Programming Elixir ≥ 1.6

Functional |> Concurrent |> Pragmatic |> Fun

Dave Thomas

The Pragmatic Bookshelf

Raleigh, North Carolina

A Vain Attempt at a Justification, Take Two

I'm a language nut. I love trying languages out, and I love thinking about their design and implementation. (I know; it's sad.)

I came across Ruby in 1998 because I was an avid reader of comp.lang.misc (ask your parents). I downloaded it, compiled it, and fell in love. As with any time you fall in love, it's difficult to explain why. It just worked the way I work, and it had enough depth to keep me interested.

Fast-forward 15 years. All that time I'd been looking for something new that gave me the same feeling.

I came across Elixir a while back, but for some reason never got sucked in. But a few months before starting this book, I was chatting with Corey Haines. I was bemoaning the fact that I wanted a way to show people functional programming concepts without the academic trappings those books seem to attract. He told me to look again at Elixir. I did, and I felt the same way I felt when I first saw Ruby.

So now I'm dangerous. I want other people to see just how great this is. I want to evangelize. So my first step is to write a book.

But I don't want to write another 900-page Pickaxe book. I want this book to be short and exciting. So I'm not going into all the detail, listing all the syntax, all the library functions, all the OTP options, or....

Instead, I want to give you an idea of the power and beauty of this programming model. I want to inspire you to get involved, and then point to the online resources that will fill in the gaps.

But mostly, I want you to have fun.

Fast forward three years. Elixir has moved on. Phoenix, its connectivity framework, introduced a whole new set of developers to the joys of a functional approach. The Nerves project makes it easy to write embedded Elixir code on

• Click HERE to purchase this book now. discuss

Linux-based microcontrollers. The Elixir base has grown—there are international, national, and regional conferences. Job ads ask for Elixir developers.

I've been moving on, too. But I'm still using Elixir daily. I just completed by second year as an adjust professor at Southern Methodist University, corrupting the programmers of tomorrow with the temptations of Elixir. I've written an https://codestool.coding-gnome.com[1].

And now I'm revving this book. To be honest, I don't really have to: Elixir 1.6 is not so different from 1.3 that the older book would not be useful. But my own thinking about Elixir has matured. I now do some things differently. And I'd like to share these things with you.

Acknowledgments

It seems to be a common thread—the languages I fall in love with are created by people who are both clever and extremely nice. José Valim, the creator of Elixir, takes both of these adjectives to a new level. I owe him a massive thank-you for giving me so much fun over the last 18 months. Along with him, the whole Elixir core team has done an amazing job of cranking out an entire ecosystem that feels way more mature than its years. Thank you, all.

A conversation with Corey Haines reignited my interest in Elixir—thank you, Corey, for good evenings, some interesting times in Bangalore, and the inspiration.

Bruce Tate is always an interesting sounding board, and his comments on early drafts of the book made a big difference. And I've been blessed with an incredible number of active and insightful beta readers who have made literally hundreds of suggestions for improvements. Thank you, all.

A big tip of the hat to Jessica Kerr, Anthony Eden, and Chad Fowler for letting me steal their tweets.

Kim Shrier seems to have been involved with my writing since before I started writing. Thanks, Kim, for another set of perceptive and detailed critiques.

The crew at Potomac did their customary stellar job of indexing.

Dave Thomas

dave@pragdave.me

Dallas, TX, August 2016

1. online Elixir course

CHAPTER 1

Take the Red Pill

The Elixir programming language wraps functional programming with immutable state and an actor-based approach to concurrency in a tidy, modern syntax. And it runs on the industrial-strength, high-performance, distributed Erlang VM. But what does all that mean?

It means you can stop worrying about many of the difficult things that currently consume your time. You no longer have to think too hard about protecting your data consistency in a multithreaded environment. You worry less about scaling your applications. And, most importantly, you can think about programming in a different way.

Programming Should Be About Transforming Data

If you come from an object-oriented world, then you are used to thinking in terms of classes and their instances. A class defines behavior, and objects hold state. Developers spend time coming up with intricate hierarchies of classes that try to model their problem, much as Victorian scientists created taxonomies of butterflies.

When we code with objects, we're thinking about state. Much of our time is spent calling methods in objects and passing them other objects. Based on these calls, objects update their own state, and possibly the state of other objects. In this world, the class is king—it defines what each instance can do, and it implicitly controls the state of the data its instances hold. Our goal is data-hiding.

But that's not the real world. In the real world, we don't want to model abstract hierarchies (because in reality there aren't that many true hierarchies). We want to get things done, not maintain state.

• Click HERE to purchase this book now. discuss

Right now, for instance, I'm taking empty computer files and transforming them into files containing text. Soon I'll transform those files into a format you can read. A web server somewhere will transform your request to download the book into an HTTP response containing the content.

