, as the function takes care of that. But what of our `place` variable from yesteryear (line 9 in `iex`)? Has that changed?

```
iex> place
"World"
```

The computer knows that the `place` from *outside* the function is different to the place *inside* the function, and so it keeps the two separate. The computer is smart enough to know that `place` on the outside may not exist and so it shouldn't rely on it being present. The function contains everything it needs to run inside itself.

One more handy thing about functions is that they're not limited to just one argument. You can define a function that accepts as many arguments as you wish:

```
iex> greeting = fn (name, gender, age) ->
...>   "Hello, #{name}! I see you're a #{gender} and you're #{age} years old."
...> end
#Function<18.52032458/3 in :erl_eval.expr/5>
```

This function has 3 arguments, and so when we run it we need to give it all three. To do that, we just separate them using a comma, similar to how we separated the items in a list earlier, just without the square brackets (`[]`) around these arguments.

```
iex> greeting.("Roberto", "Male", "30ish")
"Hello, Roberto! I see you're a Male and you're 30ish years old!"
```

We can go nuts with the number of arguments that a function is defined with. However, we need to take a modicum of caution when running the functions. If we specify the wrong number of arguments, Elixir will tell us off with big red text:

```
iex> greeting.("Roberto")
** (BadArityError) #Function<18.52032458/3 in :erl_eval.expr/5>
  with arity 3 called with 1 argument ("Roberto")
```

Oh no the computer is angry with us. Well, if the computer felt any emotion from brutal indifference it would *probably* be some version of angry, or at least disappointed. The computer is reprimanding us: telling us that we caused a `BadArityError` and that means that we had a "<function> with arity 3 called with 1 argument".

"What on earth is an 'arity'?", Roberto asks, clearly flummoxed (and perhaps a bit affronted) by the word. Unlike the computer, Roberto is not brutally indifferent when it comes to these things. After all, Roberto thought he was getting a handle on this Elixir thing.

Arity is a fancy computer term which means "arguments". This error is saying that while the `greeting` function is defined with 3 arguments ("with arity 3") we're only running ("calling") it with 1 argument. Helpfully, it tells us what arguments we tried to give the function. To avoid the computer reprimanding us we should make sure to call functions with the right number of arguments.

Saving code for later

While it's all good and well to run code through the `iex` prompt, you may want to save some code to run later on. Maybe at this point you might have an idea of something to try out and you want to save it *somewhere* for later on because an `iex` prompt will only remember stuff as long as it is open. Once it's shut, that code is gone for good and you will need to type it all out again.

To save your Elixir code, you can create a new file in your favourite text editor, put in the code you want and then save it with a name like `hello.exs`. That `.exs` on the end symbolises that the file is an Elixir Script file.

To run the code inside the file, simply run `elixir hello.exs`. No need for a prompt now!

Exercises

- Make a function which returns the number of seconds in the specified amount of days. For example, `seconds.(2)` should tell us how many seconds there are in 2 days.
- Make a function which takes two maps with `"age"` keys in them and returns the average age.
- Save either of these two solutions in their own file and run them through the `elixir` command-line tool.

Pattern matching

Back in Chapter 2 you saw that the equals sign (=) made the computer remember things.

```
iex> sentence = "A really long and complex sentence we'd rather not repeat."
"A really long and complex sentence we'd rather not repeat."
iex> score = 2 / 5 * 100
40
```

While this is indeed still true, the equals sign (=) can do more than just set one value at a time. (Roberto double-takes at the last sentence, while the crowd murmurs.) There's a hidden feature of Elixir that we haven't shown yet, and that feature is called *pattern matching*. You'll use this feature quite a lot when programming with Elixir -- just as much as functions! -- so we'll spend a while talking about it here in this chapter too.

Equals is not just for equality

The equals sign isn't just about assigning things to make the computer remember them, but it can also be used for *matching* things. You can think of it like the equals sign in mathematics, where the left-hand-side must match the right-hand-side.

For instance, if we tried to make `2 + 2 = 5`, much like Nineteen Eighty Four's Party would want us to believe, Elixir would not have a bar of it:

```
iex> 2 + 2 = 5
** (MatchError) no match of right hand side value: 5
```

Unlike the famous Mr. Winston Smith, Elixir cannot ever be coerced into disbelieving reality. Here, Elixir is telling us that `2 + 2` is indeed not 5. In Elixir, the left-hand-side has to evaluate to the same as the right-hand-side. This will make the computer happy:

```
iex> 2 + 2 = 4
4
```

Similarly, having two identical strings on either side of the equals sign will also make the computer happy:

```
iex> "dog" = "dog"
"dog"
```

Let's do something more complex than having the same thing on both sides of the equals sign. Let's take a look at how we can pattern match on lists.

