The Pragmatic Programmers

Programming Elixir

Functional
|> Concurrent
|> Pragmatic
|> Fun

Dave Thomas

Foreword by
José Valim,
Creator of Elixir

edited by Lynn Beighley

Prepared exclusively for Richard Bishop

Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at *pragprog.com/my_account*. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at *pragprog.com/titles/elixir/errata*, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

Dave & Andy

Prepared exclusively for Richard Bishop

Programming Elixir

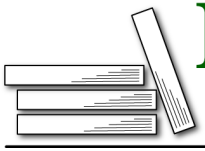
Functional |> Concurrent |> Pragmatic |> Fun

Dave Thomas

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Prepared exclusively for Richard Bishop



Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2014 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-937785-58-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: B14.3a—July 15, 2014

Prepared exclusively for Richard Bishop

Contents

Prepared exclusively for Richard Bishop

Prepared exclusively for Richard Bishop

Prepared exclusively for Richard Bishop

Part II — Concurrent Programming

Prepared exclusively for Richard Bishop

Part III — More Advanced Elixir

Prepared exclusively for Richard Bishop

Prepared exclusively for Richard Bishop

Introduction to the Beta

First, I want to say "thank you!"

If you're reading this, it means you are interested in exploring new ideas, and finding better ways to do things. I genuinely salute you for that—it is sadly rare in our industry. I think Elixir will repay you.

I also want to thank you for reading the beta version of this book. We aren't too far off finishing, but there still remain a lot of rough edges and missing content. I also need to provide links to help on each of the exercises.

If you come across something that's wrong, or that doesn't make sense, I'd really appreciate it if you could spend a minute and let me know by posting to the book's errata. If you are reading this in PDF form, there's a link at the bottom of each page. Otherwise, you can get to the errata at http://pragprog.com/titles/elixir/errata.

Enjoy!

Dave Thomas

dave@pragprog.com

Dallas, TX, May 2013

Prepared exclusively for Richard Bishop

Changes in the Betas

B14.3a — July 15, 2014

- I jumped the gun with the previous release number. It was for Elixir 0.14.2. This release is for 0.14.3
- mix.escriptize is now mix escript.build
- Options for bitfields are now separated by hyphens, rather than being comma separated in brackets.
- If you have multiple heads for a function that also has default parameters, you must now list those defaults in an empty head preceding the normal definitions.
- The parameter to exit has been codified, and I've changed code accordingly.
- Functions can be used to make get_in and get_and_update_in more dynamic.
- Fixed known errata.

B14.3.14.3 — July 1, 2014

- Gave the *Organizing A Project* chapter some TLC.
  - Updated to the new mix and project conventions.
  - Switched back to using the cannonical httpotion.
  - Added a section illustrating the use of project configuration.
- Kernel.size has been removed. Use byte_size and tuple_size.
- Switched back to using the myfreeweb version of httpotion now that it works with 0.14.
- Added back 4 exercises that got lost in a content refactoring a while back.
- Fixed pending errata.

B14.0.14.0 — June 18, 2014

- get_in/2 has been deprecated

Prepared exclusively for Richard Bishop

B14.0.14.0 — June 17, 2014

- Documented nested structure accessors (get_in, put_in etc)
- Described the ability to @derive Access for structs
- Removed defrecord
- Removed the ability to have heredocs where there was additional code on the line that introduced the string.
- Fix pending errata.

B13.3.13.3 — May 27, 2014

- Updated the OTP chapters to use the new built-in OTP support (was GenX).
- Added a new chapter on Tasks and Agents.
- IEx configuration now done using IEx.configure.
- Fix pending errata.

B13.0.13.0 — April 21, 2014

- Updated for Elixir 0.13. This is a big change, as it adds a new basic data type, the Map, and a new derived type, the struct. It also downplays records. Protocols are now defined on structs.
- Reorganized the chapters on lists and enumerables:
  - ListDict is deprecated in favor of Map.
  - The various Dict implementations are now created using Enum.into, instead of calling new with a collection.
  - Updated for the new list comprehension syntax (for).
- GenFSM is deprecated.
- is_range and is_regex are now Range.range? and Regex.regex?.
- Changed the Issues project to use jsx rather than Jsonex.
- Pending errata fixed

B12.0.12.3 — February 7, 2014

- Default parameter values are now denoted using \\ and not //.
- Sigils are now introduced using ~ and not %. In addition, the set of characters that can be used as delimiters has been reduced.
- iex now reads from .iex.exs and not .iex on startup.

Prepared exclusively for Richard Bishop

B12.0.12.2 — January 16, 2014

- The pid <- msg notation is deprecated in favor of send(pid, msg).
- Rewrote the section on Protocols and Records to reflect changes in 0.12 and 0.12.2.
- Enum.first and the raw option to inspect are deprecated.
- Fix errata (more big thanks to Bernard Kaiflin).

B12.0.12.0 — December 16, 2013

- Skipped a beta number to get in step with Elixir numbering.
- Moved to Elixir 0.12.0. This release of Elixir reimplements the internals of streaming, but the change should not affect you unless you've written your own streaming types.
- Update the section on streams to document Stream.unfold and Stream.resource.
- Changes to some sample code: Enum.chunks is now Enum.chunk; Enum.zip now stops when the shortest source is exhausted.
- Fixed errata (thank you Bernard Kaiflin for a stunning list of suggestions).

B10.0.11.2 — November 13, 2013

- Moved to Elixir 0.11.2
- Update the OTP Application and the Type Specifications chapters to use the the _build directory tree (where Elixir now stores all build products)
- Use ExUnit.CaptureIO in the Project chapter to test the table formatter.
- Document the new escript_main_module option. In future, this will be required whenever mix escriptize is used.

B9.0.11.0 — November 6, 2013

- Moved to Elixir 0.11.1
- Fixed errata
- Many changes to standard library. In particular, what were once binaries are now (largely) strings.
- Changes to the defaults of a new project (the addition of a default supervisor and an application entry point). As a result, the supervisor chapter is changed slightly, and the start of the OTP Applications chapter is changed extensively.
- Removed @only and @except annotations from protocols, and document @fallback_to_any.

Prepared exclusively for Richard Bishop

B8.0.10.3 — October 2, 2013

- Updated to Elixir 0.10.3
- Moved the concat function from List to Enum.
- Fixed known errata.

B7.0.10.2 — September 6, 2013

- Updated to Elixir 0.10.2
- Fix ExUnit output to show new interleaved failures
- Update example in Project chapter to handle extra element in tuple returned by OptionParser.parse.
- Update Enum examples to use min_by (rather than min/2)
- Change IO.stream(dev) to IO.stream(dev, :line)
- to_binary is now to_chars, and the Binary.Chars protocol is now String.Chars.
- Update import documentation to remove first optional parameter. Now pass :functions or :macros to only:.
- list_to_binary is now String.from_char_list!
- Enum.reduce now has a two-parameter form that takes the first element of the collection as the initial accumulator.
- Show examples of Enum.chunks and Enum.is_even.
- Document String.ljust and rjust.
- function/3 has been removed, and is replaced by Module.function\3.
- Errata fixes.

B6.0.10.1 - August 3, 2013

- Change to use the new & function capture syntax.
- Added a short chapter on gen_fsm.
- A few new library functions, including String.reverse and Enum.shuffle.
- Fix errata

B5.0.10.0 - July 15, 2013

- Covers Elixir 0.10.0
- Split the description of Enum out of the Lists chapter, and added the new Stream module for lazy enumerations.
- Added Enum.reject and Enum.with_index
- Binary.Inspect is now Inspect, and uses algebra documents (impress your friends).

Prepared exclusively for Richard Bishop

- Code updates to reflect the new stricter syntax for parentheses on function calls.
- Updated iex output to show new pretty-printed results.
- Discuss the optimization on expressions such as &1+&2 (now maps to Kernel.+/2).
- Removed the List and Enum API descriptions, and substituted some examples of the API in use
- Fixed errata
- Initial set of "Your Turn" solutions now online. This covers roughly the first half of the book. The rest will be added over the coming weeks.

B4.0.9.3 - June 24, 2013

- Tidied the FizzBuzz example and the table_formatter code in the Projects chapter
- Talk about the new binding: option to quote
- Can now set attribute types directly in defrecord
- Covers Elixir 0.9.3
  - Sigils now dispatch to sigil_x (instead of __x__)
  - Code.string_to_ast/1 and Macro.to_binary/1 have been deprecated in favor of Code.string_to_quoted/1 and Macro.to_string/1

B3.0 - June 14, 2013

- Fixed errata
- Added information of how to specify modifiers on heredoc sigils
- New section on implementing your own sigils
- New section on umbrella projects
- New appendix on type specifications and type checking
- Covers Elixir 0.9.2
  - Added Enum.fetch and fetch!, removed Enum.at! and Enum.equal?
  - Added String.contains?

B2.0 - May 27, 2013

- New chapters on Protocols, Macros, and Metaprogramming
- Note new elixir-lang-talk mailing list
- Covers Elixir 0.9 (previous version was 0.8).
  - This now requires Erlang R16 to be installed
  - Change of Elixir module naming from Elixir-Xxx to Elixir.Xxx

Prepared exclusively for Richard Bishop

- Add String.starts_with? and .ends_with?
- IEx customization with IEx.Options
- Replacement of Enum.Iterator with Enumerator

Prepared exclusively for Richard Bishop

Foreword

I have always been fascinated with how changes in hardware affect how we write software.

A couple decades ago, memory was a very limited resource. It made sense back then for our software to take a hold of some piece of memory and mutate it as necessary. However, allocating this memory and cleaning up after we no longer needed it was a very error prone task. Some memory was never freed; sometimes memory was allocated over another structure, leading to faults. At the time, Garbage Collection was a known technique, but we needed faster CPUs in order to use it in our daily software and free ourselves from manual memory management. That has happened—most of our languages are now garbage collected.

Today, a similar phenomena is happening. Our CPUs are not getting any faster. Instead, our computers get more and more cores. This means that new software needs to use as many cores as it can if it is to maximum its use of the machine. This conflicts directly with how we currently write software.

In fact, mutating our memory state actually slows down our software when many cores are involved. If you have four cores trying to access and manipulate the same piece of memory, they can trip over each other. This potentially corrupts memory unless some kind of synchronization is applied.

I quickly learned that applying this synchronization is manual, error prone, hurts performance and tiresome. I suddenly realized that's not how I wanted to spend time writing software in the next years of my career and set out to study new languages and technologies.

It was on this quest that I fell in love with the Erlang Virtual Machine and ecosystem.

In the Erlang VM, all code runs in tiny processes running concurrently, each with its own state. Processes talk to each other via messages. And since all communication happens by message-passing, exchanging messages between

Prepared exclusively for Richard Bishop

different machines on the same network is handled transparently by the VM, making it a perfect environment for building distributed software!

However I felt there was still a gap in the Erlang ecosystem. I missed first-class support for some of the features I find necessary in my daily work, things such as meta-programming, polymorphism, and first-class tooling. From this need, Elixir was born.

Elixir is a pragmatic approach to functional programming. It values its functional foundations and it focuses on developer productivity. Concurrency is the backbone of Elixir software. As garbage collection once freed developers from the shackles of memory management, Elixir is here to free you from antiquated concurrency mechanisms and bring you joy when writing concurrent code.

By being a functional programming language, we now think in terms of functions that transform data. This transformation never mutates data. Instead, each function application creates a new, fresh version of it. This greatly reduces the need for data synchronization mechanisms.

Elixir also empowers developers by providing macros. Elixir code is nothing more than data, and therefore can be manipulated via macros as any other value in the language.

Finally, Object-Oriented programmers are going to find many of the mechanisms they consider essential to write good software, like polymorphism, in Elixir.

All this is powered by the Erlang VM, a 20-year-old virtual machine built from scratch to support robust, concurrent, and distributed software. Elixir and the Erlang VM are going to change how you write software and make you ready to tackle the upcoming years in programming.

José Valim
Creator of Elixir
Tenczynek, Poland, July 2014

Prepared exclusively for Richard Bishop

A Vain Attempt at a Justification

I'm a language nut. I love trying them 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 stuck in. But a few months ago I was chatting with Corey Haines. I was bemoaning the fact that I wanted to find a way to show people functional programming concepts without the kind of 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. I want this book to be short and exciting. So I'm not going in to all the detail, listing all the syntax, all the library functions, or 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.

Dave Thomas

dave@pragprog.com

Dallas, TX, June 2013

Prepared exclusively for Richard Bishop

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 multi-threaded environment. It means that you worry less about scaling your applications. And, most importantly, it means that 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 gentleman scientists created taxonomies of butterflies.

When you code with objects, you're thinking about state. Much of your time is spent calling methods in objects, 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. Your goal is data hiding.

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

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

Prepared exclusively for Richard Bishop

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

Combine Transformations with Pipelines

Unix users are very 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 can be used by the next tool (or by a human).

It 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 pipeline of functions 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.

spawn/pmap.exs

```
defmodule Parallel do
  def pmap(collection, fun) do
    me = self

    collection
  |>
    Enum.map(fn (elem) ->
       spawn_link fn -> (send me, { self, fun.(elem) }) end
     end)
  |>
    Enum.map(fn (pid) ->
      receive do { ^pid, result } -> result end
    end)
  end
```

Prepared exclusively for Richard Bishop

```
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 won't make much sense, but by about half way through the book, you'll be writing this kind of thing for yourself.

Source file naming

Source file names are written in lower case 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.

The PMap example used .exs, as it was essentially throw-away code.

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 having hundreds of these machines interoperating. Bruce Tate commented on this paragraph with "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 some kind of 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.

Prepared exclusively for Richard Bishop

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 of things in terms of responsibilities and start thinking in terms of getting things done.

And just about everyone can agree that will be fun.

Installing Elixir

The most up-to-date instructions for installing Elixir are available online at http://elixir-lang.org/getting_started/1.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 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 17 [erts-6.0] [source] [64-bit] [smp:4:4] [async-threads:10]
              [hipe] [kernel-poll:false]

Interactive Elixir (x.y.z) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

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

Prepared exclusively for Richard Bishop

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(2)> 3 + 7
10
iex(3)> 5 *
...(3)> 8
40
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.

As well as Elixir code, 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 as `h/0` (since it expects 0
arguments).

There are many other helpers available:

* `c/2`       — compiles a file at the given path
* `cd/1`      — changes the current directory
* `clear/0`   — clears the screen
* `flush/0`   — flushes all messages sent to the shell
* `h/0`       — prints this help
* `h/1`       — prints help for the given module, function or macro
* `l/1`       — loads the given module's beam code and purges the current version
* `ls/0`      — lists the contents of the current directory
* `ls/1`      — lists the contents of the specified directory
* `pwd/0`     — prints the current working directory
* `r/0`       — recompile and reload all modules that were previously reloaded
* `r/1`       — recompiles and reloads the given module's source file
* `respawn/0` — respawns the current shell
* `s/1`       — prints spec information
* `t/1`       — prints type information
* `v/0`       — prints the history of commands evaluated in the session
* `v/1`       — retrieves the nth value from the history
* `import_file/1` — evaluates the given file in the shell's context

Help for functions in this module can be consulted
```

Prepared exclusively for Richard Bishop

```
directly from the command line, as an example, try:

    h(c/2)

You can also retrieve the documentation for any module
or function. Try these:

    h(Enum)
    h(Enum.reverse/1)
```

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, later on, when we talk about projects, 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.

The majority of the functions expect data encoded in UTF-8 and will do a
conversion to string, via the String.Chars protocol (as shown in typespecs).

The functions starting with bin* expects iodata as arguments, i.e. iolists or
binaries with no particular encoding.
```

(Don't worry about all the "strings" and "binaries" stuff—there's a whole chapter about it later.)

This book frequently uses the `puts` function in the `IO` module. Let's get the documentation.