I don't want to hide data. I want to transform it.

Combine Transformations with Pipelines

Unix users are used to the philosophy of small, focused command-line tools that can be combined in arbitrary ways. Each tool takes an input, transforms it, and writes the result in a format that the next tool (or a human) can use.

This philosophy is incredibly flexible and leads to fantastic reuse. The Unix utilities can be combined in ways undreamed of by their authors. And each one multiplies the potential of the others.

It's also highly reliable—each small program does one thing well, which makes it easier to test.

There's another benefit. A command pipeline can operate in parallel. If I write

```
$ grep Elixir *.pml | wc -l
```

the word-count program, `wc`, runs at the same time as the `grep` command. Because `wc` consumes `grep`'s output as it is produced, the answer is ready with virtually no delay once `grep` finishes.

Just to give you a taste of this kind of thing, here's an Elixir function called `pmap`. It takes a collection and a function, and returns the list that results from applying that function to each element of the collection. But...it runs a separate process to do the conversion of each element. Don't worry about the details for now.

spawn/pmap1.exs

```
defmodule Parallel do
  def pmap(collection, func) do
    collection
    |> Enum.map(&(Task.async(fn -> func.(&1) end)))
    |> Enum.map(&Task.await/1)
  end
end
```

We could run this function to get the squares of the numbers from 1 to 1000.

```
result = Parallel.pmap 1..1000, &(&1 * &1)
```

And, yes, I just kicked off 1,000 background processes, and I used all the cores and processors on my machine.

The code may not make much sense, but by about halfway through the book, you'll be writing this kind of thing for yourself.

Functions Are Data Transformers

Elixir lets us solve the problem in the same way the Unix shell does. Rather than have command-line utilities, we have functions. And we can string them together as we please. The smaller—more focused—those functions, the more flexibility we have when combining them.

If we want, we can make these functions run in parallel—Elixir has a simple but powerful mechanism for passing messages between them. And these are not your father's boring old processes or threads—we're talking about the potential to run millions of them on a single machine and have hundreds of these machines interoperating. Bruce Tate commented on this paragraph with this thought: "Most programmers treat threads and processes as a necessary evil; Elixir developers feel they are an important simplification." As we get deeper into the book, you'll start to see what he means.

This idea of transformation lies at the heart of functional programming: a function transforms its inputs into its output. The trigonometric function *sin* is an example—give it $\pi/4$, and you'll get back 0.7071…. An HTML templating system is a function; it takes a template containing placeholders and a list of named values, and produces a completed HTML document.

But this power comes at a price. You're going to have to unlearn a whole lot of what you *know* about programming. Many of your instincts will be wrong. And this will be frustrating, because you're going to feel like a total n00b.

Personally, I feel that's part of the fun.

You didn't learn, say, object-oriented programming overnight. You are unlikely to become a functional programming expert by breakfast, either.

But at some point things will click. You'll start thinking about problems in a different way, and you'll find yourself writing code that does amazing things with very little effort on your part. You'll find yourself writing small chunks of code that can be used over and over, often in unexpected ways (just as `wc` and `grep` can be).

Your view of the world may even change a little as you stop thinking in terms of responsibilities and start thinking in terms of getting things done.

And just about everyone can agree that will be fun.

• Click HERE to purchase this book now. discuss

Installing Elixir

This book assumes you're using at least Elixir 1.6. The most up-to-date instructions for installing Elixir are available at http://elixir-lang.org/install.html. Go install it now.

Running Elixir

In this book, I show a terminal session like this:

```
$ echo Hello, World
Hello, World
```

The terminal prompt is the dollar sign, and the stuff you type follows. (On your system, the prompt will likely be different.) Output from the system is shown without highlighting.

iex—Interactive Elixir

To test that your Elixir installation was successful, let's start an interactive Elixir session. At your regular shell prompt, type iex.

```
$ iex
Erlang/OTP 20 [erts-9.1] [source] [64-bit] [smp:4:4] [ds:4:4:10]
              [async-threads:10] [hipe] [kernel-poll:false]h
Interactive Elixir (x.y.z) - press Ctrl+C to exit (type h() ENTER for h
elp)
iex(1)>
```

(The various version numbers you see will likely be different—I won't bother to show them on subsequent examples.)

Once you have an iex prompt, you can enter Elixir code and you'll see the result. If you enter an expression that continues over more than one line, iex will prompt for the additional lines with an ellipsis (...).

```
iex(1)> 3 + 4
7
iex(2)> String.reverse "madamimadam"
"madamimadam"
iex(3)> 5 *
...(3)> 6
30
iex(4)>
```

The number in the prompt increments for each complete expression executed. I'll omit the number in most of the examples that follow.

There are several ways of exiting from iex—none are tidy. The easiest two are typing Ctrl-C twice or typing Ctrl-G followed by q and Return. On some systems, you can also use a single Ctrl-\.