Pattern matching with lists

Let's say we have a list of all the people assembled here, like we had back in Chapter 3:

```
iex> those_who_are_assembled = [
"Roberto",
"The Author",
"The Reader",
"Juliet",
"Mary",
"Bobalina",
"Charlie",
"Charlie (no relation)"
]
["Roberto", ...]
```

And let's also say that we wanted to grab the names of the first 3 people in this list, but then ignore the remainder -- given that the first three are clearly the most important people here. We can use some pattern matching on this list:

```
iex> [first, second, third | others] = those_who_are_assembled
```

With this code, we're telling Elixir to assign the first, second and third items from the list to the variables `first`, `second` and `third`. We're also telling it to assign the remainder of the list to the `others` variable, and we specify that by using the pipe symbol (`|`). In this code we've selected the first 3 items from the list, but we could also just select the very first one, or the first 5. It doesn't have to be exactly 3.

We can check what this has done exactly in `iex` by looking at the values of each of these variables:

```
iex> first
"Roberto"
iex> second
"The Author"
iex> third
"The Reader"
iex> others
["Juliet", "Mary", "Bobalina", "Charlie", "Charlie (no relation)"]
```

"Does this mean that I could do the same for the `others` list to get the next 3 people?", Roberto asks. Yes it does mean that you can do that:

```
iex> [first, second, third | remainder] = others
"Roberto"
```

Now when we check the values of `first`, `second` and `third` they'll be the names of the next 3 people in the list:

```
iex> first
"Juliet"
iex> second
"Mary"
iex> third
"Bobalina"
```

And our `remainder` variable will be the remaining names in the list, which are just the two Charlies:

```
iex> remainder
["Charlie", "Charlie (no relation)"]
```

If we now try to pull out the next 3 people in the list, Elixir won't be able to do that because there are only two names left in the `remainder` list:

```
iex> [first, second, third | those_still_remaining] = remainder
** (MatchError) no match of right hand side value: ["Charlie", "Charlie (no relat
```

It's always a good idea to be careful here with pattern matching lists with the right number of expected items to avoid `MatchErrors` like these.

That's enough about lists for now. We'll revisit pattern matching them a little later on in this book. Let's look at how we can work with maps.

Pattern matching with maps

Let's say that we have a map containing a person's information:

```
iex> person = %{"name" => "Roberto", "age" => "30ish"}
%{"name" => "Roberto", "age" => "30ish"}
```

We've seen before that we could pull out the value attached to `"name"` or `"age"` at whim:

```
iex> person["name"]
"Roberto"
iex> person["age"]
"30ish"
```

But what if we wanted to pull out both of these values *at the same time*? Or even in the shortest possible code? Well, for that we have this *pattern matching* thing that I've been banging on about for over a page. Let's take a look at how pattern matching can help get both the name and the age out of the `person` map.

```
iex> %{"name" => name, "age" => age} = person
```

"Hey, what gives? The left-hand-side here is *clearly* not the same as the right hand side!", cries Roberto. Yes, you're absolutely right. On the right hand side here we have a map which has a `"name"` key which points to a value of `"Roberto"`, but on the left hand side that `"name"` key points to a value of `name`. This is a trick of pattern matching: the left-hand-side can be used to assign multiple variables; it doesn't have to match the right-hand-side exactly.

If we check the value of `name` and `age` here, we'll see that those values are the values from our map.

```
iex> name
  "Roberto"
  iex> age
  "30ish"
```

Well, would you look at that? We were able to pull out these two values *at the same time*. The crowd cheers as if we've just performed a magic trick. Just wait until you see our next trick!