```
iex> h IO.puts

* def puts(device \\ group_leader(), item)

Writes the argument to the device, similarly to write
but adds a new line at the end. The argument is expected
to be a chardata.
```

Prepared exclusively for Richard Bishop

There are several ways of exiting from iex—none are tidy. The easiest two are either typing control-C twice, or typing control-G followed by `q` and return.

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

Customizing iex

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

```
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 stacktraces
• :ls_directory - … for directory entries (ls helper)
• :ls_device    - … device entries (ls helper)

This is an aggregate option that encapsulates all color settings used by the
shell. See documentation for the `IO.ANSI` module for the list of supported
colors and attributes.
 . . .
```

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

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

Prepared exclusively for Richard Bishop

If your iex session looks messed up (and there are things such as [33m appearing 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 .ex are intended to be compiled into byte-codes and then run, whereas those ending .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, while the tests have .exs, as 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 filename 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[1] and clicking on the *Source Code* link.)

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"
```

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

Prepared exclusively for Richard Bishop

```
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 free-standing 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 where to ask them. So try to 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[2] and consider joining the Elixir mailing list.[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 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[4] to see the list of exercise topics.

2. http://forums.pragprog.com/forums/elixir
3. https://groups.google.com/forum/?fromgroups#!forum/elixir-lang-talk
4. http://forums.pragprog.com/forums/322

Prepared exclusively for Richard Bishop

Think Different(ly)

This is a book about thinking differently; about accepting that some of the things that 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 foundations of programming are not assignment, 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 that Elixir is a magic potion (well, technically it is, but you know what I mean). There isn't the ONE TRUE WAY to write code. But Elixir is 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.

Acknowledgements

Prepared exclusively for Richard Bishop

Part I

Conventional Programming

Elixir is great for writing highly parallel, reliable applications.

But to be a great language for parallel programming, you first have to be a great language for conventional, sequential programming. In this part of the book we'll learn how to write Elixir code, and see the idioms and conventions that make Elixir so powerful.

Prepared exclusively for Richard Bishop

In this chapter, we'll see
- pattern matching binds values to variables
- matching handles structured data
- _ lets you ignore a match

CHAPTER 2

Pattern Matching

We started the previous chapter by saying that Elixir engenders a different way of thinking about programming.

To illustrate this, and to set the foundation for a lot of Elixir programming, let's start reprogramming your brain by looking at something that's one of the cornerstones of all programming languages—assignment.

Assignment: I do not think it means what you think it means

Let's look at a really simple piece of code:[1]

```
iex> a = 1
1
iex> a + 3
4
```

Most programmers would look at this code and say "OK, we assign one to a variable `a`, then on the next line we add 3 to `a`, giving 4."

But when it comes to Elixir, they'd be wrong. In Elixir, the equals sign is not an assignment. Instead it's like an assertion. It succeeds if Elixir can find a way of making the left hand side equal the right hand side. Elixir calls `=` a *match operator*.

Now in this case, the left hand side is a variable, and the right hand side is an integer literal, so Elixir can make the match true by binding the variable

1. Remember, this is an example of the interactive Elixir shell, iex. You normally start it at a command prompt using the iex command. Once started, it prompts you for code (the iex> in our example), and displays the value.

Prepared exclusively for Richard Bishop

a to value 1. You could argue it *is* just an assignment. But let's take it up a notch.

```
iex> a = 1
1
iex> 1 = a
1
iex> 2 = a
** (MatchError) no match of right hand side value: 1
```

Look at the second line of code, 1 = a. It's another match, and it passes. The variable a already has the value 1 (it was set in the first line), and so what's on the left of the equals sign is the same as what's on the right, and the match succeeds.

But the third line, 2 = a, raises an error. You might have expected it to assign 2 to a, as that would make the match succeed, but Elixir will only change the value of a variable on the left hand side of an equals sign—on the right hand side, a variable is replaced with its value. This failing line of code is the same as 2 = 1, which causes the error.

More Complex Matches

First, a little background syntax. Elixir lists can be created using square brackets containing a comma separated set of values. Here are some lists:

```
[ "Humperdinck", "Buttercup", "Fezzik" ]
[ "milk", "butter", [ "iocane", 12 ] ]
```

Back to the match operator.

```
iex> list = [ 1, 2, 3 ]
[1, 2, 3]
```

To make the match true, Elixir bound the variable list to the list [1, 2, 3].

But let's try something else:

```
iex> list = [1, 2, 3]
[1, 2, 3]
iex> [a, b, c ] = list
[1, 2, 3]
iex> a
1
iex> b
2
iex> c
3
```

Prepared exclusively for Richard Bishop

Elixir looks for a way to make the value of the left hand side the same as the right. The left hand side is a list containing three variables, and the right a list of three values, so the two sides could be made the same by setting the variables to the corresponding values.

Elixir calls this process *pattern matching*. A pattern (the left hand side) is matched if the values (the right hand side) have the same structure, and if each term in the pattern can be matched to the corresponding term in the values. A literal value in the pattern matches that exact value, and a variable in the pattern matches by taking on the corresponding value.

Let's look at a few more examples.

```
iex> list = [1, 2, [ 3, 4, 5 ] ]
[1, 2, [3, 4, 5]]
iex> [a, b, c ] = list
[1, 2, [3, 4, 5]]
iex> a
1
iex> b
2
iex> c
[3, 4, 5]
```

The value on the right hand side corresponding the term `c` on the left hand side is the sublist `[3,4,5]`; that is the value given to `c` to make the match true.

Let's try a pattern containing some values and some variables.

```
iex> list = [1, 2, 3]
[1, 2, 3]
iex> [a, 2, b ] = list
[1, 2, 3]
iex> a
1
iex> b
3
```

Here, the literal 2 in the pattern matched the corresponding term on the right, so the match succeeds by setting the values of `a` and `b` to 1 and 3. But...

```
iex> list = [1, 2, 3]
[1, 2, 3]
iex> [a, 1, b ] = list
** (MatchError) no match of right hand side value: [1, 2, 3]
```

But here the 1 (the second term in the list) cannot be matched against the corresponding element in the right hand side, and so no variables are set and

Prepared exclusively for Richard Bishop

the match fails. This gives us a way of matching a list that meets certain criteria, in this case a length of 3 and with 1 as its second element.

You can even match on ranges (a range represents the start and end of a set of values):

```
iex> a..b = 1..10
1..10
iex> a
1
iex> b
10
```

Your turn…

➤ *Exercise: PatternMatching-1*

Which of the following would match?

- a = [1, 2, 3]
- a = 4
- 4 = a
- [a, b] = [1, 2, 3]
- a = [[1, 2, 3]]
- [a..5] = [1..5]
- [a] = [[1, 2, 3]]
- [[a]] = [[1, 2, 3]]

Ignoring a Value With _ (Underscore)

If we didn't need to capture a value during the match, we could use the special variable _ (an underscore). This acts like a variable, but immediately discards any value given to it—in a pattern match, it is like a wildcard saying "I'll accept any value here." The following example matches any three element list that has a one as its first element.

```
iex> [1, _, _] = [1, 2, 3]
[1, 2, 3]
iex> [1, _, _] = [1, "cat", "dog"]
[1, "cat", "dog"]
```

Variables Bind Once (Per Match)

Once a variable has been bound to a value in the matching process, that's the value it keeps for the remainder of the match.

```
iex> [a, a] = [1, 1]
```

Prepared exclusively for Richard Bishop

```
[1, 1]
iex> a
1
iex> [a, a] = [1, 2]
** (MatchError) no match of right hand side value: [1, 2]
```

The first expression in this example succeeds because `a` is initially matched with the first 1 in the right-hand side (RHS). The value in `a` is then used in the second term to match the second 1 in the RHS.

In the second expression, the second `a` in the pattern is looking for a 1 in the second element of the RHS, which is not there. As a result, the match fails.

However, a variable can be bound to a new value in a subsequent match, and its current value does not participate in the new match.

```
iex> a = 1
1
iex> [1, a, 3] = [1, 2, 3]
[1, 2, 3]
iex> a
2
```

What if you instead want to force Elixir to use the existing value of the variable in the pattern? Prefix it with an ^ (a caret).

```
iex> a = 1
1
iex> a = 2
2
iex> ^a = 1
** (MatchError) no match of right hand side value: 1
```

This also works if the variable is a component of a pattern:

```
iex> a = 1
1
iex> [^a, 2, 3 ] = [ 1, 2, 3 ]      # use existing value of a
[1, 2, 3]
iex> a = 2
2
iex> [ ^a, 2 ] = [ 1, 2 ]
** (MatchError) no match of right hand side value: [1, 2]
```

There's one more important part of pattern matching, which we'll look at when we start digging deeper into lists on page 63.

Your turn…

➤ *Exercise: PatternMatching-2*

Which of the following will match?

Prepared exclusively for Richard Bishop

- `[ a, b, a ] = [ 1, 2, 3 ]`
- `[ a, b, a ] = [ 1, 1, 2 ]`
- `[ a, b, a ] = [ 1, 2, 1 ]`

➤ *Exercise: PatternMatching-3*

If you assume the variable `a` initially contains the value `2`, which of the following will match?

- `[ a, b, a ] = [ 1, 2, 3 ]`
- `[ a, b, a ] = [ 1, 1, 2 ]`
- `a = 1`
- `^a = 2`
- `^a = 1`
- `^a = 2 - a`

Another way of looking at the equals sign

Elixir's pattern matching is similar to that of Erlang (the main difference being that Elixir allows a match to assign to a variable that was assigned in a prior match, whereas in Erlang a variable can only be assigned once).

Joe Armstrong, the creator of Erlang, compares the equals sign in Erlang to that used in algebra. When you write the equation $x = a+1$, you are not assigning the value of $a+1$ to x. Instead, you're simply asserting that the expressions x and $a+1$ have the same value. If you know the value of x, you can work out the value of a, and vice versa.

His point is that you had to unlearn that meaning when you first came across assignment in imperative programming languages. Now's the time to ununlearn it.

That's why I talk about pattern matching as the first chapter in this part of the book. It is a core part of Elixir—we'll also use it in conditions, function calls, and function invocation.

But, really, I wanted to get you thinking differently about programming languages, and to show you that some of your existing assumptions won't work in Elixir.

Prepared exclusively for Richard Bishop

Change and decay in all around I see…

➤ *Henry Francis Lyte, Abide With Me*

CHAPTER 3

Immutability

If you listen to functional programming aficionados, you'll hear people making a big deal about immutability—the fact that in a functional program, data cannot be altered once created.

And, indeed, Elixir enforces immutable data.

But why?

You Already Have (Some) Immutable Data

Forget about Elixir for a moment. Think about your current programming language of choice. Let's imagine you'd written:

```
count = 99
do_something_with(count)
print(count)
```

You'd expect it to output 99. In fact, you'd be very surprised if it didn't. At your very core, you believe that 99 will always have the value 99.

Now, you could obviously bind a new value to your *variable*, but that doesn't change the fact that the value 99 is still 99.

Now imagine programming in a world where you couldn't rely on that—where some other code, possibly running in parallel with your own, could change the value of 99. In that world, the call to `do_something_with` might kick off code that runs in the background, passing it the value 99 as an argument. And that could change the contents of the parameter it receives. Suddenly, 99 could be 100.

You'd be (rightly) upset. And, what's worse, you'd never really be able to guarantee your code produced the correct results.

Now, still thinking about your current language, consider

Prepared exclusively for Richard Bishop

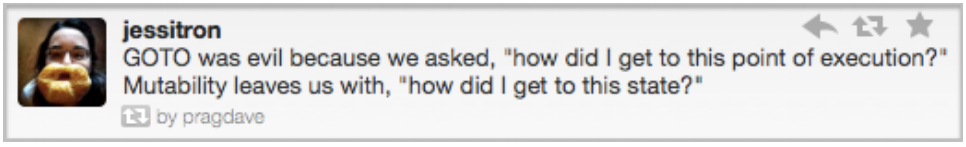
```
array = [ 1, 2, 3 ]
do_something_with(array)
print(array)
```

Again, you'd hope that the print call will output [1,2,3]. But in most languages, do_something_with will receive the array as a reference. If it decides to change the second element, or delete the contents entirely, the output won't be what you expect. Now it's harder to look at your code and reason about what it does.

Take this a step further—run multiple threads, all with access to the array. Who knows what state it will be in if they all start changing it?

All this is because most compound data structures in most programming languages are mutable—you can change all or part of their content. And if pieces of your code do this in parallel, you're in a world of hurt.

By coincidence, Jessica Kerr (@jessitron) tweeted the following on the day I updated this section.



It's spot on.

Immutable Data Is Known Data

Elixir sidesteps these problems. In Elixir, all values are immutable. The most complex nested list, the database record—all these things behave just like the simplest integer. Their values are all immutable.

In Elixir, once a variable references a list such as [1,2,3], you know it will always reference those same values (until you rebind the variable). And this makes concurrency a lot less frightening.

But what if you *need* to add 100 to each element in [1,2,3]. Elixir does it by producing a copy of the original containing the new values. The original remains unchanged, and any other code holding a reference to that original will not be affected by your operation.

When you think about it, this fits in nicely with the idea that programming is about transforming data. When we update [1,2,3], we don't hack it in place. Instead we transform it into something new.

Prepared exclusively for Richard Bishop

Performance Implications of Immutability

You might feel that all this copying is inefficient, but the reverse is true. Because Elixir knows that existing data is immutable, it can reuse it, in part or as a whole, when building new structures.

Consider this code. (It uses a new operator, [head | tail] which builds a new list with head as its first element and tail as the rest. We'll spend a whole chapter on this when we talk about Lists and Recursion. For now, just trust….)

```
iex> list1 = [ 3, 2, 1 ]
[3, 2, 1]
iex> list2 = [ 4 | list1 ]
[4, 3, 2, 1]
```

In most languages, list2 would be built by creating a new list containing a 4, a 3, a 2, and a 1. The three values in list1 would be copied in to the tail of list2. And that would be necessary, because list1 would be mutable.

But Elixir knows that list1 will never change, so it simply constructs a new list with a head of 4 and a tail of list1.

Garbage Collection

The other issue with a transformational language is that you quite often end up leaving old values unused when you create new values from them. This ends up leaving a bunch of old values on the heap, using up memory, so garbage collection has to reclaim them.

Most modern languages have a garbage collector, and developers have (rightly) grown to be suspicious of them—they can impact performance quite badly.

But the cool thing about Elixir is that you write your code using lots and lots of processes, and each process has its own heap. The data in your application is divvied up between these processes, so each individual heap is much, much smaller than would have been the case if all the data had been in a single heap. As a result, garbage collection runs faster. And for those times when a process terminates before its heap becomes full, all its data is simply discarded—no garbage collection is required.

Coding With Immutable Data

Once you accept the concept, coding with immutable data is surprisingly easy. You just have to remember that any function that transforms data will

Prepared exclusively for Richard Bishop

return a new copy of it. Thus, we never capitalize a string. Instead, we return a capitalized copy of a string.

```
iex> name = "elixir"
"elixir"
iex> cap_name = String.capitalize name
"Elixir"
iex> name
"elixir"
```

If you're coming from an OO language, you'll probably initially dislike the idea that we write `String.capitalize name` and not `name.capitalize()`. But, in OO languages, objects mostly have mutable state. When you make a call such as `name.capitalize()` you have no immediate indication whether you are changing the internal representation of the name, returning a capitalized copy, or both. There's plenty of scope for ambiguity.

In a functional language, we *always* transform data. We never modify it in place. The syntax reminds us of this every time we use it.

So, that's enough theory. Let's start learning the language. In the next chapter, we'll quickly go over the basic data types and some syntax, and in the chapters following it we'll look at functions and modules.

Prepared exclusively for Richard Bishop

In this chapter, we'll see
- 5 value types
- 2 system types
- 4 collection types
- naming, operators, etc

CHAPTER 4

Elixir Basics

In this chapter we'll look at the types that are baked in to Elixir, along with a few other things you need to know to get started. This chapter is deliberately terse—you're a programmer and you know what an integer is, so I'm not going to insult you. Instead, I try to cover the Elixir-specific stuff you'll need to know.

Elixir's built-in types are:

- Value types:
  - Arbitrary-sized integers
  - Floating point numbers
  - Atoms
  - Regular expressions
  - Ranges
- System types:
  - PIDs and Ports
  - References
- Collection types
  - Tuples
  - Lists
  - Maps
  - Binaries

In Elixir, functions are a type, too. They have their own chapter, following this one.

You might be surprised that this list doesn't include things such as strings and structures. Well, Elixir has them, but they are built using the basic types from this list. However, they are important. Strings have their own chapter,

Prepared exclusively for Richard Bishop

and we have a couple of chapters on lists and maps (and other dictionary-like types). The maps chapter also describes the Elixir structure facilities.

Finally, there's some debate about whether regular expressions and ranges are value types. Technically, they aren't—underneath the covers they are actually just structures. But right now it's convenient to treat them as being distinct types.

Value Types

The value types in Elixir represent numbers, names, ranges, and regular expressions.

Integers

Integer literals can be written as decimal (`1234`), hexadecimal (`0xcafe`), octal (`0765`), and binary (`0b1010`).

Decimal numbers may contain underscores—these are often used to separate groups of 3 digits when writing large numbers, so one million could be written `1_000_000`.

There is no fixed limit on the size of integers—their internal representation grows to fit their magnitude.