IEx Helpers

iex has a number of helper functions. Type h (followed by return) to get a list:

```
iex> h
                                IEx.Helpers

Welcome to Interactive Elixir. You are currently seeing the documentation
for the module IEx.Helpers which provides many helpers to make Elixir's
shell more joyful to work with.

This message was triggered by invoking the helper h(), usually referred to
as h/0 (since it expects 0 arguments).

You can use the h/1 function to invoke the documentation for any Elixir
module or function:

    iex> h(Enum)
    iex> h(Enum.map)
    iex> h(Enum.reverse/1)

You can also use the i/1 function to introspect any value you have in the
shell:

    iex> i("hello")

There are many other helpers available, here are some examples:

• b/1            - prints callbacks info and docs for a given module
• c/1            - compiles a file into the current directory
• c/2            - compiles a file to the given path
• cd/1           - changes the current directory
• clear/0        - clears the screen
• exports/1      - shows all exports (functions + macros) in a module
• flush/0        - flushes all messages sent to the shell
• h/0            - prints this help message
• h/1            - prints help for the given module, function or macro
• i/0            - prints information about the last value
• i/1            - prints information about the given term
• ls/0           - lists the contents of the current directory
• ls/1           - lists the contents of the specified directory
• open/1         - opens the source for the given module or function
                   in your editor
• pid/1          - creates a PID from a string
• pid/3          - creates a PID with the 3 integer arguments passed
• ref/1          - creates a Reference from a string
• ref/4          - creates a Reference with the 4 integer arguments
                   passed
• pwd/0          - prints the current working directory
• r/1            - recompiles the given module's source file
```

```
• recompile/0    - recompiles the current project
• runtime_info/0 - prints runtime info (versions, memory usage, stats)
• v/0            - retrieves the last value from the history
• v/1            - retrieves the nth value from the history

Help for all of those functions can be consulted directly from the command
line using the h/1 helper itself. Try:

    iex> h(v/0)

To list all IEx helpers available, which is effectively all exports
(functions and macros) in the IEx.Helpers module:

    iex> exports(IEx.Helpers)

This module also includes helpers for debugging purposes, see IEx.break!/4
for more information.

To learn more about IEx as a whole, type h(IEx).
```

In the list of helper functions, the number following the slash is the number of arguments the helper expects.

Probably the most useful is `h` itself. With an argument, it gives you help on Elixir modules or individual functions in a module. This works for any modules loaded into iex (so when we talk about projects later on, you'll see your own documentation here, too). For example, the `IO` module performs common I/O functions. For help on the module, type `h(IO)` or `h IO`.

```
iex> h IO     # or...
iex> h(IO)

Functions handling IO.

Many functions in this module expects an IO device as argument. An IO device
must be a PID or an atom representing a process. For convenience, Elixir
provides :stdio and :stderr as shortcuts to Erlang's :standard_io and
:standard_error....
```

This book frequently uses the `puts` function in the `IO` module, which in its simplest form writes a string to the console. Let's get the documentation.

```
iex> h IO.puts
    def puts(device \\ :stdio, item)

Writes item to the given device, similar to write/2, but adds a
newline at the end.

By default, the device is the standard output. It returns :ok if it
succeeds.

## Examples

    IO.puts "Hello World!"
    #=> Hello World!
```

```
    IO.puts :stderr, "error"
    #=> error
```

Another informative helper is i, which displays information about a value:

```
iex> i 123
Term
  123
Data type
  Integer
Reference modules
  Integer

iex> i "cat"
Term
  "cat"
Data type
  BitString
Byte size
  3
Description
  This is a string: a UTF-8 encoded binary. It's printed surrounded by
  "double quotes" because all UTF-8 codepoints in it are printable.
Raw representation
  <<99, 97, 116>>
Reference modules
  String, :binary

iex> i %{ name: "Dave", likes: "Elixir" }
Term
  %{likes: "Elixir", name: "Dave"}
Data type
  Map
Reference modules
  Map

iex> i Map
Term
  Map
Data type
  Atom
Module bytecode
  /Users/dave/Play/elixir/bin/../lib/elixir/ebin/Elixir.Map.beam
Source
  /Users/dave/Play/elixir/lib/elixir/lib/map.ex
Version
  [136119987195443140315307232506105292657]
Compile time
  2015-12-29 16:33:20
Compile options
  [:debug_info]
```

```
Description
  Use h(Map) to access its documentation.
  Call Map.module_info() to access metadata.
Raw representation
  :"Elixir.Map"
Reference modules
  Module, Atom
```

iex is a surprisingly powerful tool. Use it to compile and execute entire projects, log in to remote machines, and access running Elixir applications.

And, if you happen to include the occasional bug in your code (deliberately, of course), iex has a simple debugger. We'll talk about it when we look at tooling on page ?.