Pattern matching inside functions

Pattern matching can be used for even more things than picking out the keys of a map or the items out of a list. We can also use it when we define our functions too, making them respond differently depending on the arguments passed. For instance, we could define a function which took the kind of road that we took; either the "high" road or the "low" road, and get it to respond differently depending on which was passed.

```
iex> road = fn
  "high" -> "You take the high road!"
  "low" -> "I'll take the low road! (and I'll get there before you)"
end
```

When we call this function with the `"high"` argument, that argument will *match* the first function line here, and "You take the high road" will be returned. Similarly, when we give it `"low"` it will return "I'll take the low road! (and I'll get there before you)". Each line inside the function here is called a *clause*. We could keep talking about the theory behind this, or we could actually try it in our `iex` prompt:

```
iex> road.("high")
"You take the high road!"
iex> road.("low")
"I'll take the low road! (and I'll get there before you)"
```

This works because of how we've defined the `road` function. In that function, we've defined two separate function *clauses*. The first function clause says that when the argument is `"high"` then the function should output the line about the "high road". The second function clause says that when we supply the `"low"` argument then it should output the line about the "low road".

Think of it like this: Elixir is pattern matching the value of the argument against the clauses of the function. If Elixir can see that the argument is equal to `"high"` then it will use the first function. If it isn't equal to "high", then Elixir will try matching against `"low"`. If the argument is `"low"` then the second clause will be used.

But what happens if it's neither? We can find out with a touch of experimentation in our iex prompt:

```
iex> road.("middle")
** (FunctionClauseError) no function clause matching
  in :erl_eval."-inside-an-interpreted-fun-"/1
```

Elixir here is showing us an error that we've never seen before: a `FunctionClauseError`. This error happens when we call a function giving it arguments that it doesn't expect. In the case of our `road` function, it's expecting either `"high"` or `"low"`, but we gave it `"middle"`. Both clauses of the function don't match the value of "middle", and so we see this error.

Matching on strings in functions is great, but as we saw earlier with the equals sign (=) we can match on more than just strings.

Matching maps inside functions

We can also match on the keys contained within a map and get the code to act differently depending on what keys are present. Let's take our `greeting` function from Chapter 5 and modify it slightly so that behaves differently depending on what kind of map we pass it:

```
iex> greeting = fn
  %{"name" => name} -> "Hello, #{name}!"
  %{} -> "Hello, Anonymous Stranger!"
end
```

"Oooh that's fancy! What is the empty map is for?", Roberto asks. Soon, Roberto. Soon. Let's see what happens if we call this `greeting` function with a map which has a `"name"` key:

```
iex> greeting.(%{"name" => "Roberto"})
"Hello, Roberto!"
```

Here, the first function clause is matching because the map we're supplying contains a key which is `"name"`, and that's what the first function clause (highlighted below in green) expects too: a map which has a key called `"name"`. So when we call this function with this map with a `"name"` key, we see the string `"Hello, Roberto!"` output from the function.

```
iex> greeting = fn
  %{"name" => name} -> "Hello, #{name}!"
  %{} -> "Hello, Anonymous Stranger!"
end
```

Now let's see what happens if we call this function with an empty map:

```
iex> greeting.(%{})
"Hello, Anonymous Stranger!"
```

Elixir is still acting as we would expect it to: we supplied an empty map and the second function clause matches an empty map, and so that's the clause that will be used here instead.

```
iex> greeting = fn
  %{"name" => name} -> "Hello, #{name}!"
  %{} -> "Hello, Anonymous Stranger!"
end
```

Ok, so what would you expect to happen here if you supplied neither a map with a `"name"` key or an empty map, but a map with a different key in it? "Based on the string test, I would expect it to fail with a FunctionClauseError!", Roberto proudly proclaims. Looks like someone has been paying attention. Dear Roberto, that is what I expected to happen too when I learned Elixir. However, maps are matched differently to strings in Elixir. Let's look:

```
iex> greeting.(%{"age" => "30ish"})
"Hello, Anonymous Stranger!"
```

The `greeting` function still displays "Hello, Anonymous Stranger!" So what gives here?

Well, in Elixir when you match two maps together it will always match on subset of the map. Let's take a look using our trusty equals sign (`=`) again:

```
iex> %{} = %{"name" => "Roberto"}
%{"name" => "Roberto"}
```

Just like in the second clause from the function above, we're comparing an empty map on the left-hand-side to a map from the right hand side. When pattern matching maps like this, it's helpful to think of the left-hand-side showing the keys that are absolutely required for the match to work. The right-hand-side *must* contain the same keys as the left-hand-side, but the right-hand-side can contain more keys than what's on the left.