```
factorial(10000) # => 28462596809170545189...and so on for 35640 more digits...
```

(You'll see how to write a function such as `factorial` in the chapter on Modules and Named Functions on page 45.)

Floating point numbers

Floating point numbers are written using a decimal point. There must be at least one digit before and after the decimal point. An optional trailing exponent may be given. These are all valid floating point literals:

```
1.0   0.2456   0.314158e1 314159.0e-5
```

Floats are IEEE 754 double precision, giving them about 16 digits accuracy and a maximum exponent of around 10^{308}.

Atoms

Atoms are constants that represent the name of something. They are written using a leading colon (:). The colon can be followed by an atom word[1] or an

1. an atom word is a sequence of letters, digits, and underscores. It may contain at signs and end with an exclamation point or question mark.

Prepared exclusively for Richard Bishop

Elixir operator. You can also create atoms containing arbitrary characters by enclosing the characters following the colon in double quotes. (Ruby calls atoms *symbols*.) These are all atoms:

```
:fred  :is_binary?  :var@2  :<>  :===  :"func/3"  :"long john silver"
```

One thing that makes atoms invaluable is that their name is their value. Two atoms with the same name will always compare as being equal, even if they were created by different applications sitting on two computers separated by an ocean.

We'll be using atoms a lot to tag values.

Ranges

Ranges are represented as *start*..*end*, where *start* and *end* can be values of any type. However, if you want to iterate over the values in a range, the two extremes must be integers.

Regular expressions

Elixir has regular expression literals, written as ~r{regexp} or ~r{regexp}opts.[2] Elixir regular expression support is provided by PCRE,[3] which basically provides a Perl-5 compatible syntax for patterns.

You can specify one or more single-character options following a regexp literal. These modify its match behavior or add additional functionality:

Opt	Meaning
f	the pattern must start to match on the first line of a multiline string
g	support named groups
i	make matches case insensitive
m	if the string to be matched contains multiple lines, ^ and $ match the start and end of these lines. \A and \z continue to match the beginning or end of the string
r	normally modifiers like * and + are greedy, matching as much as possible. The r modifier makes them *reluctant*, matching as little as possible
s	allows . to match any newline characters
u	enable unicode specific patterns like \p

2. Here I show the delimiters for regular expression literals as { and }, but they are considerably more flexible. You can actually chose any nonalphanumeric characters as delimiters, as described when we talk about sigils on page 110. Some people use ~r/.../, for nostalgic reasons, but this is actually less convenient than the bracketed forms, as any forward slashes inside the pattern must be escaped.
3. http://www.pcre.org/

Prepared exclusively for Richard Bishop

Opt	Meaning
x	extended mode—ignore whitespace and comments (# to end of line)

You manipulate regular expressions with the Regex module.

```
iex> Regex.run ~r{[aeiou]}, "caterpillar"
["a"]
iex> Regex.scan ~r{[aeiou]}, "caterpillar"
[["a"], ["e"], ["i"], ["a"]]
iex> Regex.split ~r{[aeiou]}, "caterpillar"
["c", "t", "rp", "ll", "r"]
iex> Regex.replace ~r{[aeiou]}, "caterpillar", "*"
"c*t*rp*ll*r"
```

System Types

These types reflect resources in the underlying Erlang VM.

Pids and Ports

A pid is a reference to a local or remote process, and a port is a reference to a resource (typically external to the application) that you'll be reading or writing.

The pid of the current process is available by calling self. A new pid is created when you spawn a new process. We'll talk about this in Part 2.

References

The function make_ref creates a globally unique reference; no other reference will be equal to it. We don't use references in this book.

Collection Types

The types we've seen so far are common in other programming languages. Now we're getting into types that are more exotic. Because of this, we're going to go into more detail here.

Elixir collections can hold values of any type (including other collections).

Tuples

A tuple is an ordered collection of values. As with all Elixir data structures, once created it cannot be modified.

You write a tuple between braces, separating the elements with commas.

```
{ 1, 2 }      { :ok, 42, "next"  }   { :error, :enoent }
```

Prepared exclusively for Richard Bishop

A typical Elixir tuple has two to four elements—any more and you'll probably want to look at maps on page 77 or structs on page 82.

You can use tuples in pattern matching:

```
iex> {status, count, action} = {:ok, 42, "next"}
{:ok, 42, "next"}
iex> status
:ok
iex> count
42
iex> action
"next"
```

It is common for functions to return a tuple where the first element is the atom :ok if there were no errors. For example, assuming you have a file called Rakefile in your current directory:

```
iex> {status, file} = File.open("Rakefile")
{:ok, #PID<0.39.0>}
```

Because the file was successfully opened, the tuple contains :ok for the status and a PID, which is how we actually access the contents.

A common idiom is to write matches that assume success:

```
iex> { :ok, file } = File.open("Rakefile")
{:ok, #PID<0.39.0>}
iex> { :ok, file } = File.open("non-existent-file")
** (MatchError) no match of right hand side value: {:error, :enoent}
```

The second open failed, and returned a tuple where the first element was :error. This caused the match to fail, and the error message shows the second element contains the reason—enoent is Unix-speak for "file does not exist."

Lists

We've already seen Elixir's list literal syntax, [1,2,3]. This might lead you to think that they are like arrays in other languages, but they are not (in fact, tuples are the closest Elixir gets to a conventional array). Instead, a list is effectively a linked data structure. A list may either be empty or it consists of a head and a tail. The head contains a value and the tail is itself a list. (If you've used the language Lisp, then this will all seem very familiar).

As we'll see in Chapter 7, *Lists and Recursion*, on page 63, this recursive definition of a list turns out to be the core of much of the programming we do in Elixir.

Prepared exclusively for Richard Bishop

Because of their implementation, lists are easy to traverse linearly, but they are expensive to access in random order (to get to the n^{th} element, you have to scan through the n-1 previous elements). It is always cheap to get the head of a list, and to extract the tail of a list.

Lists have one other performance characteristic. Remember that we said that all Elixir data structures are immutable? That means that once a list has been made, it will never be changed. So, if we want to remove the head from a list, leaving just the tail, we never have to copy the list. Instead, we can just return a pointer to the tail. This is the basis of all the list traversal tricks we'll see in the chapter on *Lists and Recursion*.

Elixir has some operators that work specifically on lists:

```
iex>  [ 1, 2, 3 ] ++ [ 4, 5, 6 ]      # concatenation
[1, 2, 3, 4, 5, 6]
iex> [1, 2, 3, 4] -- [2, 4]           # difference
[1, 3]
iex> 1 in [1,2,3,4]                   # membership
true
iex> "wombat" in [1, 2, 3, 4]
false
```

Keyword Lists

Because we often need simple lists of key/value pairs, Elixir gives us a shortcut. If you write

```
[ name: "Dave", city: "Dallas", likes: "Programming" ]
```

Elixir converts it into a list of 2-value tuples:

```
[ {:name, "Dave"}, {:city, "Dallas"}, {:likes, "Programming"} ]
```

Elixir allows you to leave off the square brackets if a keyword list is the last argument in a function call. Thus

```
DB.save record, [ {:use_transaction, true}, {:logging, "HIGH"} ]
```

can be written more cleanly as

```
DB.save record, use_transaction: true, logging: "HIGH"
```

You can also leave off the brackets if a keyword list appears as the last item in any context where a list of values is expected.

```
iex> [1, fred: 1, dave: 2]
[1, {fred: 1}, {dave: 2}]
iex> {1, fred: 1, dave: 2}
{1, [fred: 1, dave: 2]}
```

Prepared exclusively for Richard Bishop

Maps

A map is a collection of key/value pairs. A map literal looks like this:

```
%{ key => value, key => value }
```

Here are some maps:

```
iex> states = %{ "AL" => "Alabama", "WI" => "Wisconsin" }
%{"AL" => "Alabama", "WI" => "Wisconsin"}

iex> response_types = %{ { :error, :enoent } => :fatal,
...>                     { :error, :busy } => :retry }
%{{:error, :busy} => :retry, {:error, :enoent} => :fatal}

iex> colors = %{ :red => 0xff0000, :green => 0x00ff00, :blue => 0x0000ff }
%{blue: 255, green: 65280, red: 16711680}
```

In the first case, the keys are strings, in the second they are tuples, and in the third they're atoms. Although it is typical that all the keys in a map are the same type, it isn't required.

```
iex> %{ "one" => 1, :two => 2, {1,1,1} => 3 }
%{:two => 2, {1, 1, 1} => 3, "one" => 1}
```

If the key is an atom, you can use the same shortcut that you use with keyword lists:

```
iex> colors = %{ red: 0xff0000, green: 0x00ff00, blue: 0x0000ff }
%{blue: 255, green: 65280, red: 16711680}
```

Why do we have both maps and keyword lists? Maps only allow one entry for a particular key, whereas keyword lists allow the key to be repeated. Maps are efficient (particularly as they grow), and they can be used in Elixir's pattern matching, which we discuss in later chapters.

In general, use keyword lists for things such as command line parameters and for passing around options, and use maps (or another data structure, the HashDict) when you want an associative array.

Accessing a Map

You extract values from a map using the key. The square-bracket syntax works with all maps:

```
iex> states = %{ "AL" => "Alabama", "WI" => "Wisconsin" }
%{"AL" => "Alabama", "WI" => "Wisconsin"}
iex> states["AL"]
"Alabama"
iex> states["TX"]
```

Prepared exclusively for Richard Bishop

```
nil

iex> response_types = %{ { :error, :enoent } => :fatal,
...>                     { :error, :busy } => :retry }
%{{:error, :busy} => :retry, {:error, :enoent} => :fatal}
iex> response_types[{:error,:busy}]
:retry
```

If the keys are atoms, you can also use a dot notation:

```
iex> colors = %{ red: 0xff0000, green: 0x00ff00, blue: 0x0000ff }
%{blue: 255, green: 65280, red: 16711680}
iex(27)> colors[:red]
16711680
iex(28)> colors.green
65280
```

You'll get a KeyError if there's no matching key if you use the dot notation.

Binaries

Sometimes you need to access data as a sequence of bits and bytes. For example, the headers in JPEG and MP3 files contain fields where a single byte may encode 2 or 3 separate values.

Elixir supports this with the binary data type. Binary literals are enclosed between << and >>.

The basic syntax packs successive integers into bytes:

```
iex> bin = << 1, 2 >>
<<1, 2>>
iex> byte_size bin
2
```

You can add modifiers to control the type and size of each individual field. Here's a single byte that contains three fields of widths 2, 4, and 2 bits. (The example uses some built-in libraries to show the binary value of the result.)

```
iex> bin = <<3 :: size(2), 5 :: size(4), 1 :: size(2)>>
<<213>>
iex> :io.format("~-8.2b~n", :binary.bin_to_list(bin))
11010101
:ok
iex> byte_size bin
1
```

Binaries are both important and arcane. They're important because Elixir uses them to represent UTF strings, They're arcane because, at least initially, you're unlikely to use them directly.

Prepared exclusively for Richard Bishop

Names, Source Files, Conventions, Operators, and So On

Identifiers in Elixir are combinations of upper and lower case ASCII characters, digits, and underscores. They may end with a question mark or an exclamation point.

Module, record, protocol, and behavior names start with an uppercase letter and are BumpyCase. All other identifiers start with a lower case letter or underscore, and by convention use underscores between words. If the first character is an underscore, Elixir doesn't report a warning if the variable is unused in a pattern match or function parameter list.

Source files are written in UTF-8, but identifiers may only use ASCII.

By convention, source files use two-character indentation to identify nest, and use spaces, not tabs, to achieve this.

Comments start with a hash (#) and run to the end of the line.

The community is compiling a coding style guide. As I write this, it is at https://github.com/alco/elixir/wiki/Contribution-Guidelines#coding-style, but I'm told that it may move in the future.

Truth

Elixir has three special values related to boolean operations, true, false, and nil. nil is treated as false in boolean contexts.

(A bit of trivia: all three of these values are simply aliases for atoms of the same name, so true is the same as the atom :true.)

In most contexts, any value other than false or nil is treated as being true. We sometimes refer to this as *truthy*, to differentiate them from the actual value true.

Operators

Elixir has a very rich set of operators. Here's a subset that we'll use in this book.

comparison operators

```
a === b    # strict equality (so 1 === 1.0 is false)
a !== b    # strict inequality (so 1 !== 1.0 is true)
a ==  b    # value equality  (so 1 == 1.0 is true)
a !=  b    # value inequality  (so 1 != 1.0 is false)
a  >  b    # normal comparison
a >=  b    #   :
a  <  b    #   :
```

Prepared exclusively for Richard Bishop

```
a <=  b    #   :
```

The ordering comparisons in Elixir are less strict than in many languages, as you can compare values of different type. If the types are the same, or are compatible (for example 3 > 2 or 3.0 < 5) the comparison uses natural ordering. Otherwise comparison is based on type according to the rule

number < atom < reference < function < port < pid < tuple < map < list < binary

boolean operators

(These operators expect true or false as their first argument.)

```
a or  b    # true if a is true, otherwise b
a and b    # false if a is false, otherwise b
a xor b    # true if only one of a or b is true
not a      # false if a is true, true otherwise
```

relaxed boolean operators

These operators take arguments of any type. Any value apart from nil or false is interpreted as true.

```
a || b  # a if a is truthy, otherwise b
a && b  # b if a is truthy, otherwise a
!a      # false if a is truthy, otherwise true
```

arithmetic operators

+ - * / div rem

Integer division yields a floating point result. Use div(a,b) to get an integer result.

rem is the *remainder operator*. It is called as a function (rem(11, 3) => 2). It differs from normal modulo operations in that the result will have the same sign as its first argument.

join operators

```
binary1 <> binary2   # concatenates two binaries (later, we'll
                     # see that binaries include strings)
list1   ++ list2     # concatenates two lists
list1   -- list2     # set difference between two lists
```

the in operator

```
a in enum      # tests if a is included in enum (for example,
               # a list or a range)
```

Prepared exclusively for Richard Bishop

End of the Basics

We've now seen the low-level ingredients of an Elixir program. In the next two chapters, we'll see how to create anonymous functions, and then how to create modules and named functions.

Prepared exclusively for Richard Bishop

CHAPTER 5

Anonymous Functions

Elixir is a functional language, so it is no surprise that functions are a basic type.

An anonymous function is created using the `fn` keyword.

```
fn
  parameter-list -> body
  parameter-list -> body ...
end
```

Think of `fn`...`end` as being a bit like the quotes that surround a string literal, except here we're returning a function as a value, and not a string. We can pass that function value to other functions. We can also invoke it, passing in arguments.

At its simplest, a function has a parameter list and a body, separated by `->`.

For example, the following defines a function, binding it to the variable `sum`, and then calls it.

```
iex> sum = fn (a, b) -> a + b end
#Function<12.17052888 in :erl_eval.expr/5>
iex> sum.(1, 2)
3
```

The first line of code creates a function that takes two parameters (named `a` and `b`). The implementation of the function follows the `->` arrow (in our case it simply adds the two parameters), and the whole thing is terminated with the keyword `end`. We store the function in the variable `sum`.

On the second line of code, we invoke the function using the syntax `sum.(1,2)`. The dot indicates the function call, and the arguments are passed between parentheses. (You'll have noticed we don't put a dot for named function calls—this is a difference between named and anonymous functions.)

Prepared exclusively for Richard Bishop

If your function takes no arguments, you still need the parentheses to call it:

```
iex> greet = fn -> IO.puts "Hello" end
#Function<12.17052888 in :erl_eval.expr/5>
iex> greet.()
Hello
:ok
```

You can, however, omit the parentheses around the parameters in the function definition, and no parentheses are needed if the function takes no parameters.

```
iex> f1 = fn a, b -> a * b end
#Function<12.17052888 in :erl_eval.expr/5>
iex> f1.(5,6)
30
iex> f2 = fn -> 99 end
#Function<12.17052888 in :erl_eval.expr/5>
iex> f2.()
99
```

Functions and Pattern Matching

When we call `sum.(2,3)`, it's easy to assume that we simply assign 2 to the parameter `a` and 3 to `b`. But that word, *assign*, should ring some bells. Elixir doesn't have assignment. Instead, it tries to match values to patterns. (We came across this when we looked at pattern matching and assignment on page 13.)

If we write

```
a = 2
```

then Elixir makes the pattern match by binding `a` to the value 2. And that's exactly what happens when our `sum` function gets called. We pass 2 and 3 as arguments, and Elixir tries to match it to the parameters `a` and `b`, which it does by giving `a` the value 2 and `b` the value 3. It's the same as when we write

```
{a, b} =  {1, 2}
```

So this means we can also perform more complex pattern matching when we call a function. For example, the following function reverses the order of elements in a two-element tuple.

```
iex> swap = fn { a, b } -> { b, a } end
#Function<12.17052888 in :erl_eval.expr/5>
iex> swap.( { 6, 8 } )
{8, 6}
```

We'll use this pattern matching capability when we look at functions with multiple implementations.

Prepared exclusively for Richard Bishop

Your turn…

➤ *Exercise: Functions-1*

Go into iex. Create and run the functions that do the following

- list_concat.([1,2,3], [4,5,6]) #=> [1,2,3,4,5,6]
- sum.(1, 2, 3) #=> 6
- pair_tuple_to_list.({ 8, 7 }) #=> [8, 7]

One Function, Multiple Bodies

A single function definition lets you define different implementations depending on the type and contents of the arguments passed. (You cannot select based on the number of arguments—each clause in the function definition must have the same number of parameters.)

At its simplest, we can use pattern matching to select which clause to run. In the example that follows, we know that the tuple returned by File.open has :ok as its first element if the file was opened, so we write a function that displays either the first line of a successfully opened file or a simple error message if the file could not be opened.