Customizing iex

You can customize iex by setting options. For example, I like showing the results of evaluations in bright cyan. To find out how to do that, I used this:

```
iex> h IEx.configure
def configure(options)

Configures IEx.

The supported options are: :colors, :inspect, :default_prompt,
:alive_prompt and :history_size.

Colors

A keyword list that encapsulates all color settings used by the shell. See
documentation for the IO.ANSI module for the list of supported colors and
attributes.

The value is a keyword list. List of supported keys:

• :enabled      - boolean value that allows for switching the coloring
                  on and off
• :eval_result  - color for an expression's resulting value
• :eval_info    - … various informational messages
• :eval_error   - … error messages
• :stack_app    - … the app in stack traces
• :stack_info   - … the remaining info in stack traces
• :ls_directory - … for directory entries (ls helper)
• :ls_device    - … device entries (ls helper)

 . . .
```

I then created a file called .iex.exs in my home directory, containing

```
IEx.configure colors: [ eval_result: [ :cyan, :bright ] ]
```

If your iex session looks messed up (and things such as [33m appear in the output), it's likely your console does not support ANSI escape sequences. In that case, disable colorization using

```
IEx.configure colors: [enabled: false]
```

You can put any Elixir code into .iex.exs.

Compile and Run

Once you tire of writing one-line programs in iex, you'll want to start putting code into source files. These files will typically have the extension .ex or .exs. This is a convention—files ending in .ex are intended to be compiled into bytecodes and then run, whereas those ending in .exs are more like programs in scripting languages—they are effectively interpreted at the source level. When we come to write tests for our Elixir programs, you'll see that the application files have .ex extensions, whereas the tests have .exs because we don't need to keep compiled versions of the tests lying around.

Let's write the classic first program. Go to a working directory and create a file called hello.exs.

intro/hello.exs

```
IO.puts "Hello, World!"
```

The previous example shows how most of the code listings in this book are presented. The bar before the code itself shows the path and file name that contains the code. If you're reading an ebook, you'll be able to click on this to download the source file. You can also download all the code by visiting the book's page on our site and clicking on the *Source Code* link.[1]

Source file names are written in lowercase with underscores. They will have the extension .ex for programs that you intend to compile into binary form, and .exs for scripts that you want to run without compiling. Our "Hello, World" example is essentially throw-away code, so we used the .exs extension for it.

1. http://pragprog.com/titles/elixir16

Having created our source file, let's run it. In the same directory where you created the file, run the elixir command:

```
$ elixir hello.exs
Hello, World!
```

We can also compile and run it inside iex using the c helper:

```
$ iex
iex> c "hello.exs"
Hello, World!
[]
iex>
```

The c helper compiled and executed the source file. The [] that follows the output is the return value of the c function—if the source file had contained any modules, their names would have been listed here.

The c helper compiled the source file as freestanding code. You can also load a file as if you'd typed each line into iex using import_file. In this case, local variables set in the file are available in the iex session.

As some folks fret over such things, the Elixir convention is to use two-column indentation and spaces (not tabs).

Suggestions for Reading the Book

This book is not a top-to-bottom reference guide to Elixir. Instead, it is intended to give you enough information to know what questions to ask and when to ask them. So approach what follows with a spirit of adventure. Try the code as you read, and don't stop there. Ask yourself questions and then try to answer them, either by coding or searching the web.

Participate in the book's discussion forums and consider joining the Elixir mailing list.[2,3]

You're joining the Elixir community while it is still young. Things are exciting and dynamic, and there are plenty of opportunities to contribute.

Exercises

You'll find exercises sprinkled throughout the book. If you're reading an ebook, then each exercise will link directly to a topic in our online forums. There

2. http://forums.pragprog.com/forums/elixir13
3. https://groups.google.com/forum/?fromgroups#!forum/elixir-lang-talk

you'll find an initial answer, along with discussions of alternatives from readers of the book.

If you're reading this book on paper, visit the forums to see the list of exercise topics.[4]

Think Different(ly)

This is a book about thinking differently—about accepting that some of the things folks say about programming may not be the full story:

- Object orientation is not the only way to design code.
- Functional programming need not be complex or mathematical.
- The bases of programming are not assignments, `if` statements, and loops.
- Concurrency does not need locks, semaphores, monitors, and the like.
- Processes are not necessarily expensive resources.
- Metaprogramming is not just something tacked onto a language.
- Even if it is work, programming should be fun.

Of course, I'm not saying Elixir is a magic potion (well, technically it is, but you know what I mean). It isn't the *one true way* to write code. But it's different enough from the mainstream that learning it will give you more perspective and it will open your mind to new ways of thinking about programming.

So let's start.

And remember to make it fun.

4. http://forums.pragprog.com/forums/322