This match will succeed because there are no keys required by the left-hand-side of this match. This story is different if we've got a map on the left-hand-side with keys, as we've seen before with the first clause of our `greeting` function:

```
iex> greeting = fn
  %{"name" => name} -> "Hello, #{name}!"
  %{} -> "Hello, Anonymous Stranger!"
end
```

In the first clause's case, it will only match if the argument passed to the `greeting` function is a map which contains a `"name"` key; this key is *required* by the match. If the map does not contain a `"name"` key then this clause will not match. The second clause matches *any* map, and so that is the clause that will be used for any map not containing a `"name"` key.

Matching anything

Now one more thing I wanted to show you is how to avoid those pesky `FunctionClauseErrors` when all the function clauses inside a function don't match. Let's take a look at the `road` function again:

```
iex> road = fn
  "high" -> "You take the high road!"
  "low" -> "I'll take the low road! (and I'll get there before you)"
end
```

If the argument supplied to this function is neither `"high"` nor `"low"` then Elixir shows us a `FunctionClauseError`:

```
iex> road.("middle")
** (FunctionClauseError) no function clause matching in :erl_eval."-inside-an-int
```

What if instead of this error we could get the function to tell whoever was running it that they have to supply either "high" or "low" as the argument? Elixir allows us to do this by using an underscore (_) as the argument in a new function clause:

```
iex> road = fn
  "high" -> "You take the high road!"
  "low" -> "I'll take the low road! (and I'll get there before you)"
  _ -> "Take the 'high' road or the 'low' road, thanks!"
end
```

This underscore (_) matches *anything* and is often used in cases like this where we want to show a message if no other function clause matches. Let's see this in action:

```
iex> road.("middle")
"Take the 'high' road or the 'low' road, thanks!"
```

This underscore doesn't just match strings, but it will match any other argument that we can pass to the function:

```
iex> road.(%{})
"Take the 'high' road or the 'low' road, thanks!"
iex> road.(["high", "low"])
"Take the 'high' road or the 'low' road, thanks!"
```

This helps keep people in line when it comes to providing the right arguments to the function, without Elixir blowing up in their face when they provide the wrong ones.

Speaking of functions: did you know Elixir has some built-in functions that you could use?

Exercises

- Make a function that could take either two or three maps containing the key "age", and averages the ages. This is similar to an exercise in the previous chapter.
- Make a function that takes either a map containing a "name" and "age", or just a map containing "name". Change the output depending on if "age" is present. What happens if you switch the order of the function clauses? What can you learn from this?

Intermission: Recap

Elixir would be pretty hard to do if you had to write all the code for your programs by yourself. We have already seen that if you want to add, subtract, multiply or divide numbers, Elixir has your back:

```
iex> 2 + 4
6
iex> 3 - 6
-3
iex> 4 * 12345
49380
iex> 1234 / 4
308.5
```

We also saw that it can run code inside of some other code, through string interpolation:

```
iex> place = "World"
iex> "Hello #{place}!"
"Hello, World!"
```

We saw that it can handle lists:

```
iex> ["chicken", "beef", "and so on"]
["chicken", "beef", "and so on"]
```

And it can also handle maps:

```
iex> person = %{"name" => "Roberto", "age" => "30ish", "gender" => "Male"}
  %{"age" => "30ish", "gender" => "Male", "name" => "Roberto"}
```

We also saw that it could remember functions for us:

```
iex> greeting = fn (name) -> "Hello, #{name}!" end
#Function<6.52032458/1 in :erl_eval.expr/5>
```

And in the last chapter we saw that we could do pattern matching:

```
iex> %{"name" => name, "age" => age} = %{"name" => "Roberto", "age" => "30ish"}
%{"name" => "Roberto", "age" => "30ish"}
iex> name
"Roberto"
iex> age
"30ish"
```

But what if I told you that it could do more than this simple math, string interpolation, remembering functions and pattern matching? What if I told you that Elixir already provided some functions that used all of the above things to aid you in building your programs if only you had asked to use them?

Well, all you need to do is ask and Elixir will provide.

Built-in functions: strings, input and output

Working with strings

Reversing a string

Have you ever wanted to reverse a sentence, but didn't want to type all the different characters yourself? Elixir has a handy function for this called `String.reverse`. Here it is in action:

```
iex> String.reverse("reverse this")
  "siht esrever"
```

Roberto elicits a noticeable "wooooowww, that's so cool" at this. "What else does Elixir have?", he asks quickly. We'll get to that, Roberto. Let's take the time now to understand what's happening here first.