```
Line 1 iex> handle_open = fn
   2 ...>   {:ok, file}  -> "Read data: #{IO.read(file, :line)}"
   3 ...>   {_,   error} -> "Error: #{:file.format_error(error)}"
   4 ...> end
   5 #Function<12.17052888 in :erl_eval.expr/5>
   6 iex> handle_open.(File.open("code/intro/hello.exs"))      #   this file exists
   7 "Read data: IO.puts \"Hello, World!\"\n"
   8 iex> handle_open.(File.open("nonexistent"))   # this one doesn't
   9 "Error: no such file or directory"
```

Let's start by looking inside the function definition. On lines 2 and 3 we define two separate function bodies. Each takes a single tuple as a parameter. The first of them requires that the first term in the tuple is :ok. The second line uses the special variable _ (underscore) to match any other value for the first term.

Now look at line 6. We call our function passing it the result of calling File.open on a file that exists. This means the function will receive the tuple {:ok,file}, and this matches the clause on line 2. The corresponding code calls IO.read to read the first line of this file.

We then call handle_open again this time with the result of trying to open a file that does not exist. The tuple that is returned ({:error,:enoent}) is passed to our function which looks for a matching clause. It fails on line 2, because the

Prepared exclusively for Richard Bishop

first term is not :ok, but it succeeds on the next line. The code in that clause formats up the error as a nice string.

There are a couple of other things to note in this code. On line 3 we call :file.format_error. The :file part of this refers to the underlying Erlang File module, so we can call its format_error function. Contrast this with the call to File.open on line 6. Here, the File part refers to Elixir built-in module. This is a good example of the underlying environment leaking through into Elixir code. It is good that you can access all the existing Erlang libraries—there are hundreds of years of effort in there just waiting for you to use. But it is also tricky, because you have to differentiate between Erlang functions and Elixir functions when you call them.

And, finally, this example shows off Elixir's *string interpolation*. Inside a string, the contents of #{...} are evaluated and the result is substituted back in.

Working with larger code examples

Our handle_open function is getting uncomfortably long to type directly into iex. One typo, and you have to type it all in again.

Instead, use your editor to type it into a file in the same directory that you were in when you started iex. Let's call the file handle_open.exs.

first_steps/handle_open.exs

```
handle_open = fn
 {:ok, file}  -> "First line: #{IO.read(file, :line)}"
 {_,   error} -> "Error:  #{:file.format_error(error)}"
end
IO.puts handle_open.(File.open("Rakefile"))      # call with a file that exists
IO.puts handle_open.(File.open("nonexistent"))   # and then with one that doesn't
```

Now, inside iex, type:

```
c "handle_open.exs"
```

This compiles and runs the code in the given file.

You can do the same thing from the command line (that is, not inside iex) using:

```
$ elixir handle_open.exs
```

We used the file extension .exs for this example. This is used for code that is meant to be run directly from a source file (think of the 's' meaning *script*). For files that you want to compile and use later, you'll use the .ex extension.

Your turn...

Prepared exclusively for Richard Bishop

➤ *Exercise: Functions-2*

Write a function that takes three arguments. If the first two are zero, return "FizzBuzz". If the first is zero, return "Fizz". If the second is zero return "Buzz". Otherwise return the third argument. Do not use any language features that we haven't yet covered in this book.

➤ *Exercise: Functions-3*

The operator rem(a, b) returns the remainder after dividing a by b. Write a function that takes a single integer (n) and which calls the function in the previous exercise, passing it rem(n,3), rem(n,5), and n. Call it 7 times with the arguments 10, 11, 12, etc. You should get "Buzz, 11, Fizz, 13, 14, FizzBuzz, 16".

(Yes, it's a FizzBuzz[1] solution with no conditional logic).

Functions Can Return Functions

Here's some strange code:

```
iex> fun1 = fn -> fn -> "Hello" end end
#Function<12.17052888 in :erl_eval.expr/5>
iex> fun1.()
#Function<12.17052888 in :erl_eval.expr/5>
iex> fun1.().()
"Hello"
```

The strange thing is the first line. It's a little hard to read, so let's spread it out.

```
fun1 = fn ->
          fn ->
             "Hello"
          end
       end
```

The variable fun1 is bound to a function. That function takes no parameters, and its body is a second function definition. That second function also takes no parameters, and it evaluates the string "Hello".

When we call the outer function (using fun1.()), it returns the inner function. When we call that (fun1.().()) the inner function is evaluated and "Hello" is returned.

1. http://c2.com/cgi/wiki?FizzBuzzTest

Prepared exclusively for Richard Bishop

You wouldn't normally write something such as fun1.().(). But you might call the outer function and bind the result to a separate variable. You might also use parentheses to make the inner function more obvious.

```
iex> fun1 = fn -> (fn -> "Hello" end) end
#Function<12.17052888 in :erl_eval.expr/5>
iex> other = fun1.()
#Function<12.17052888 in :erl_eval.expr/5>
iex> other.()
"Hello"
```

Functions Remember Their Original Environment

Let's take this idea of nesting functions a little further.

```
iex> greeter = fn name -> (fn -> "Hello #{name}" end) end
#Function<12.17052888 in :erl_eval.expr/5>
iex> dave_greeter = greeter.("Dave")
#Function<12.17052888 in :erl_eval.expr/5>
iex> dave_greeter.()
"Hello Dave"
```

Now the outer function has a name parameter. Like any parameter, name is available for use throughout the body of the function. In this case, we use it inside the string in the inner function.

When we call the outer function, it returns the inner function definition. At this point it has not yet substituted the name into the string. But when we call this inner function (dave_greeter.()), the substitution takes place, and the greeting appears.

But something strange happened here. The inner function uses the name parameter of the outer function. But by the time greeter.("Dave") returns, that outer function has finished executing, and the parameter has gone out of scope. And yet when we run the inner function, it merrily uses that parameter's value.

This works because functions in Elixir automatically carry with them the bindings of variables in the scope in which they are defined. In our example, the variable name is bound in the scope of the outer function. When the inner function is defined, it inherits this scope, and carries the binding of name around with it. This is a *closure*—the scope encloses the bindings of its variables, packaging them into something that can be saved and used later.

Let's play with this some more.

Prepared exclusively for Richard Bishop

Parameterized Functions

In the previous example, the outer function took an argument, and the inner one did not. Let's try a different example where both take arguments.

```
iex> add_n = fn n -> (fn other -> n + other end) end
#Function<12.17052888 in :erl_eval.expr/5>
iex> add_two = add_n.(2)
#Function<12.17052888 in :erl_eval.expr/5>
iex> add_five = add_n.(5)
#Function<12.17052888 in :erl_eval.expr/5>
iex> add_two.(3)
5
iex> add_five.(7)
12
```

Here the inner function adds the value of its parameter, `other` to the value of the outer function's parameter, `n`. Each time we call the outer function, we give it a value for `n`, and it returns a function that adds `n` to its own parameter.

Your turn…

➤ *Exercise: Functions-4*

Write a function `prefix` that takes a string. It should return a new function that takes a second string. When that second function is called, it will return a string containing the first string, a space, and the second string.

```
iex> mrs = prefix.("Mrs")
#Function<erl_eval.6.82930912>
iex> mrs.("Smith")
"Mrs Smith"
iex> prefix.("Elixir").("Rocks")
"Elixir Rocks"
```

Passing Functions as Arguments

Functions are just values, so we can pass them to other functions.

```
iex> times_2 = fn n -> n * 2 end
#Function<12.17052888 in :erl_eval.expr/5>
iex> apply = fn (fun, value) -> fun.(value) end
#Function<12.17052888 in :erl_eval.expr/5>
iex> apply.(times_2, 6)
12
```

In this example, `apply` is a function that takes a function and a value. It returns the result of invoking that function with the value as an argument.

Prepared exclusively for Richard Bishop

It turns out we use the ability to pass functions around pretty much everywhere in Elixir code. For example, the built-in Enum module has a function called map. It takes two arguments, a collection and a function. It returns a list which is the result of applying that function to each element of the collection. Here are some examples:

```
iex> list = [1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
iex> Enum.map list, fn elem -> elem * 2 end
[2, 6, 10, 14, 18]
iex> Enum.map list, fn elem -> elem * elem end
[1, 9, 25, 49, 81]
iex> Enum.map list, fn elem -> elem > 6 end
[false, false, false, true, true]
```

The & Notation

The strategy of creating short helper functions is so common that Elixir provides a shortcut. Let's look at it in use before I explain what's going on.

```
iex> add_one = &(&1 + 1)       # same as add_one = fn (n) -> n + 1 end
#Function<6.17052888 in :erl_eval.expr/5>
iex> add_one.(44)
45
iex> square = &(&1 * &1)
#Function<6.17052888 in :erl_eval.expr/5>
iex> square.(8)
64
iex> speak = &(IO.puts(&1))
&IO.puts/1
iex> speak.("Hello")
Hello
:ok
```

The & operator converts the expression that follows into a function. Inside that expression, the placeholders &1, &2, and so on, correspond to the first, second, and subsequent parameters of the function. So &(&1 + &2) will be converted to fn p1, p2 -> p1 + p2 end and &byte_size(&1) will become fn p1 -> byte_size(p1) end.

(And, in case you're wondering about the erl_eval in the previous iex session, remember that Elixir runs on the Erlang VM.)

Elixir is even cleverer. Look at the speak line in the previous code. Normally, Elixir would have generated an anonymous function, so &(IO.puts(&1)) would become fn x -> IO.puts(x) end. But Elixir noticed that the body of the anonymous function was simply a call to a named function (the IO function puts) and that the parameters were in the correct order (that is, the first parameter to the

Prepared exclusively for Richard Bishop

anonymous function was the first parameter to the named function, and so on). So Elixir optimized away the anonymous function, replacing it with a direct reference to the function, IO.puts/1.

For this to work, the arguments must be in the correct order:

```
iex> rnd = &(Float.round(&1, &2))
&Float.round/2
iex> rnd = &(Float.round(&2, &1))
#Function<12.17052888 in :erl_eval.expr/5>
```

Literal lists and tuples can also be turned into functions.[2] Here's a function that returns a tuple containing the quotient and remainder of dividing two integers.

```
iex> divrem = &{ div(&1,&2), rem(&1,&2) }
#Function<12.17052888 in :erl_eval.expr/5>
iex> divrem.(13, 5)
{2, 3}
```

There's a second form of the & function capture operator. You can give it the name and arity (number of parameters) of an existing function, and it will return an anonymous function that calls it. The arguments you pass to the anonymous function will in turn be passed to the named function. We've already seen this: when we entered &(IO.puts(&1)) into iex, it displayed the result as &IO.puts/1. In this case, puts is a function in the IO module, and it takes one argument. The Elixir way of naming this is to write IO.puts/1. Put an & in front of this, and you wrap it in a function. Here are some other examples:

```
iex> l = &length/1
&:erlang.length/1
iex> l.([1,3,5,7])
4
iex> len = &Enum.count/1
&Enum.count/1
iex> len.([1,2,3,4])
4
iex> m = &Kernel.min/2   # This is just an alias for the Erlang function
&:erlang.min/2
iex> m.(99,88)
88
```

This works with named functions that you write, as well (but we haven't covered how to write them yet).

The & shortcut gives us a wonderful way to pass functions to other functions.

2. [] and {} are actually operators in Elixir.

Prepared exclusively for Richard Bishop

```
iex> Enum.map [1,2,3,4], &(&1 + 1)
[2, 3, 4, 5]
iex> Enum.map [1,2,3,4], &(&1 * &1)
[1, 4, 9, 16]
iex> Enum.map [1,2,3,4], &(&1 < 3)
[true, true, false, false]
```

Your turn…

➤ *Exercise: Functions-5*

Use the &… notation to rewrite the following.

- Enum.map [1,2,3,4], fn x -> x + 2 end
- Enum.each [1,2,3,4], fn x -> IO.inspect x end

Functions Are The Core

At the start of the book, we said that the basis of programming is transforming data. Functions are the little engines that perform that transformation. They are at the very heart of Elixir.

So far, we've been looking at anonymous functions—although we can bind them to variables, the functions themselves have no name.

Elixir also has named functions. In the next chapter we'll see how to work with them.

Prepared exclusively for Richard Bishop

CHAPTER 6

Modules and Named Functions

Once a program grows beyond a couple of lines, you're going to want to structure it. Elixir makes this easy. You break your code into *named functions*, and organize these functions into *modules*. In fact, in Elixir named functions must be written inside modules.

Let's look at a simple example. Navigate to a working directory and create an Elixir source file called times.exs.

mm/times.exs

```
defmodule Times do
  def double(n) do
    n * 2
  end
end
```

I'm going to show you two ways to compile this file and load it into iex. First, if you're at the command line, you can do this:

```
$ iex times.exs
iex> Times.double 4
8
```

Give iex the name of a source file, and it compiles and loads it before it displays a prompt.

If you're already in iex, you can compile your file without returning to the command line using the `c` helper.

```
iex> c "times.exs"
[Times]
iex> Times.double(4)
8
iex> Times.double(123)
246
```

Prepared exclusively for Richard Bishop

The line c "times.exs" compiles your source file and loads it into iex. We then call the double function in the Times module a couple of times using Times.double.[1]

What happens if we make our function fail by passing it a string rather than a number?

```
iex> Times.double("cat")
** (ArithmeticError) bad argument in arithmetic expression
    times.exs:3: Times.double/1
```

An exception (ArithmeticError) gets raised, and we see a stack backtrace. The first line tells us what went wrong (we tried to perform arithmetic on a string, and the next line tells us where. But look at what it writes for the name of our function: Times.double/1.

In Elixir, a named function is identified by both its name and by the number of parameters it has (its *arity*). Our double function takes one parameter, so Elixir knows it as double/1. If we had another version of double that took three parameters, it would be known as double/3. There is no connection between these two functions—they are totally separate as far as Elixir is concerned. But, from a human perspective, you'd imagine that if two functions have the same name, they are somehow related, even if they have a different number of parameters. For that reason, don't use the same name for two functions that do unrelated things.

The Body of the Function is a Block

The do…end block is one way of grouping expressions and passing them to other code. They are used in module and named function definitions, control structures—just any place in Elixir where code needs to be handled as an entity.

However, do…end is not actually the underlying syntax. The actual syntax looks like this:

```
def double(n), do: n * 2
```

If you want to, you can pass multiple lines to do: by grouping them with parentheses.

```
def greet(greeting, name), do: (
  IO.puts greeting
```

1. Elixir compiles a source file called name.ex into a binary file called Elixir.Name.beam. BEAM is both a binary file format and the name of the Erlang interpreter implementation that reads it.

Prepared exclusively for Richard Bishop

```
  IO.puts "How're you doing, #{name}"
)
```

The do…end form is just a lump of syntactic sugar—during compilation it is turned into the do: form.[2] Typically people use the do: syntax for single line blocks, and do…end for multiline ones.

This means our times example would probably be written as

mm/times1.exs

```
defmodule Times do
  def double(n), do: n * 2
end
```

You could even write it as

```
defmodule Times, do: (def double(n), do: n*2)
```

(but, please, don't).

Your turn…

➤ *Exercise: ModulesAndFunctions-1*
Extend the Times module with a triple function, that multiplies its parameter by three.

➤ *Exercise: ModulesAndFunctions-2*
Run the result in iex. Use both techniques to compile the file.

➤ *Exercise: ModulesAndFunctions-3*
Add a quadruple function. (Maybe it could call the double function….)

Function Calls and Pattern Matching

In the previous chapter we saw that anonymous functions use pattern matching to bind their parameter list to the passed arguments. The same is true of named functions. The difference is that you write what look like multiple function definitions, each with its own parameter list and body.[3]

When you call a named function, Elixir tries to match your arguments with the parameter list of the first definition (clause). If it cannot match them, it tries the next definition of the same function (remember, this will have to

2. And the do: form itself is nothing special: it is simply a term in a keyword list.
3. Although these look like multiple function definitions, purists will tell you they are multiple clauses of the same definition.

Prepared exclusively for Richard Bishop

have the same arity), and checks to see if it matches. It continues until it runs out of candidates.

Let's play with this. The factorial of *n* (written *n!*) is the product of all numbers from 1 to *n*. By convention, 0! is 1.

Another way of expressing this is to say:

- factorial(0) → 1
- factorial(n) → n * factorial(n-1)

This is a specification of factorial, but it is also very close to an Elixir implementation:

mm/factorial1.exs

```
defmodule Factorial do
  def of(0), do: 1
  def of(n), do: n * of(n-1)
end
```

Here we have two definitions *of the same function*. If we call `Factorial.of(2)`, Elixir matches the 2 against the first function's parameter, 0. This fails, so it tries the second definition, which succeeds when Elixir binds 2 to `n`. It then evaluates the body of this function, which calls `Factorial.of(1)`. The same process applies, and the second definition is run. This in turn calls `Factorial.of(0)`. Now the first function definition matches. This function returns 1, and the recursion ends. Elixir now unwinds the stack, performing all the multiplications, and the answer is returned.[4]

Let's play with this code:

```
iex> c "factorial1.exs"
[Factorial]
iex> Factorial.of(3)
6
iex> Factorial.of(7)
5040
iex> Factorial.of(10)
3628800
iex> Factorial.of(1000)
40238726007709377354370243392300398571937486421071463254379991042993851239862
90205920442084869694048004799886101971960586316668729948085589013238296699445
...
00624271243416909004153690105933983835777939410970027753472000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
```

4. This implementation of factorial works, but it could be significantly improved. We'll do that improvement when we look at tail recursion on page 170.

Prepared exclusively for Richard Bishop

```
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

This pattern of design and coding is very common in Elixir (and almost all functional languages). First, look for the simplest possible case, one that has a definite answer. This will be the anchor. Then look for a recursive solution that will at some point end up calling the anchor case.