The functions that Elixir provides are separated into something akin to kitchen drawers or toolboxes, called *modules*. Whereas in your top kitchen drawer you might have forks, knives, sporks, and spoons (like every sensible person's kitchen does), and in another you might have measuring cups, and in another tea towels, in Elixir the functions to work with the different kinds of data are separated into different modules. This makes finding functions to work with particular kinds of data in Elixir very easy.

Here, we're using the `reverse` function from the `String` *module* ("drawer" / "toolbox"). We're passing this `String.reverse` function one argument, which is a string `"reverse this"`. This function takes the string and flips it around, producing the reversed string: `"siht esrever"`.

Note here how we *don't* need to put a dot between the function name and its arguments, like we had to do with the functions we defined ourselves. You don't need to do this when you're running a function from a module. You only need the dot if you've defined the function and assigned it to a variable. For instance, with our old `greeting` function:

```
iex> greeting = fn (place) -> "Hello, #{place}!" end
#Function<6.52032458/1 in :erl_eval.expr/5>
greeting.("world")
```

When calling the `String.reverse` function, Elixir knows that it's a function because of that `String.` prefix. We don't need a dot right after the function name:

```
iex> String.reverse("reverse this")
"siht esrever"
```

Splitting a string

What about if we wanted to split a string into its individual words? For that, we can use the `split` function:

```
iex> String.split("split my string into pieces")
["split", "my", "string", "into", "pieces"]
```

We now have a list of the words in the string. We'll see what we could do with such a list a little later in the chapter. For now, let's look at what else is in this `String` module.

Making all the letters of a string uppercase

What about if we wanted to make the computer turn a string into its shouty variant? We can use `upcase` for this:

```
iex> String.upcase("not so quiet any more")
"NOT SO QUIET ANY MORE"
```

Making all the letters of a string lowercase

At the opposite end of that particular spectrum, there is `downcase`:

```
iex> String.downcase("LOUD TO QUIET")
"loud to quiet"
```

So as you can see, the `String` module has some helpful functions that can help us whenever we need to split a string, turn it all into upper / lower ("down") case. There's plenty more functions in the `String` module, and we'll see some of those in due time.

Let's take a look at working with something brand new that we have never worked with before: input and output.

Input and output

Input and output are two fundamental things that you'll work with while programming. Programming is all about taking some data as an input and turning it into some form of an output. We've seen this multiple times already with the functions we've defined and used throughout this book. For instance, in that `String.downcase` function just above, the string `"LOUD TO QUIET"` is the input, and the `"loud to quiet"` generated by the method is the output.

What we'll cover in this section is getting some input from a different source: a new prompt. We'll prompt the user for their name and then we will use whatever they enter to output a message containing that input.

Making our own prompts

Let's say that we wanted to prompt people for their names and we wanted to prompt them in a way that meant that they didn't have to read Joy of Elixir to understand that strings had to be wrapped in double quotes and that they had to enter their input into an `iex` prompt.

Fortunately for us, Elixir has a module that provides us a function to do just this. That module is called `IO` (Input / Output) and the function is called `gets`. The name `gets` means "get string" and it will allow us to do exactly that. Let's see this function in practice:

```
iex> name = IO.gets "What is your name?"
  What is your name?
```

"Hey what happened to our `iex>` prompt?", Roberto asks. Good question! We're using `gets` and passing it a string. This string then becomes a new prompt. This prompt is asking us for our name. Let's type in our name and press enter:

```
iex> name = IO.gets "What is your name?"
What is your name? The Reader
"The Reader\n"
```

Ok, so there's some output here. But what does it mean? If we check our `name` variable's contents we'll see that it contains this `"The Reader\n"` string.

```
iex> name
"The Reader\n"
```

Roberto continues asking questions: "What's that `\n` on the end?". That is a *new line character* and tells the computer that we pressed enter. While the `IO.gets` function stopped prompting us after we pressed enter, it still kept the enter in case we wanted it too. In this particular case we don't really want that character. We can get rid of it by using another function from the `String` module, called `trim`.

```
iex> name = String.trim(name)
  "The Reader"
```

That's much better! Now we have our name with that pesky new line character suffixed. What `String.trim` does is remove all the extra spacing from the end of a string, giving us just the important parts.