Here are a couple of examples:

Sum of the first n numbers

- the sum of the first 0 numbers is 0
- the sum of the numbers up to *n* is *n* + the sum of the numbers up to *n-1*

Length of a list

- the length of an empty list is 0
- the length of any other list is 1 + the length of the tail of that list

One point worth stressing—the order of these clauses can make a difference when you translate them into code. Elixir tries functions from the top down, executing the first match. So the following code will not work:

mm/factorial1-bad.exs

```
defmodule BadFactorial do
  def of(n), do: n * of(n-1)
  def of(0), do: 1
end
```

The first function definition will always match, and the second will never be called. But Elixir has you covered—when you try to compile this, you'll get a warning:

```
iex> c "factorial1-bad.exs"
.../factorial1-bad.ex:3: this clause cannot match because a previous clause at
                         line 2 always matches
```

One more thing. When you have multiple implementations of the same function, they should be adjacent in the source file.

Your turn...

➤ *Exercise: ModulesAndFunctions-4*

Implement and run a function `sum(n)` that uses recursion to calculate the sum of the integers from 1 to *n*. You'll need to write this function inside a module in a separate file. Then load up iex, compile that file, and try your function.

Prepared exclusively for Richard Bishop

➤ *Exercise: ModulesAndFunctions-5*

Write a function `gcd(x,y)` that finds the greatest common divisor between two nonnegative integers. Algebraically, *gcd(x,y)* is *x* if *y* is zero, *gcd(y, rem(x,y))* otherwise.

Guard Clauses

We've seen that pattern matching allows Elixir to decide which function to invoke based on the arguments passed. But what if we need to distinguish based on their types, or on some test involving their values? For this, you use *guard clauses*. These are predicates which are attached to a function definition using one or more `when` keywords. When doing pattern matching, Elixir first does the conventional parameter-based match, and then evaluates any `when` predicates, only executing the function if at least one returns true.

mm/guard.exs

```
defmodule Guard do

  def what_is(x) when is_number(x) do
    IO.puts "#{x} is a number"
  end

  def what_is(x) when is_list(x) do
    IO.puts "#{inspect(x)} is a list"
  end

  def what_is(x) when is_atom(x) do
    IO.puts "#{x} is an atom"
  end

end

Guard.what_is(99)        # => 99 is a number
Guard.what_is(:cat)      # => cat is an atom
Guard.what_is([1,2,3])   # => [1,2,3] is a list
```

Recall our previous factorial example on page 48.

mm/factorial1.exs

```
defmodule Factorial do
  def of(0), do: 1
  def of(n), do: n * of(n-1)
end
```

If we were to pass it a negative number, it would loop forever—no matter how many times you decrement `n`, it will never be zero. So it is a good idea to add a guard clause to stop this from happening:

Prepared exclusively for Richard Bishop

mm/factorial2.exs
```
defmodule Factorial do
  def of(0), do: 1
  def of(n) when n > 0 do
    n * of(n-1)
  end
end
```

If you run this code with a negative argument, none of the functions will match:

```
iex> c "factorial2.exs"
[Factorial]
iex> Factorial.of -100
** (FunctionClauseError) no function clause matching in Factorial.of/1...
```

Guard Clause Limitations

You can only write a subset of Elixir expressions in guard clauses. The following list comes from the Getting Started guide.[5]

comparison operators
: ==, !=, ===, !==, >, <, <=, >=

boolean and negation operators
: or, and, not, !. Note that || and && are not allowed

arithmetic operators
: +, -, *, /

join operators
: <> and ++ as long as the left side is a literal

the in operator
: membership in a collection

type check functions
: These built-in Erlang functions return true if their argument is a given type. You can find their documentation online.[6]

  is_atom is_binary is_bitstring is_boolean is_exception is_float is_function is_integer is_list is_map is_number is_pid is_port is_record is_reference is_tuple

other functions
: These built-in functions return values (not true or false). Their documentation is online on the same page as the type check functions.

5. http://elixir-lang.org/getting_started/5.html
6. http://erlang.org/doc/man/erlang.html#is_atom-1

Prepared exclusively for Richard Bishop

abs(number) bit_size(bitstring) byte_size(bitstring) div(number,number) elem(tuple, n) float(term) hd(list) length(list) node() node(pid|ref|port) rem(number,number) round(number) self() tl(list) trunc(number) tuple_size(tuple)

Default Parameters

When you define a named function, you can give default values to any of its parameters using the syntax param \\ value. When you call a function with defaulted parameters, Elixir compares the number of actual arguments you are passing with the number of required parameters for the function. If the count is less, then there's no match. If it is equal, then the required parameters take the values of the passed arguments, and the other parameters take their default values. If the count of passed arguments is greater than the number of required parameters, Elixir uses the excess to override the default values of some or all parameters. Parameters are matched left to right.

mm/default_params.exs

```
defmodule Example do

  def func(p1, p2 \\ 2, p3 \\ 3, p4) do
    IO.inspect [p1, p2, p3, p4]
  end

end

Example.func("a", "b")              # => ["a",2,3,"b"]
Example.func("a", "b", "c")         # => ["a","b",3,"c"]
Example.func("a", "b", "c", "d")    # => ["a","b","c","d"]
```

Default arguments can behave surprisingly when Elixir does pattern matching. For example, compile the following:

```
def func(p1, p2 \\ 2, p3 \\ 3, p4) do
  IO.inspect [p1, p2, p3, p4]
end

def func(p1, p2) do
  IO.inspect [p1, p2]
end
```

and you'll get the error

```
** (CompileError) default_params.exs:7: def func/2 conflicts with
                  defaults from def func/4
```

That's because the first function definition (with the default parameters) matches any call with 2, 3, or 4 arguments.

Prepared exclusively for Richard Bishop

There's one more thing with default parameters. Here's a function with multiple heads that also has a default parameter:

mm/default_params1.exs
```
def func(p1, p2 \\ 123) do
  IO.inspect [p1, p2]
end

def func(p1, 99) do
  IO.puts "you said 99"
end
```

If you compile this, you'll get the error:

```
** (CompileError) default_params1.exs.exs:8: def func/2 has default
                  values and multiple clauses, define a function head
                  with the defaults
```

The intent behind this is to reduce the confusion that can arise with defaults. Simply add a function head with no body that contains the default parameters, and use regular parameters for the rest. The defaults will apply to all calls to the function.

mm/default_params2.exs
```
def func(p1, p2 \\ 123)

def func(p1, 123) do
  IO.puts "You used the default"
end

def func(p1, 99) do
  IO.puts "you said 99"
end
```

Your turn…

➤ *Exercise: ModulesAndFunctions-6*

I'm thinking of a number between 1 and 1000…

The most efficient way to find the number is to guess halfway between the low and high numbers of the range. If our guess is too big, then the answer lies between the bottom of the range and one less than our guess. If it is too small, then the answer lies between one more than our guess and the end of the range.

Code this up. Your API will be `guess(actual, range)`, where `range` is an Elixir range.

Your output should look similar to:

Prepared exclusively for Richard Bishop

```
iex> Chop.guess(273, 1..1000)
Is it 500
Is it 250
Is it 375
Is it 312
Is it 281
Is it 265
Is it 273
273
```

Hints:

- You may need to implement helper functions with an additional parameter (the currently guessed number).
- the div(a,b) function performs integer division
- guard clauses are your friends
- patterns can match the low and high parts of a range (a..b=4..8)

Private Functions

The defp macro defines a private function—one that can only be called within the module that declares it.

You can define private functions with multiple heads, just as you can with def. However, you cannot have some heads private and others public. That is, the following code is not valid:

```
def fun(a) when is_list(a), do: true
defp fun(a), do: false
```

|> — The Amazing Pipe Operator

I've saved the best for last, at least when it comes to functions.

You've all seen code like this:

```
people = DB.find_customers
orders = Orders.for_customers(people)
tax    = sales_tax(orders, 2013)
filing = prepare_filing(tax)
```

Bread and butter programming. We do it, because the alternative was to write

```
filing = prepare_filing(sales_tax(Orders.for_customers(DB.find_customers), 2013))
```

and that's the kind of code that you use to get kids to eat their vegetables. Not only is it hard to read, but you have to read it inside-out if you want to see the order that things get done.

Prepared exclusively for Richard Bishop

Elixir has a better way of writing it.

```
filing = DB.find_customers
           |> Orders.for_customers
           |> sales_tax(2013)
           |> prepare_filing
```

The |> operator takes the result of the expression on its left and inserts it as the first parameter of the function invocation to its right. So the list of customers returned by the first call becomes the argument passed to the for_customers function. The resulting list of orders becomes the first argument to sales_tax, and the given parameter, 2013, becomes the second.

val |> f(a,b) is basically the same as calling f(val,a,b), and

```
list
|> sales_tax(2013)
|> prepare_filing
```

is the same as prepare_filing(sales_tax(list, 2013)).

In the previous example, I wrote each term in the expression on a separate line, and that's perfectly valid Elixir. But you can also chain terms on the same line.

```
iex> (1..10) |> Enum.map(&(&1*&1)) |> Enum.filter(&(&1 < 40))
[1, 4, 9, 16, 25, 36]
```

Note that I had to use parentheses in the above—the & shortcut and the pipe operator fight otherwise.

Let me repeat that—you should always use parentheses around function parameters in pipelines.

I think the key thing about the pipe operator is that it lets you write code that pretty much follows the form of your spec. For the sales tax example, you might have jotted on some paper

- get the customer list
- generate a list of their orders
- calculate tax on the orders
- prepare the filing

To take this from a napkin spec to running code, you just put |> between the items, and implement each as a function.

```
DB.find_customers
  |> Orders.for_customers
  |> sales_tax(2013)
  |> prepare_filing
```

Prepared exclusively for Richard Bishop

Programming is transforming data. And the |> operator makes that transformation explicit.

And now the subtitle of this book makes sense....

Modules

Modules provide namespaces for things you define. We've already seen them encapsulating named functions. They also act as wrappers for macros, structs, protocols, and other modules.

If you want to reference a function defined in a module from outside that module, you'll need to prefix that reference with the module's name. You don't need that prefix if code references something inside the same module as itself. This is shown in the following example.

```
defmodule Mod do
  def func1 do
    IO.puts "in func1"
  end
  def func2 do
    func1
    IO.puts "in func2"
  end
end

Mod.func1
Mod.func2
```

`func2` can call `func1` directly, because it is inside the same module. Outside the module, you have to use the fully qualified name, `Mod.func1`.

Just as you do in your favorite language, Elixir programmers use nested modules to impose structure for readability and reuse. After all, we are all library writers.

To access a function in a nested module from the outside scope, prefix it with all the module names. To access it within the containing module, you can either use the fully qualified name or just the inner module name as a prefix.

```
defmodule Outer do
  defmodule Inner do
    def inner_func do
    end
  end

  def outer_func do
    Inner.inner_func
  end
```

Prepared exclusively for Richard Bishop

```
end

Outer.outer_func
Outer.Inner.inner_func
```

In reality, module nesting in Elixir is an illusion—all modules are defined at the top level. When you define a module inside another, Elixir simply prepends the outer module's name to the inner module name, putting a dot between the two. This means you can directly define a nested module:

```
defmodule Mix.Tasks.Doctest do
  def run do
  end
end

Mix.Tasks.Doctest.run
```

It also means there's no particular relationship between the modules Mix and Mix.Tasks.Doctest.

Directives for Modules

Elixir has three directives that make working with modules easier. All three are executed as your program runs, and the effect of all three is *lexically scoped*—it starts at the point the directive is encountered, and ends at the end of the enclosing scope. This means that a directive in a module definition takes effect from the place you wrote it until the end of the module; a directive in a function definition runs to the end of the function.

The import Directive

The import directive brings the functions and/or macros of a module into the scope that uses it. If you use a particular module a lot in your code, import can cut down the clutter in your source files by eliminating the need to repeat the module name time and again.

For example, if you import the flatten function from the List module, you'd be able to call it in your code without having to specify the module name as a prefix.

mm/import.exs

```
defmodule Example do
  def func1 do
     List.flatten [1,[2,3],4]
  end
  def func2 do
    import List, only: [flatten: 1]
    flatten [5,[6,7],8]
```

Prepared exclusively for Richard Bishop

```
  end
end
```

The full syntax of import is

```
import Module [, only:|except: ]
```

The optional second parameter lets you control which functions or macros are imported. You write only: or except: followed by a list of name: arity pairs. It is a good idea to use import in the smallest possible enclosing scope, and to use only: to import just those functions you need.

```
import List, only: [ flatten: 1, duplicate: 2 ]
```

Alternatively, you can give only: one of the atoms :functions or :macros, and only those things will be imported.

The alias Directive

The alias directive creates an alias for a module. One obvious use is to cut down on typing.

```
defmodule Example do
  def func do
    alias Mix.Tasks.Doctest, as: Doctest
    doc = Doctest.setup
    doc.run(Doctest.defaults)
  end
end
```

We could have abbreviated this alias directive to alias Mix.Tasks.Doctest, because the as: parameter defaults to the last part of the module name.

The require Directive

You require a module if you want to use the macros defined in that module. It ensures that the given module is loaded before your code tries to use any of the macros it defines. We'll talk about require when we talk about macros on page 243.

Module Attributes

Elixir modules each have associated metadata. Each item of metadata is called an *attribute* of the module, and is identified by a name. Inside a module, you can access these attributes by prefixing the name with an at-sign (@).

You can give an attribute a value using the syntax

```
@name  value
```

Prepared exclusively for Richard Bishop

This only works at the top-level of a module—you cannot set an attribute value inside a function definition. You can, however, access attributes inside functions.

mm/attributes.exs
```
defmodule Example do

  @author  "Dave Thomas"

  def get_author do
    @author
  end

end

IO.puts "Example was written by #{Example.get_author}"
```

You can set the same attribute multiple times in a module. If you access that attribute in a named function in that module, the value you see will be the value in effect when the function is defined.

mm/attributes1.exs
```
defmodule Example do

  @attr "one"
  def first, do: @attr
  @attr "two"
  def second, do: @attr

end

IO.puts "#{Example.first}  #{Example.second}"    # => one  two
```

These attributes are not variables in the conventional sense. Use them for configuration and metadata only. (Many Elixir programmers use them where users of Java or Ruby might use constants.)

Module Names: Elixir, Erlang, and Atoms

When we write modules in Elixir, they have names such as String or PhotoAlbum. We call functions in them using calls such as String.length("abc").

What's actually happening here is subtle. Module names are internally just atoms. When you write a name starting with an upper case letter, like IO, Elixir converts it internally into an atom called Elixir.IO.

```
iex> is_atom IO
true
iex> to_string IO
```

Prepared exclusively for Richard Bishop

```
"Elixir.IO"
iex> :"Elixir.IO" === IO
true
```

So a call to a function in a module is really an atom followed by a dot followed by the function name. And, indeed, we can call functions like this:

```
iex> IO.puts 123
123
:ok
iex> :"Elixir.IO".puts 123
123
:ok
```

Calling a Function in an Erlang Library

The Erlang conventions for names are different—variables start with an uppercase letter, and atoms are simple lowercase names. So, for example, the Erlang module timer is called just that, the atom timer. In Elixir, we write that as :timer. If you want to refer to the tc function in timer, you'd write :timer.tc. (Note the colon at the start of :timer.)

Let's look at an example. Say we want to output a floating point number in a 3 character-wide field with one decimal place. Erlang has a function for this. A search for erlang format takes us to the description of the format function in the Erlang io module.[7]

Reading the description, we see that Erlang expects us to call io.format. So, in Elixir, we simply change the module name to an atom:

```
iex> :io.format("The number is ~3.1f~n", [5.678])
The number is 5.7
:ok
```

Finding Libraries

If you're looking for a library to use in your app, you'll want to look first for existing Elixir modules. The built-in ones are documented on the Elixir website,[8] and others are listed at expm.co and on GitHub (search for *elixir*).

If that fails, search for a built-in Erlang library,[9] or search the web. If you find something written in Erlang, you'll be able to use it in your project (we'll tell you how in the chapter on projects on page 133). But, be aware that the Erlang documentation for a library follows Erlang conventions. Variables start

7. http://erlang.org/doc/man/io.html#format-2>
8. http://elixir-lang.org/docs/
9. http://erlang.org/doc/ and http://erldocs.com/R15B/ (the latter is slightly out-of-date).

Prepared exclusively for Richard Bishop

with uppercase letters, and identifiers starting with a lowercase letter are atoms (so Erlang would say `tomato` and Elixir would say `:tomato`.) There is a summary of the differences between Elixir and Erlang online.[10]

A Tale of Two Libraries

Elixir runs on the Erlang VM, and is totally compatible with Erlang code. This means that code you write can call any of the thousands of Erlang functions, both built-in and in external libraries.

At the same time, Elixir also has a set of libraries unique to itself. Some are bundled with the base system, and others are available from places such as GitHub.

Which should you use? My advice is to use Elixir libraries where available, but don't be afraid to use the Erlang APIs, too. My suggestion is that when you *do* find yourself using Erlang APIs, wrap that usage in an Elixir module—Erlang APIs can be quite inconsistent in their conventions, and wrapping them gives you a chance to produce something a little more rational.