Taking input and making it output

We've now got some input, but what's the point of taking input if you're not going to *do* anything with it? So let's do something with it! What we'll do with this input is to output a greeting message.

Lets deviate here from using the `iex` prompt and instead write our code inside one of those Elixir Script (`.exs`) files we mentioned back at the end of Chapter 5. Let's call this file `greet.exs` and put this content inside of it:

```
name = IO.gets "What is your name? "
age = IO.gets "And what is your age? "
IO.puts "Hello, #{String.trim(name)}! You're #{String.trim(age)}? That's so old!"
```

Well that's a bit sneaky of that `IO.puts` to just appear out of *nowhere*! Just like `gets` means "get string", `puts` means "put string". This function will generate some output when our script runs. If we didn't have this `IO.puts` here, our program would only take input, and it would not generate any output.

In this function we're interpolating the output of the `String.trim` function twice. Remember: we're doing this to remove the new line character (`\n`) from the result of the `IO.gets` calls.

There's some more new syntax that we've never seen before either. We've seen that we could interpolate variables into strings, but we've never seen that we could call functions while interpolating too. It's absolutely something you can do in Elixir. When interpolating inside a string you can put any code inside the interpolation brackets (`#{}`) — but as a general rule-of-thumb it's good to keep this interpolated code as short and simple as possible.

Let's run our `greet.exs` script now. First, we'll need to stop our `iex` prompt, which we can do by pressing `Ctrl+C` twice. Then we can run the script with this command:

```
elixir greet.exs
```

Here's what we'll see initially:

```
What is your name?
```

The script is prompting us for our name and it is doing that because the first line of code in that script is running the `IO.gets` function. Let's enter our name again and press enter.

```
What is your name? The Reader
And what is your age?
```

This little script is now prompting us for our age. This is because the second line is calling another `IO.gets`. Let's enter our age and then press enter again,

```
What is your name? The Reader
And what is your age? 30ish
Hello, Ryan! You're 30ish? That's so old!
```

Our script gets to the third and final line, where it runs the `IO.puts` function and outputs its little teasing message. Apparently being 30ish is old! Kids these days have no respect.

This is just a small example of what we can do with `IO.gets` and `IO.puts`. We could use any number of `IO.gets` and `IO.puts` function calls to build up a program that took user input and generated some output from it.

Exercises

- Make a program that generates a very short story. Get it to take some input of a place, an action and an object and combine all three into a little sentence.
- Ponder on what happens when you remove the `IO.puts` from the beginning of Line 3 in `greet.exs` and then run the program with `elixir greet.exs`. Think about how this would be different if you put that code into an `iex` prompt.

We've done a lot of work with strings so far in this chapter. Let's look at lists and maps again in the next chapter and the built-in functions that we can use with them.

Built-in functions: lists, maps and more

Working with lists

We looked earlier at how we could reverse a string, but how about if we wanted to reverse something else, like this list?

```
iex> animals_or_derivatives_of_animals = ["cat", "dog", "cow", "turducken"]
```

We wouldn't reach into the `String` drawer to work with a *list*, because strings are not lists and lists are not strings! Just like we wouldn't reach into the top drawer in your kitchen to pull out the milk: it's in the fridge, silly!

We should reach into the `List` drawer instead, right? Let's try that:

```
iex> List.reverse(animals_or_derivatives_of_animals)
List.reverse(animals_or_derivatives_of_animals)
** (UndefinedFunctionError) function List.reverse/1 is undefined or private
    (elixir) List.reverse(["cat", "dog", "cow", "turducken"])
```

Uh oh, there's that red text again. The computer is telling us that Elixir doesn't know about a function called `List.reverse`, or the function is "*private*". The computer (slyly) won't tell us which one of non-existence or privateness it is, but we'll assume the first case here: that the function is undefined.

Roberto asks: "Hey, what's that `/1` after the function?"