On a practical note, you call the 'map' function in the Elixir 'Enum' module using 'Enum.map(...)'. If instead you wanted to call the similar Erlang function, which resides in the Erlang 'lists' module, you'd write ':lists.map(...)'. Throughout Elixir, you access an Erlang module by prefixing its name with a colon.

Your turn…

➤ *Exercise: ModulesAndFunctions-7*

Find the library functions to do the following, and then use each in iex. (If there's the word Elixir or Erlang at the end of the challenge, then you'll find the answer in that set of libraries.)

- Convert a float to a string with 2 decimal digits. (Erlang)
- Get the value of an operating system environment variable. (Elixir)
- Return the extension component of a file name (so return `.exs` if given `"dave/test.exs"` (Elixir)
- Return the current working directory of the process. (Elixir)
- Convert a string containing JSON into Elixir data structures. (Just find, don't install)
- Execute a command in your operating system's shell

10. http://elixir-lang.org/crash-course.html

Prepared exclusively for Richard Bishop

In this chapter, we'll see
- The recursive structure of lists
- Traversing and building lists
- Accumulators
- Implementing map and reduce

CHAPTER 7

Lists and Recursion

When we program with lists in conventional languages, we treat them as things to be iterated—it seems natural to loop over them. So why do we have a chapter on *Lists and Recursion*? Because if you look at the problem in the right way, recursion is a perfect tool for processing lists.

Heads and Tails

Earlier we said *a list may either be empty or it consists of a head and a tail. The head contains a value and the tail is itself a list.*

This is a recursive definition. Let's imagine we could represent the split between the head and the tail using a pipe character, |. We'll represent the empty list as

```
[]
```

The single element list we normally write as [3] is actually the value 3 joined to the empty list:

```
[ 3  |  []  ]
```

When we see the pipe character, we say that what is on the left is the head of a list, and what's on the right is the tail.

So let's look at the list [2, 3]. The head is 2, and the tail is the single element list containing 3. And we know what that list looks like—it is our previous example. So we could write [2,3] as

```
[ 2  |  [ 3 | [] ] ]
```

At this point, part of your brain is telling you to go read today's XKCD—this list stuff can't be useful. Ignore that small voice, just for a second. We're about to do something magical. But before we do, let's add one more term, making

Prepared exclusively for Richard Bishop

our list [1, 2, 3]. This is the head 1 followed by the list [2, 3], which is what we just derived above:

```
[ 1  |  [ 2  |  [ 3 | [] ] ] ]
```

Now it turns out that this is valid Elixir syntax. Type it into iex:

```
iex> [ 1  |  [ 2  |  [ 3 | [] ] ] ]
[1, 2, 3]
```

And here's the magic. When we discussed pattern matching, we said that the pattern could be a list, and the values in that list would be assigned from the right-hand side:

```
iex> [a, b, c ] = [ 1, 2, 3 ]
[1, 2, 3]
iex> a
1
iex> b
2
iex> c
3
```

We can also use the pipe character in the pattern. What's to the left of it matches the head value of the list, and what's to the right matches the tail.

```
iex> [ head | tail ] = [ 1, 2, 3 ]
[1, 2, 3]
iex> head
1
iex> tail
[2, 3]
```

How Lists Are Displayed by iex

In Chapter 11, *Strings and Binaries*, on page 109, we'll see that Elixir has two representations for strings. One is the familar sequence of characters in consecutive memory locations. Literals written with double quotes use this form.

The second form, using single quotes, represents strings as a list of integer codepoints. So the string 'cat' is the three codepoints 99, 97, and 116.

This is a headache for iex. When it sees a list like '[99,97,116]' it doesn't know if it is supposed to be the string "cat" or a list of three numbers. So it uses a heuristic. If all the values in a list represent printable characters, it displays the list as a string, otherwise it displays a list of integers.

```
iex> [99, 97, 116]
'cat'
iex> [99, 97, 116, 0]   # '0' is nonprintable
[99, 97, 116, 0]
```

Prepared exclusively for Richard Bishop

In Chapter 11, *Strings and Binaries*, on page 109 we'll see how to bypass this behavior. In the meantime, don't be surprised if a string pops up when you were expecting a list.

Using Head and Tail to Process a List

Now we can split a list into its head and its tail, and we can construct a list from a value and a list, which become the head and tail of that new list.

So why talk about lists after we talk about modules and functions? Because lists and recursive functions go together like fish and chips. Let's look at finding the length of a list.

- The length of an empty list is 0
- The length of a list is 1 + the length of the tail of that list

Writing that in Elixir is easy:

lists/mylist.exs

```
defmodule MyList do
  def len([]), do: 0
  def len([head|tail]), do: 1 + len(tail)
end
```

The only tricky part is the definition of the second variant of the function:

```
def len([ head | tail ]) ...
```

This is a pattern match for any nonempty list. When it does match, the variable head will hold the value of the first element of the list, and tail will hold the rest of the list. (And remember that every list is terminated by an empty list, so the tail can be [].)

Let's see this at work with the list [11, 12, 13, 14, 15]. At each step, we take off the head, and add one to the length of the tail:

```
len([11,12,13,14,15])
= 1 + len([12,13,14,15])
= 1 + 1 + len([13,14,15])
= 1 + 1 + 1 + len([14,15])
= 1 + 1 + 1 + 1 + len([15])
= 1 + 1 + 1 + 1 + 1 + len([])
= 1 + 1 + 1 + 1 + 1 + 0
= 5
```

Let's try our code to see if theory works in practice:

Prepared exclusively for Richard Bishop

```
iex> c "mylist.exs"
...mylist.exs:3: variable head is unused
[MyList]
iex> MyList.len([])
0
iex> MyList.len([11,12,13,14,15])
5
```

It works, but we have a compilation warning—we never used the variable head in the body of our function. We can fix that, and make our code more explicit, using the special variable _ (underscore), which basically acts as a placeholder. We can also use an underscore in front of any variable name to turn off the warning if that variable isn't used. I sometimes like to do this to document what the unused parameter is.

lists/mylist1.exs

```
defmodule MyList do
  def len([]),             do: 0
  def len([_head | tail]), do: 1 + len(tail)
end
```

When we compile, the warning has gone.[1]

```
iex> c "mylist1.exs"
[MyList]
iex> MyList.len([1,2,3,4,5])
5
iex> MyList.len(["cat", "dog"])
2
```

Using Head and Tail to Build a List

Let's get more ambitious. Let's write a function that takes a list of numbers and returns a new list containing the square of each. We don't show it, but these definitions are inside the MyList module.

lists/mylist1.exs

```
def square([]),              do: []
def square([ head | tail ]), do: [ head*head | square(tail) ]
```

There's a lot going on here. First, look at the parameter patterns for the two definitions of square. The first matches an empty list and the second matches all other lists.

Second, look at the body of the second definition:

```
def square([ head | tail ]), do: [ head*head | square(tail) ]
```

1. If you compile the second version of MyList, you may get a warning about "redefining module MyList." This is just Elixir being cautious.

Prepared exclusively for Richard Bishop

When we match a nonempty list, we return a new list whose head is the square of the original list's head, and whose tail is list of squares of the tail. This is the recursive step.

Let's try it:

```
iex> c "mylist1.exs"
[MyList]
iex> MyList.square []        # this calls the 1st definition
[]
iex> MyList.square [4,5,6]   # and this calls the 2nd
[16, 25, 36]
```

Let's do something similar—a function that adds one to every element in the list:

lists/mylist1.exs

```
def add_1([]),              do: []
def add_1([ head | tail ]), do: [ head+1 | add_1(tail) ]
```

And call it:

```
iex> c "mylist1.exs"
[MyList]
iex> MyList.add_1 [1000]
[1001]
iex> MyList.add_1 [4,6,8]
[5, 7, 9]
```

Creation of a Map Function

With both `square` and `add_1`, all the work is done in the second function definition. And that definition looks about the same for each—it returns a new list whose head is the result of either squaring or incrementing the head of its argument, and whose tail is the result of calling itself recursively on the tail of the argument. So let's generalize this. We'll define a function called `map` that takes a list and a function, and returns a new list containing the result of applying that function to each element in the original.

lists/mylist1.exs

```
def map([], _func),             do: []
def map([ head | tail ], func), do: [ func.(head) | map(tail, func) ]
```

The `map` function is pretty much identical to the `square` and `add_1` functions. It returns an empty list if passed an empty list, otherwise it returns a list where the head is the result of calling the passed-in function and the tail is a recursive call to itself. Note that in the case of an empty list, we use `_func` as

Prepared exclusively for Richard Bishop

the second parameter. The underscore prevents Elixir warning us about an unused variable.

To call this function, pass in a list and a function (defined using fn).

```
iex> c "mylist1.exs"
[MyList]
iex> MyList.map [1,2,3,4], fn (n) -> n*n end
[1, 4, 9, 16]
```

A function is just a built-in type, defined between fn and the end. Here we pass a function as the second argument (func) to map. This is invoked inside map using func.(head), which squares the value in head, using the result to build the new list.

We can call map with a different function:

```
iex> MyList.map [1,2,3,4], fn (n) -> n+1 end
[2, 3, 4, 5]
```

and another

```
iex> MyList.map [1,2,3,4], fn (n) -> n > 2 end
[false, false, true, true]
```

And we can do the same using the & shortcut notation.

```
iex> MyList.map [1,2,3,4], &(&1 + 1)
[2, 3, 4, 5]
iex> MyList.map [1,2,3,4], &(&1 > 2)
[false, false, true, true]
```

Keeping Track of Values During Recursion

So far you've seen how to process each element in a list, but what if we want to sum all of the elements. The difference here is that we need to remember the partial sum as we process each element in turn.

In terms of a recursive structure, it's easy:

- sum([]) → 0
- sum([head |tail]) → «*total*» + sum(tail)

But the basic scheme gives us nowhere to record the total as we go along. Remember that one of our goals is to have immutable state, so we can't keep the value in a global or module-local variable.

But we *can* pass the state in a function's parameter.

lists/sum.exs

```
defmodule MyList do
```

Prepared exclusively for Richard Bishop

```
  def sum([], total),              do: total
  def sum([ head | tail ], total), do: sum(tail, head+total)
end
```

Our sum function now has two parameters, the list and the total so far. In the recursive call, we pass it the tail of the list, and increment the total by the value of the head.[2]

When we call sum we have to remember to pass both the list and the initial total value (which will be 0):

```
iex> c "sum.exs"
[MyList]
iex> MyList.sum([1,2,3,4,5], 0)
15
iex> MyList.sum([11,12,13,14,15], 0)
65
```

Having to remember that extra zero is a little tacky, so the convention in Elixir is to hide it—our module has a public function that takes just a list, and it calls private functions to do the work.

lists/sum2.exs

```
defmodule MyList do

  def sum(list), do: _sum(list, 0)

  # private methods
  defp _sum([], total),              do: total
  defp _sum([ head | tail ], total), do: _sum(tail, head+total)
end
```

Two things to notice here. First, we use defp to define a private function. You won't be able to call these functions outside the module.

Second, we chose to give our helper functions the same name as our public function, but with a leading underscore. Elixir treats them as being independent of each other, but a human reader can see that they are clearly related.[3]

Your turn...

2. There's a technique I find useful when thinking about these types of functions. At all times, they maintain an *invariant*, a condition that is true on return from any call (or nested call). In this case, the invariant is that, at any stage of the recursion, the sum of the elements in the list parameter plus the current total will equal the total of the entire list. Thus, when the list becomes empty, the total will be the value we want.
3. As they have a different arity from the original sum, Elixir would treat them as different functions even if the name was the same. The leading underscore simply makes it explicit. You'll find some library code also uses do_*xxx* for these helpers.

Prepared exclusively for Richard Bishop

➤ *Exercise: ListsAndRecursion-0*

I defined our sum function to carry a partial total around as a second parameter. I did this so I could illustrate how to use accumulators to build values. The sum function can also be written without an accumulator. Can you do it?

Generalizing Our Sum Function

The sum function takes a collection and reduces it to a single value. Clearly there are other functions that need to do something similar—return the greatest/least value, the product of the elements, a string containing the elements with spaces between them, and so on. So how could we write a general purpose function that reduces a collection to a value?

We know it has to take a collection. We also know we need to pass in some initial value (just like our sum/1 function passed a 0 as an initial value to its helper). And we also need to pass in a function that takes the current value of the reduction along with the next element of the collection, and returns the next value of the reduction. So it looks like our reduce function will be called with three arguments:

```
reduce(collection, initial_value, fun)
```

Now let's think about the recursive design:

- reduce([], value, _) → value
- reduce([head |tail], value, fun) → reduce(tail, fun.(head, value), fun)

So reduce applies the function to the head of the list and the current value, and passes the result as the new current value when reducing the tail of the list.

Here's our code for reduce. See how closely it follows the design.

lists/reduce.exs

```
defmodule MyList do
  def reduce([], value, _) do
    value
  end
  def reduce([head | tail], value, func) do
    reduce(tail, func.(head, value), func)
  end
end
```

And, again, we can use the shorthand notation to pass in the function:

```
iex> c "reduce.exs"
[MyList]
```

Prepared exclusively for Richard Bishop

```
iex> MyList.reduce([1,2,3,4,5], 0, &(&1 + &2))
15
iex> MyList.reduce([1,2,3,4,5], 1, &(&1 * &2))
120
```

Your turn…

➤ *Exercise: ListsAndRecursion-1*

Write a function mapsum that takes a list and a function. It applies the function to each element of the list, and then sums the result, so

```
iex> MyList.mapsum [1, 2, 3], &(&1 * &1)
14
```

➤ *Exercise: ListsAndRecursion-2*

Write max(list) that returns the element with the maximum value in the list. (This is slightly trickier than it sounds.)

➤ *Exercise: ListsAndRecursion-3*

An Elixir single quoted string is actually a list of individual character codes. Write a function caesar(list, n) that adds n to each element of the list, wrapping if the addition results in a character greater than *z*.

```
iex> MyList.caesar('ryvkve', 13)
?????? :)
```

➤ *Exercise: ListsAndRecursion-3a*

Now report yourself to your national antiterror authority for creating encryption code. SHAME ON YOU.

More Complex List Patterns

Not every list problem can be easily solved by processing one element at a time. Fortunately, the join operator, |, supports multiple values to its left. Thus you could write

```
iex> [ 1, 2, 3 | [ 4, 5, 6 ]]
[1, 2, 3, 4, 5, 6]
```

The same thing works in patterns, so you can match multiple individual elements as the head. For example, the following program swaps pairs of values in a list.

lists/swap.exs

```
defmodule Swapper do

  def swap([]), do: []
```

Prepared exclusively for Richard Bishop

```
  def swap([ a, b | tail ]), do: [ b, a | swap(tail) ]
  def swap([_]), do: raise "Can't swap a list with an odd number of elements"

end
```

We can play with it in iex:

```
iex> c "swap.exs"
[Swapper]
iex> Swapper.swap [1,2,3,4,5,6]
[2, 1, 4, 3, 6, 5]
iex> Swapper.swap [1,2,3,4,5,6,7]
** (RuntimeError) Can't swap a list with an odd number of elements
```

The third definition of `swap` matches a list with a single element. This will happen if we get to the end of the recursion and only have one element left. As we take two values off the list on each cycle, the initial list must have had an odd number of elements.

Lists of Lists

Let's imagine we had recorded temperatures and rainfall at a number of weather stations. Each reading looks like

```
[ timestamp, location_id, temperature, rainfall ]
```

Our code is passed a list containing a number of these readings, and we want to report on the conditions for one particular location, number 27.

lists/weather.exs

```
defmodule WeatherHistory do

  def for_location_27([]), do: []
  def for_location_27([ [time, 27, temp, rain ] | tail]) do
    [ [time, 27, temp, rain] | for_location_27(tail) ]
  end
  def for_location_27([ _ | tail]), do: for_location_27(tail)

end
```

This is a standard *recurse until the list is empty* stanza. But look at the second definition of our function.[4] Where we'd normally match into a variable called `head`, here the pattern is

```
for_location_27([ [ time, 27, temp, rain ] | tail])
```

4. Technically there's only one function definition here. It has three *clauses*. But given that each clause starts with a `def` keyword, I think we can bend the terminology a little.

Prepared exclusively for Richard Bishop

For this to match, the head of the list must itself be a 4 element list, and the second element of this sublist must be 27. This function will only execute for entries from the desired location. But, when we do this kind of filtering, we also have to remember to deal with the case when our function doesn't match. That's what the third line does. We could have written

```
for_location_27([ [ time, _, temp, rain ] | tail])
```

but in reality we don't care *what* is in the head at this point.

In the same module we define some simple test data:

lists/weather.exs

```
def test_data do
  [
   [1366225622, 26, 15, 0.125],
   [1366225622, 27, 15, 0.45],
   [1366225622, 28, 21, 0.25],
   [1366229222, 26, 19, 0.081],
   [1366229222, 27, 17, 0.468],
   [1366229222, 28, 15, 0.60],
   [1366232822, 26, 22, 0.095],
   [1366232822, 27, 21, 0.05],
   [1366232822, 28, 24, 0.03],
   [1366236422, 26, 17, 0.025]
  ]
end
```

We can use that to play with our function in iex:[5]

```
iex> c "weather.exs"
[WeatherHistory]
iex> import WeatherHistory
nil
iex> for_location_27(test_data)
[[1366225622, 27, 15, 0.45], [1366229222, 27, 17, 0.468],
 [1366232822, 27, 21, 0.05]]
```

Our function is specific to a particular location, which is pretty limiting. We'd like to be able to pass in the location as a parameter. We can use pattern matching for this.

lists/weather2.exs

```
defmodule WeatherHistory do

  def for_location([], _target_loc), do: []
```

5. In this example we use the import function. This adds the functions in WeatherHistory to our local name scope. After calling import we don't have to put the module name in front of every function call.

Prepared exclusively for Richard Bishop

```
➤  def for_location([ [time, target_loc, temp, rain ] | tail], target_loc) do
     [ [time, target_loc, temp, rain] | for_location(tail, target_loc) ]
   end

   def for_location([ _ | tail], target_loc), do: for_location(tail, target_loc)

end
```

Now the second function only fires when the location extracted from the list head equals the target location passed as a parameter.

But we can improve on this. Our filter doesn't care about the other three fields in the head—it just needs the location. But we do need the value of the head itself to create the output list. Fortunately, Elixir pattern matching is recursive, and we can match patterns inside patterns.

lists/weather3.exs

```
defmodule WeatherHistory do

  def for_location([], target_loc), do: []

➤  def for_location([ head = [_, target_loc, _, _ ] | tail], target_loc) do
     [ head | for_location(tail, target_loc) ]
   end

   def for_location([ _ | tail], target_loc), do: for_location(tail, target_loc)

end
```

The key change here is this line:

```
def for_location([ head = [_, target_loc, _, _ ] | tail], target_loc)
```

Compare that with the previous version.

```
def for_location([ [ time, target_loc, temp, rain ] | tail], target_loc)
```

In the new version, we use placeholders for the fields we don't care about. But we also match the entire 4-element array into the parameter `head`. It's as if we said "match the head of the list where the second element is matched to `target_loc` and then match that whole head with the variable `head`". We've extracted an individual component of the sublist as well as the entire sublist.

In the original body of `for_location`, we generated our result list using the individual fields:

```
def for_location([ [ time, target_loc, temp, rain ] | tail], target_loc)
  [ [ time, target_loc, temp, rain ] | for_location(tail, target_loc) ]
end
```

Prepared exclusively for Richard Bishop

In the new version, we can just use the head, making it a lot clearer.

```
def for_location([ head = [_, target_loc, _, _ ] | tail], target_loc)  do
  [ head | for_location(tail, target_loc) ]
end
```

Your turn…

➤ *Exercise: ListsAndRecursion-4*

Write a function MyList.span(from, to) that returns a list of the numbers from from up to to.

The List Module in Action

The List module provides a set of functions that operate on lists.

```
#
# Concatenate lists
#
iex> [1,2,3] ++ [4,5,6]
[1, 2, 3, 4, 5, 6]
#
# Flatten
#
iex> List.flatten([[[1], 2], [[[3]]]])
[1, 2, 3]
#
# Folding (like reduce, but can choose direction)
#
iex> List.foldl([1,2,3], "", fn value, acc -> "#{value}(#{acc})" end)
"3(2(1()))"
iex> List.foldr([1,2,3], "", fn value, acc -> "#{value}(#{acc})" end)
"1(2(3()))"
#
# Merging lists and splitting them apart
#
iex> l = List.zip([[1,2,3], [:a,:b,:c], ["cat", "dog"]])
[{1, :a, "cat"}, {2, :b, "dog"}]
iex> List.unzip(l)
[[1, 2], [:a, :b], ["cat", "dog"]]
#
# Accessing tuples within lists
#
iex> kw = [{:name, "Dave"}, {:likes, "Programming"}, {:where, "Dallas", "TX"}]
[{:name, "Dave"}, {:likes, "Programming"}, {:where, "Dallas", "TX"}]
iex> List.keyfind(kw, "Dallas", 1)
{:where, "Dallas", "TX"}
iex> List.keyfind(kw, "TX", 2)
{:where, "Dallas", "TX"}
```

Prepared exclusively for Richard Bishop

```
iex> List.keyfind(kw, "TX", 1)
nil
iex> List.keyfind(kw, "TX", 1, "No city called TX")
"No city called TX"
iex> kw = List.keydelete(kw, "TX", 2)
[name: "Dave", likes: "Programming"]
iex> kw = List.keyreplace(kw, :name, 0, {:first_name, "Dave"})
[first_name: "Dave", likes: "Programming"]
```

Prepared exclusively for Richard Bishop

In this chapter, we'll see
- The Two-and-a-half Dictionary Data Types
- The General Dictionary API
- Pattern matching and updating maps
- Structs
- Nested Data Structures

CHAPTER 8

Dictionaries: Maps, HashDicts, Keyword, Sets, and Structs

A dictionary is a data type that associates keys with values.

We've already looked briefly at a couple of dictionary types, maps and keyword lists. In this short chapter we'll see how to use them with pattern matching, and how to update them. We'll also look at `HashDict`, another implementation of dictionaries. Finally we'll look at the `Keyword` module, which implements a specialized dictionary intended for storing function and program options, and the `Set` module, that implements sets.

First, though, let's answer a common question—how do we choose an appropriate dictionary type for a particular need?

How to Choose Between Maps, HashDicts, and Keywords

Ask yourself these questions (and in this order):

- Will I want more than one entry with the same key?

  If so, you'll have to use the `Keyword` module.

- Do I need to guarantee the elements are ordered?

  Again, use the `Keyword` module.

- Do I want to pattern match against the contents (for example matching a dictionary that has a key of `:name` somewhere in it)?

  If so, use a map.

- Will I be storing more than a few hundred entries in it?

Prepared exclusively for Richard Bishop

If so, use a HashDict. With R17 of the BEAM virtual machine (which runs Erlang), maps are slow, particularly when adding new items.[1] This should improve in R18.

Dictionaries

Maps and HashDicts both implement the Dict behaviour. The Keyword module largely does, too, but with some differences to allow for the fact that it supports duplicate keys.

In general, you'll want to use the Dict module's methods to access this functionality, as this allows you the flexibility to change the underlying implementation between (say) a map and a HashDict.[2]

maps/use_dict.exs

```
defmodule Sum do
  def values(dict) do
    dict |> Dict.values |> Enum.sum
  end
end

# Sum a HashDict
hd = [ one: 1, two: 2, three: 3 ] |> Enum.into HashDict.new
IO.puts Sum.values(hd)  # => 6

# Sum a Map
map = %{ four: 4, five: 5, six: 6 }
IO.puts Sum.values(map) # => 15
```

Let's play with some of the Dict API:

```
iex> kw_list = [name: "Dave", likes: "Programming", where: "Dallas"]
[name: "Dave", likes: "Programming", where: "Dallas"]
iex> hashdict = Enum.into kw_list, HashDict.new
#HashDict<[name: "Dave", where: "Dallas", likes: "Programming"]>
iex> map = Enum.into kw_list, Map.new
%{likes: "Programming", name: "Dave", where: "Dallas"}
iex> kw_list[:name]
"Dave"
iex> hashdict[:likes]
"Programming"
iex> map[:where]
"Dallas"
iex> hashdict = Dict.drop(hashdict, [:where, :likes])
#HashDict<[name: "Dave"]>
```

1. https://github.com/pragdave/map_performance
2. This example uses Enum.into, which is how you can easily map one kind of collection into another. We'll look at this in the next chapter.

Prepared exclusively for Richard Bishop

```
iex> hashdict = Dict.put(hashdict, :also_likes, "Ruby")
#HashDict<[name: "Dave", also_likes: "Ruby"]>
iex> combo = Dict.merge(map, hashdict)
%{also_likes: "Ruby", likes: "Programming", name: "Dave", where: "Dallas"}
```

Keyword lists allow duplicate values, but you have to use the Keyword module to access them:

```
iex(51)> kw_list = [name: "Dave", likes: "Programming", likes: "Elixir"]
[name: "Dave", likes: "Programming", likes: "Elixir"]
iex(53)> kw_list[:likes]
"Programming"
iex(54)> Dict.get(kw_list, :likes)
"Programming"
iex(55)> Keyword.get_values(kw_list, :likes)
["Programming", "Elixir"]
```

As usual, the full API documentation is available online.[3]

Pattern Matching and Updating Maps

The question we most often ask of our maps is "do you have the following keys (and maybe values):"

- is there an entry with the key :name?
- are there entries for the keys :name and :height?
- does the entry with key :name have the value "Dave"?

Here's how we ask these questions using Elixir patterns.

```
iex> person = %{ name: "Dave", height: 1.88 }
%{height: 1.88, name: "Dave"}
iex> %{ name: a_name } = person
%{height: 1.88, name: "Dave"}
iex> a_name
"Dave"
iex> %{ name: _, height: _ } = person
%{height: 1.88, name: "Dave"}
iex> %{ name: "Dave" } = person
%{height: 1.88, name: "Dave"}
```

Our map does not have the key :weight, so the following pattern match fails:

```
iex> %{ name: _, weight: _ } = person
** (MatchError) no match of right hand side value: %{height: 1.88, name: "Dave"}
```

It's worth noting how the first pattern match destructured the map, extracting the value associated with the key :name. We can use this in many ways. Here's

3. http://elixir-lang.org/docs/

Prepared exclusively for Richard Bishop

one example—we can use destructuring in a list comprehension to give us a simple query capability.

maps/query.exs
```
people = [
  %{ name: "Grumpy",    height: 1.24 },
  %{ name: "Dave",      height: 1.88 },
  %{ name: "Dopey",     height: 1.32 },
  %{ name: "Shaquille", height: 2.16 },
  %{ name: "Sneezy",    height: 1.28 }
]

for person = %{ height: height } <- people,
    height > 1.5,
    do: IO.inspect person
```

This produces:

```
%{height: 1.88, name: "Dave"}
%{height: 2.16, name: "Shaquille"}
```

In this code, we feed a list of maps to our comprehension. The generator clause binds each map (as a whole) to `person`, and also binds the height from that map to `height`. The filter selects only those maps where the height exceeds 1.5, and the `do` block prints the whole map.

Clearly pattern matching is just pattern matching, so this capability of maps works equally well in `cond` expressions, function head matching, and any other time patterns are used.

maps/book_room.exs
```
defmodule HotelRoom do

  def book(%{name: name, height: height})
  when height > 1.9 do
    IO.puts "Need extra long bed for #{name}"
  end

  def book(%{name: name, height: height})
  when height < 1.3 do
    IO.puts "Need low shower controls for #{name}"
  end

  def book(person) do
    IO.puts "Need regular bed for #{person.name}"
  end

end

people |> Enum.each(&HotelRoom.book/1)
```

Prepared exclusively for Richard Bishop

```
#=> Need low shower controls for Grumpy
#   Need regular bed for Dave
#   Need regular bed for Dopey
#   Need extra long bed for Shaquille
#   Need low shower controls for Sneezy
```

A Limitation

Maps do not allow you to bind a value to a key during pattern matching. Thus, you can write this:

```
iex> %{ 2 => state } = %{ 1 => :ok, 2 => :error }
%{1 => :ok, 2 => :error}
iex> state
:error
```

but not this:

```
iex> %{ item => :ok } = %{ 1 => :ok, 2 => :error }
** (CompileError) iex:5: illegal use of variable item in map key
```

Updating a Map

In the previous chapter we saw how lists are updated through a combination of copying and changing the head.

With maps, we can add new key/value entries and update existing entries without traversing the whole structure. But, as with all values in Elixir, a map is immutable, and so the result of the update is a new map.

The simplest way to update a map is with this syntax:

```
new_map = %{ old_map | key => value, … }
```

This creates a new map which is a copy of the old, but with the values associated with the keys on the right of the pipe character are updated:

```
iex> m = %{ a: 1, b: 2, c: 3 }
%{a: 1, b: 2, c: 3}
iex> m1 = %{ m | b: "two", c: "three" }
%{a: 1, b: "two", c: "three"}
iex> m2 = %{ m1 | a: "one" }
%{a: "one", b: "two", c: "three"}
```

However, this syntax will not add a new key to a map. To do this, you have to use the Dict.put_new/3 function

Prepared exclusively for Richard Bishop

Maps and Structs

When Elixir sees %{ ... } it knows it is looking at a map. But it doesn't know much more. In particular, it doesn't know what you intend to do with it, whether only certain keys are allowed, and if some keys should have default values.

That's fine for anonymous maps. But what if you want to create a typed-map—a map with a fixed set of fields, with default values for those fields, and which you can pattern match by type as well as content.

Enter the *struct*.

A struct is simply a module that wraps a limited form of map. It's limited because the keys must be atoms, and because these maps don't have dict or access capabilities.

The name of the module becomes the name of the type of the map. Inside the module, you use the defstruct macro to define the characteristics of the map.

maps/defstruct.exs

```
defmodule Subscriber do
  defstruct name: "", paid: false, over_18: true
end
```

Let's play with this in iex:

```
$ iex defstruct.exs
iex> s1 = %Subscriber{}
%Subscriber{name: "", over_18: true, paid: false}
iex> s2 = %Subscriber{ name: "Dave" }
%Subscriber{name: "Dave", over_18: true, paid: false}
iex> s3 = %Subscriber{ name: "Mary", paid: true }
%Subscriber{name: "Mary", over_18: true, paid: true}
```

The syntax for creating a struct is the same as the syntax for creating a map—you simply add the module name between the % and the {.

You access the fields in a struct using the dot notation, or with pattern matching:

```
iex> s3.name
"Mary"
iex> %Subscriber{name: a_name} = s3
%Subscriber{name: "Mary", over_18: true, paid: true}
iex> a_name
"Mary"
```

And updates follow suite:

```
iex> s4 = %Subscriber{ s3 | name: "Marie"}
```

Prepared exclusively for Richard Bishop

```
%Subscriber{name: "Marie", over_18: true, paid: true}
```

Why are structs wrapped in a module? The idea is that you are likely to want to add struct-specific behaviour.

maps/defstruct1.exs

```
defmodule Attendee do
  defstruct name: "", paid: false, over_18: true

  def may_attend_after_party(attendee = %Attendee{}) do
    attendee.paid && attendee.over_18
  end

  def print_vip_badge(%Attendee{name: name}) when name !=  "" do
    IO.puts "Very cheap badge for #{name}"
  end

  def print_vip_badge(%Attendee{}) do
    raise "missing name for badge"
  end
end
```

```
$ iex defstruct1.exs
iex> a1 = %Attendee{name: "Dave", over_18: true}
%Attendee{name: "Dave", over_18: true, paid: false}
iex> Attendee.may_attend_after_party(a1)
false
iex> a2 = %Attendee{a1 | paid: true}
%Attendee{name: "Dave", over_18: true, paid: true}
iex> Attendee.may_attend_after_party(a2)
true
iex(22)> Attendee.print_vip_badge(a2)
Very cheap badge for Dave
:ok
iex(23)> a3 = %Attendee{}
%Attendee{name: "", over_18: true, paid: false}
iex(26)> Attendee.print_vip_badge(a3)
** (RuntimeError) missing name for badge
     defstruct1.exs:13: Attendee.print_vip_badge/1
```

Structs also play a large role when implementing protocols on page 259.

Another Way to Access Structs

The previous examples accessed struct attributes using the dot notation. This might have surprised you, as structs and maps clearly have a lot in common, and you access maps using `some_map[:name]`.

The reason is that maps implement the Access protocol (which defines the ability to access fields using square bracket notation) and structs do not.

Prepared exclusively for Richard Bishop

However, you can easily add this ability to your structs—simply using the @derive directive:

maps/derive.exs

```
defmodule Attendee do
  @derive Access
  defstruct name: "", over_18: false
end

iex> a = %Attendee{name: "Sally", over_18: true}
%Attendee{name: "Sally", over_18: true}
iex> a[:name]
"Sally"
iex> a[:over_18]
true
iex> a.name
"Sally"
```

Nested Dictionary Structures

The various dictionary types let us associate keys with values. But those values can themselves be dictionaries. For example, we may have a bug reporting system, where bugs are reported by customers. We could represent this using the following:

maps/nested.exs

```
defmodule Customer do
  defstruct name: "", company: ""
end

defmodule BugReport do
  defstruct owner: %{}, details: "", severity: 1
end
```

Let's create a simple report:

```
iex> report = %BugReport{owner: %Customer{name: "Dave", company: "Pragmatic"},
...>                     details: "broken"}
%BugReport{details: "broken",
           owner: %Customer{company: "Pragmatic", name: "Dave"},
           severity: 1}
```

The owner attribute of the report is itself a Customer struct.

We can access nested fields using the regular dot notation:

```
iex> report.owner.company
"Pragmatic"
```

But now our customer complains the company name is incorrect—it should be "PragProg". Let's fix it:

Prepared exclusively for Richard Bishop

```
iex> report = %BugReport{ report | owner:
...>                         %Customer{ report.owner | company: "PragProg" }}
%BugReport{details: "broken",
 owner: %Customer{company: "PragProg", name: "Dave"},
 severity: 1}
```

That's pretty ugly—we had to update the owner attribute of the overall bug report with an updated customer structure. This is verbose, hard to read, and error prone.