Glad you asked! It's indicating the *arity* of the function. Elixir functions can not only differ by *name* but also by the number of arguments that the function takes. This error message is telling us that we tried to call the `List.reverse/1` function, and if we passed two arguments instead of one it would tell us that we tried to call the `List.reverse/2` function instead.

```
iex> List.reverse(animals_or_derivatives_of_animals, [1, 2, 3])
List.reverse(animals_or_derivatives_of_animals, [1, 2, 3])
** (UndefinedFunctionError) function List.reverse/2 is undefined or private
    (elixir) List.reverse(["cat", "dog", "cow", "turducken"], [1, 2, 3])
```

A good example of this is the `String.split` function that we saw earlier. It has two variants, one that takes a single argument (`String.split/1`) and one that takes two arguments (`String.split/2`). We only saw the single-argument version before:

```
iex> String.split("Hello, World!")
["Hello,", "World!"]
```

However, if we supply a second argument to this function it behaves differently:

```
iex> String.split("Hello, World!", "o")
["Hell", ", W", "rld!"]
```

The second argument to `String.split/2` tells the computer where to split the string. With the single argument version, the computer assumed we wanted to split on the spaces between the words. With the two argument version -- that is, the `/2` version -- we tell it that we want to split on the `"o"` character instead.

At this point it's also worth mentioning that the `String.reverse` function we called earlier has a proper name of `String.reverse/1`. Just like `/2` indicates the function takes two arguments, the `/1` here indicates that this function takes one argument.

This is an important distinction to make in Elixir, and so I'll say it again, this time a bit bigger:

Functions not only differ by name, but by the number of arguments that they take.

Ok, so we've talked about what `/1` means (Roberto is now deep in thought), so let's talk about why `List.reverse/1` doesn't exist.

Thank you Reader, but our ~~princess~~ function is in another ~~castle~~ module

Another brief interlude, this time about enumerables

Lists are a type of data in Elixir called an *enumerable*. Maps are also *enumerable*. This means that they can be *enumerated* through, and you can do something which each item in the list or map. For instance, if we were to write out our list on a piece of paper, it might look something like this:

- Cat
- Dog
- Cow
- Turducken

It's possible to write each item from the list separately from the other items in the list. We could try to do the same thing for a number (like 1,354), but it wouldn't make sense:

- 1
- 3
- 5
- 4

Numbers are not enumerable because it doesn't make sense for them to be, unlike our list. We can go through each of this list of animals one-at-a-time if we wish. Similar to this, we could *enumerate* through a map. If we were to take one of our maps from earlier...

```
%{"name" => "Roberto", "age" => "30ish", "gender" => "Male"}
```

...and write each key and value pair down, they may look something like this:

Name
: Roberto

Age
: 30ish

Gender
: Male

Again, it makes sense for a map to be an enumerable because you can *enumerate* through each of the key and value pairs in the map.

Ok, but where is `List.reverse/1`?

We need to keep in mind that when we're working with *enumerable* things in Elixir the function that we need might live elsewhere. Sometimes, we need to look in the `Enum` module / drawer too. `Enum` is short for `Enumerable`, and it's done that way because nobody has the time to write out `Enumerable` correctly every single time. Even as this book's author I have a hard time spelling it correctly each time!

We tried looking in the `List` module to find the `reverse/1` function so that we could turn our list around but it wasn't there. So let's look in the `Enum` module instead. Before that, let's get our list in Elixir form again. It's been a while since we've seen it that way:

```
iex> animals_or_derivatives_of_animals = ["cat", "dog", "cow", "turducken"]
```

Since we now know that lists are *enumerables*, and that the `List.reverse/1` function doesn't exist *but* we also (now) know that there's an `Enum` module to work with this sort of thing, we can probably guess that there's going to be an `Enum.reverse/1` function. Let's try it and see:

```
iex> Enum.reverse(animals_or_derivatives_of_animals)
  ["turducken", "cow", "dog", "cat"]
```

Hooray! We were able to reverse our list.

That function *might* be in the Enum module instead

(I thought I would put this in an aside so it's more visible. I know it's not a traditional aside, but it's an important concept and needs highlighting. So with the actual aside-ing out of the way:)

It's important to keep in mind that when we're working with enumerable objects such as lists and maps that we might not be able to find the function in the `List` or `Map` modules because the function might be inside the `Enum` module instead. If you can't find it where you think it might be, make sure you remember to look at `Enum` too!

Roberto emerges from his deep thought trance to ask another question: "Hey, you mentioned before you could *enumerate* through a list or a map, but you didn't show an example of that. What gives?" You're absolutely right, Roberto. I was too distracted with explaining why `List.reverse/1` didn't exist to explain how to enumerate through an enumerable. Let's all now take a look at how to do that before we move onto other functions. We've talked about enums briefly and it would be a shame to stop so early.

Enumerating the enumerables

To appease Roberto (and the masses that he leads), we're going to need to look at how to enumerate through enumerables. What this means is that we're going to get an enumerable (a list or a map) and we're going to go through each of the enumerable's items and do something with them.

Let's start with a list. How about a list of the seasons, just for something different?

```
iex> seasons = ["summer", "fall", "winter", "spring"]
```

The simplest way of enumerating through a list in Elixir is to use the `Enum.each` function. This function takes two arguments: the thing we want to enumerate through and a function to run for each of the items. Let's take a look at an example:

```
iex> Enum.each(seasons, &IO.puts/1)
```

The first argument here is our list of seasons. The second argument is the function that we want to run for each element in this list: specifically the built-in `IO.puts/1` function.

Finish explaining Enum.each

Map, but not the kind that you know already

The names of these seasons should be capitalized because they are nouns, but whoever created this list neglected to capitalize them. Oops! What we should have is a list with proper capitalization:

```
["Summer", "Fall", "Winter", "Spring"]
```

What we want to do in this case is to enumerate through each item in the list and to capitalize the first letter of each word, leaving all the others as lower case. There's a function to do this called `String.capitalize/1`, and for a single string we can call it like this:

```
iex> String.capitalize("summer")
  "Summer"
```

"But how do we run that function on each item in a list?", Roberto asks. He then cries: "I don't want to have to rewrite the whole list!" Roberto is being a little dramatic — given it is a small list of four items — but I can see his point. We *could* rewrite the list to make it what we want it to be, but we're *computer programmers*, so let's program the computer to do our bidding instead.

We're going to need a function that goes through each item in the list and applies `String.capitalize/1`; something that would turn our existing list into a list like this one:

```
iex> seasons = [
    String.capitalize("summer"),
    String.capitalize("fall"),
    String.capitalize("winter"),
    String.capitalize("spring"),
  ]
  ["Summer", "Fall", "Winter", "Spring"]
```

But since we've already got a list we're going to have to do *something* to that list to turn it into what we want. That something is to use another function from the `Enum` module called `Enum.map/2`. This `map` function is different from the map kind of data (`%{"name" => "Roberto"}`), in that it will take the specified enumerable and function, enumerate through each item in the list and run that function for each item, and then return the result.

Enough jibber-jabber. Let's see this in actual practice:

```
iex> Enum.map(seasons, &String.capitalize/1)
["Summer", "Fall", "Winter", "Spring"]
```

Hooray! Each of our seasons now has a correctly capitalized name. What's happened here is that we've told the `map` function that we want it to run another function -- that'd be the `String.capitalize/1` function -- on each item of the `seasons` list. This is how we've been able to go from our existing list with non-capitalized season names to a new list with capitalized names.

Working with maps

Now that we've looked at enumerating through lists, let's look at enumerating through maps. We spoke a little while ago about doing so:

> Similar to this, we could *enumerate* through a map. If we were to take one of our maps from earlier...
>
> ```
> %{"name" => "Roberto", "age" => "30ish", "gender" => "Male"}
> ```
>
> ...and write each key and value pair down, they may look something like this:
>
> **Name**
> Roberto
> **Age**
> 30ish
> **Gender**
> Male

To enumerate through each item in the map we can use the same function as when we were enumerating through each item in a list: `Enum.each/2`. Here's an example:

```
iex> person = %{"name" => "Roberto", "age" => "30ish", "gender" => "Male"}
%{"name" => "Roberto", "age" => "30ish", "gender" => "Male"}
iex> Enum.each(person, fn ({k, v}) -> IO.puts v end)
```

Before I can go any further, Roberto interrupts. "Hey, those curly braces are weird! It looks like a map but it's not a map. And it's not a list because the curliness of the brackets. So what is it?"

* Explain enumerating through maps here, showing that kv pairs are returned as tuples. * Explain what tuples are and how they differ from lists. (iirc, tuples cannot be appended to or iterated (or can they?))

Where to find more functions

TODO: Link to https://hexdocs.pm/elixir/

Chapter is currently unfinished... please hold.

Exercises

Wherein the buzzwords are explained

Explain the following: Elixir is a functional, concurrent, general-purpose programming language that runs on the Erlang virtual machine (BEAM). Particiularly: Functional, concurrent, Erlang, virtual machine, BEAM