Fortunately, Elixir has a set of nested dictionary access functions. One of these, put_in, lets us set a value in a nested structure:

```
iex> put_in(report.owner.company, "PragProg")
%BugReport{details: "broken",
 owner: %Customer{company: "PragProg", name: "Dave"},
 severity: 1}
```

This isn't magic—it's simply a macro that generates the long-winded code we'd have to have written otherwise.

The update_in function lets us apply a function to a value in a structure.

```
iex> update_in(report.owner.name, &("Mr. " <> &1))
%BugReport{details: "broken",
  owner: %Customer{company: "PragProg", name: "Mr. Dave"},
   severity: 1}
```

The other two nested access functions are get_in and get_and_update_in The documentation in iex contains everything you need for these. However, there's one cool trick that both these functions support: rathe

Nested Accessors and Nonstructs

The nested accessor functions use the Access protocol to strip apart and reassemble data structures. This means that if you are using maps or keyword lists, you can supply the keys as symbols:

```
iex> report = %{ owner: %{ name: "Dave", company: "Pragmatic" }, severity: 1}
%{owner: %{company: "Pragmatic", name: "Dave"}, severity: 1}
iex> put_in(report[:owner][:company], "PragProg")
%{owner: %{company: "PragProg", name: "Dave"}, severity: 1}
iex> update_in(report[:owner][:name], &("Mr. " <> &1))
%{owner: %{company: "Pragmatic", name: "Mr. Dave"}, severity: 1}
```

Dynamic (Runtime) Nested Accessors

The nested accessors we've seen so far are macros—they operate at compile time. As a result, they have limitations, including:

- the number of keys you pass a particular call is static, and

Prepared exclusively for Richard Bishop

- you can't pass the set of keys as parameters between functions.

These are a natural consequence of the way that the macros bake their parameters into code at compile time.

To overcome this, get_in, put_in, update_in, and get_and_update_in can all take a list of keys as a separate parameter. Adding this parameter changes them from macros to function calls, and so they become dynamic.

	Macro	Function
get_in	*no*	(dict, keys)
put_in	(path, value)	(dict, keys, value)
update_in	(path, fn)	(dict, keys, fn)
get_and_update_in	(path, fn)	(dict, keys, fn)

Here's a simple example:

maps/dynamic_nested.exs

```
nested = %{
    buttercup: %{
      actor: %{
        first: "Robin",
        last:  "Wright"
      },
      role: "princess"
    },
    westley: %{
      actor: %{
        first: "Carey",
        last:  "Ewes"     # typo!
      },
      role: "farm boy"
    }
}

IO.inspect get_in(nested, [:buttercup])
# => %{actor: %{first: "Robin", last: "Wright"}, role: "princess"}

IO.inspect get_in(nested, [:buttercup, :actor])
# => %{first: "Robin", last: "Wright"}

IO.inspect get_in(nested, [:buttercup, :actor, :first])
# => "Robin"

IO.inspect put_in(nested, [:westley, :actor, :last], "Elwes")
# => %{buttercup: %{actor: %{first: "Robin", last: "Wright"}, role: "princess"},
# =>     westley: %{actor: %{first: "Carey", last: "Elwes"}, role: "farm boy"}}
```

Prepared exclusively for Richard Bishop

There's a cool trick that the dynamic versions of both get_in and get_and_update_in support—if you pass a function as a key, that function is invoked to return the corresponding values.

maps/get_in_func.exs

```
authors = [
  %{ name: "José",  language: "Elixir" },
  %{ name: "Matz",  language: "Ruby" },
  %{ name: "Larry", language: "Perl" }
]

languages_with_an_r = fn (:get, collection, next_fn) ->
   for row <- collection do
     if String.contains?(row.language, "r") do
       next_fn.(row)
     end
   end
end

IO.inspect get_in(authors, [languages_with_an_r, :name])
#=> [ "José", nil, "Larry" ]
```

Sets

There is currently just one implementation of sets, the HashSet.

```
iex> set1 = Enum.into 1..5, HashSet.new
#HashSet<[1, 2, 3, 4, 5]>
iex> Set.member? set1, 3
true
iex> set2 = Enum.into 3..8, HashSet.new
#HashSet<[3, 4, 5, 6, 7, 8]>
iex> Set.union set1, set2
#HashSet<[7, 6, 4, 1, 8, 2, 3, 5]>
iex> Set.difference set1, set2
#HashSet<[1, 2]>
iex> Set.difference set2, set1
#HashSet<[6, 7, 8]>
iex> Set.intersection set1, set2
#HashSet<[3, 4, 5]>
```

Prepared exclusively for Richard Bishop

CHAPTER 9

An Aside—What Are Types?

The preceding two chapters described the basics of lists and dictionaries. But you may have noticed that, although I talked about them as types, we didn't really say what I meant.

The first thing to understand is that the primitive data types are not necessarily the same as the types they can represent.

For example, a primitive Elixir list is just an ordered list of values. We can use the [...] literal to create a list, and the | operator to deconstruct and build them. That is the primitive side.

Then there's another layer. Elixir has the List module, which provides a set of functions that operate on lists. Often, these functions simply use recursion and the | operator to add this extra functionality.

In my mind, there's a difference between the primitive list and the functionality of the List module. The primitive list is an implementation, while the List module adds a layer of abstraction. Both implement types, but the type is different. Primitive lists, for example, don't have a flatten function.

Maps are also primitive type. And, like lists, they have an Elixir module that implements a richer, derived map type.

The Keyword type is an Elixir module. But it is implemented as a list of tuples:

```
options = [ {:width, 72}, {:style, "light"}, {:style, "print"} ]
```

Clearly this is still a list, and all the list functions will work on it. But Elixir adds some additional functionality to give you dictionary-like behaviour.

```
iex> options = [ {:width, 72}, {:style, "light"}, {:style, "print"} ]
[width: 72, style: "light", style: "print"]
iex> List.last options
{:style, "print"}
```

Prepared exclusively for Richard Bishop

```
iex> Keyword.get_values options, :style
["light", "print"]
```

In a way, this is a form of the Duck Typing that is talked about in dynamic Object Oriented languages.[1] The `Keyword` module doesn't have an underlying primitive data type. It simply assumes that any value it works on is a list that has been structured a certain way.

This means that the APIs for collections in Elixir are fairly broad. Working with a keyword list, you have access to the APIs in the primitive list type, and the List and Keyword modules. (You also get Enum and Collectable, which we haven't talked about yet).

1. http://en.wikipedia.org/wiki/Duck_typing

Prepared exclusively for Richard Bishop

In this chapter, we'll see
- The Enum module
- The Stream module
- The Collectable Protocol
- Comprehensions

CHAPTER 10

Processing Collections—Enum and Stream

Elixir comes with a number of types that act as collections. We've already seen lists and dictionaries. There are also things such as ranges, files, dictionaries, and even functions. And, as we'll see when we look at protocols on page 259, you can also define your own.

Collections differ in their implementation. But they all share something: you can iterate through them. Some of them share an additional capability—you can add things to them.

Technically, things that can be iterated are said to implement the Enumerable protocol.

Elixir provides two modules that provide a bunch of iteration functions. The Enum module is the workhorse for collections. You'll find yourself using it all the time. I'd strongly recommend getting to know it.

The second module, Stream, lets you enumerate a collection lazily. This means that the next value is only calculated when it is needed. You'll use this less often, but when you do, it's a lifesaver.

I don't want to fill this book with a list of all the various APIs. You'll find the definitive (and up-to-date) list online.[1] Instead, I'll try to illustrate some common uses, and let you browse the documentation for yourself. (But, please do remember to do so. Much of the power of Elixir comes from these libraries.)

Enum—Processing Collections

The Enum module is probably the most used of all the Elixir libraries. Use it to iterate, filter, combine, split, and otherwise manipulate collections. Here are some common tasks:

1. http://elixir-lang.org/docs/

Prepared exclusively for Richard Bishop

```
#
# Convert any collection into a list
#
iex> list = Enum.to_list 1..5
[1, 2, 3, 4, 5]
#
# Concatenate collections
iex> Enum.concat([1,2,3], [4,5,6])
[1, 2, 3, 4, 5, 6]
iex> Enum.concat [1,2,3], 'abc'
[1, 2, 3, 97, 98, 99]
#
# Change Each Element in a Collection
#
iex> Enum.map(list, &(&1 * 10))
[10, 20, 30, 40, 50]
iex> Enum.map(list, &String.duplicate("*", &1))
["*", "**", "***", "****", "*****"]
#
# Select elements
#
iex> Enum.at(10..20, 3)
13
iex> Enum.at(10..20, 20)
nil
iex> Enum.at(10..20, 20, :no_one_here)
:no_one_here
iex> Enum.filter(list, &(&1 > 2))
[3, 4, 5]
iex> Enum.filter(list, &Integer.even?/1)
[2, 4]
iex> Enum.reject(list, &Integer.even?/1)
[1, 3, 5]
#
# Sort and compare elements
#
iex> Enum.sort ["there", "was", "a", "crooked", "man"]
["a", "crooked", "man", "there", "was"]
iex> Enum.sort ["there", "was", "a", "crooked", "man"],
...>        &(String.length(&1) < String.length(&2))
["a", "man", "was", "there", "crooked"]
iex(4)> Enum.max ["there", "was", "a", "crooked", "man"]
"was"
iex(5)> Enum.max_by ["there", "was", "a", "crooked", "man"], &String.length/1
"crooked"
#
# Split a collection
#
iex> Enum.take(list, 3)
[1, 2, 3]
```

Prepared exclusively for Richard Bishop

```
iex> Enum.take_every list, 2
[1, 3, 5]
iex> Enum.take_while(list, &(&1 < 4))
[1, 2, 3]
iex> Enum.split(list, 3)
{[1, 2, 3], [4, 5]}
iex> Enum.split_while(list, &(&1 < 4))
{[1, 2, 3], [4, 5]}
#
# Join a collection
#
iex> Enum.join(list)
"12345"
iex> Enum.join(list, ", ")
"1, 2, 3, 4, 5"
#
# Predicate operations
#
iex> Enum.all?(list, &(&1 < 4))
false
iex> Enum.any?(list, &(&1 < 4))
true
iex> Enum.member?(list, 4)
true
iex> Enum.empty?(list)
false
#
# Merge collections
#
iex> Enum.zip(list, [:a, :b, :c])
[{1, :a}, {2, :b}, {3, :c}]
iex> Enum.with_index(["once", "upon", "a", "time"])
[{"once", 0}, {"upon", 1}, {"a", 2}, {"time", 3}]
#
# Fold elements into a single value
#
iex> Enum.reduce(1..100, &(&1+&2))
5050
iex> Enum.reduce(["now", "is", "the", "time"],fn word, longest ->
...>        if String.length(word) > String.length(longest) do
...>           word
...>        else
...>           longest
...>        end
...> end)
"time"
iex> Enum.reduce(["now", "is", "the", "time"], 0, fn word, longest ->
...>        if String.length(word) > longest do
...>          String.length(word)
...>        else
```

Prepared exclusively for Richard Bishop

```
...>          longest
...>        end
...> end)
4

#
# Deal a hand of cards
#
iex> import Enum
iex> deck = for rank <- '23456789TJQKA', suit <- 'CDHS', do: [suit,rank]
['C2', 'D2', 'H2', 'S2', 'C3', 'D3', ... ]
iex> deck |> shuffle |> take(13)
['DQ', 'S6', 'HJ', 'H4', 'C7', 'D6', 'SJ', 'S9', 'D7', 'HA', 'S4', 'C2', 'CT']
iex> hands = deck |> shuffle |> chunk(13)
[['D8', 'CQ', 'H2', 'H3', 'HK', 'H9', 'DK', 'S9', 'CT', 'ST', 'SK', 'D2', 'HA'],
 ['C5', 'S3', 'CK', 'HQ', 'D3', 'D4', 'CA', 'C8', 'S6', 'DQ', 'H5', 'S2', 'C4'],
 ['C7', 'C6', 'C2', 'D6', 'D7', 'SA', 'SQ', 'H8', 'DT', 'C3', 'H7', 'DA', 'HT'],
 ['S5', 'S4', 'C9', 'S8', 'D5', 'H4', 'S7', 'SJ', 'HJ', 'D9', 'DJ', 'CJ', 'H6']]
```

A note on sorting

In our example of sort, we used

```
Enum.sort(["once", "upon", "a", "time"],
          fn a, b -> String.length(a) <= String.length(b) end)
```

It's important to use <= and not just < if you want the sort to be *stable*.

Your turn…

➤ *Exercise: ListsAndRecursion-5*

Implement the following Enum functions using no library functions or list comprehensions: all?, each, filter, split, and take. You may need to use an if statement to implement filter. The syntax for this is:

```
if condition do
  expression(s)
else
  expression(s)
end
```

➤ *Exercise: ListsAndRecursion-6*

(Harder) Write a function flatten(list) that takes a list that may contain any number of sublists, and those sublists may contain sublists, to any depth. It returns the elements of these lists as a flat list.

```
iex> MyList.flatten([ 1, [ 2, 3, [4] ], 5, [[[6]]]])
[1,2,3,4,5,6]
```

Hint: You may have to use Enum.reverse to get your result in the correct order.

Prepared exclusively for Richard Bishop

Streams—Lazy Enumerables

In Elixir, the Enum module is greedy. This means that when you pass it a collection, it potentially consumes all the contents of that collection. It also means that the result will typically be another collection. Look at the following pipeline:

enum/pipeline.exs

```
[ 1, 2, 3, 4, 5 ]
|> Enum.map(&(&1*&1))
|> Enum.with_index
|> Enum.map(fn {value, index} -> value - index end)
|> IO.inspect   #=> [1,3,7,13,21]
```

The first map function takes the original list and creates a new list of its squares. with_index takes this list, and returns a list of tuples. The next map then subtracts the index from the value, generating a list that gets passed to IO.inspect.

So, this pipeline generates 4 lists on its way to outputing the final result.

Let's look at something different. Here's some code that reads lines from a file and returns the longest.

enum/longest_line.exs

```
IO.puts File.read!("/usr/share/dict/words")
        |> String.split
        |> Enum.max_by(&String.length/1)
```

In this case, we read the whole dictionary into memory (on my machine that's 2.4Mb), then split into a list of words (236k elements) before processing it to find the longest (which happens to be *formaldehydesulphoxylate*).

In both of these examples, our code is suboptimal, because each call to Enum is a self-contained thing. Each call takes a collection, and returns a collection.

What we really want to do is process the elements in the collection as we need them. We don't need to store intermediate results as full collections. Instead, we just need to pass the current element from function to function. And that's what streams do.

A Stream is a Composable Enumerator

Here's a simple example of creating a Stream:

```
iex> s = Stream.map [1, 3, 5, 7], &(&1 + 1)
Stream.Lazy[enum: [1, 3, 5, 7],
            funs: [#Function<32.133702391 in Stream.map/2>],
            accs: [], after: [], last: nil]
```

Prepared exclusively for Richard Bishop

If we'd called Enum.map, we'd have seen the result [2,4,6,8] come back immediately. Instead, we get back a Stream.Lazy record that contains a specification of what we intended.

How do we get the stream to start giving us results? Just treat it as a collection, and pass it to a function in the Enum module:

```
iex> s = Stream.map [1, 3, 5, 7], &(&1 + 1)
Stream.Lazy . . .
iex> Enum.to_list s
[2, 4, 6, 8]
```

Because streams are enumerable, you can also pass a stream to a stream function. Because of this, we say that streams are *composable*.

```
iex> squares = Stream.map [1, 2, 3, 4], &(&1*&1)
Stream.Lazy[enum: [1, 2, 3, 4],
 funs: [#Function<32.133702391 in Stream.map/2>],
 accs: [], after: [], last: nil]

iex> plus_ones = Stream.map squares, &(&1+1)
Stream.Lazy[enum: [1, 2, 3, 4],
  funs: [#Function<32.133702391 in Stream.map/2>,
         #Function<32.133702391 in Stream.map/2>],
  accs: [], after: [], last: nil]

iex> odds = Stream.filter plus_ones, fn x -> rem(x,2) == 1 end
Stream.Lazy[enum: [1, 2, 3, 4],
  funs: [#Function<26.133702391 in Stream.filter/2>,
         #Function<32.133702391 in Stream.map/2>,
         #Function<32.133702391 in Stream.map/2>],
  accs: [], after: [], last: nil]

iex> Enum.to_list odds
[5, 17]
```

Of course, in real life we'd have written this as

enum/stream1.exs

```
[1,2,3,4]
|> Stream.map(&(&1*&1))
|> Stream.map(&(&1+1))
|> Stream.filter(fn x -> rem(x,2) == 1 end)
|> Enum.to_list
```

Note that we're never creating intermediate lists—we're just passing successive elements of each of the collections to the next in the chain. The Stream.Lazy record shown in the previous iex session gives a hint of how this works—chained streams are represented as a list of functions, each of which is applied in turn to each element of the stream as it is processed.

Prepared exclusively for Richard Bishop

Streams aren't only for lists. More and more Elixir modules now support streams. For example, here's our longest word code written using streams:

enum/stream2.exs
```
IO.puts File.open!("/usr/share/dict/words")
        |> IO.stream(:line)
        |> Enum.max_by(&String.length/1)
```

The magic here is the call to IO.stream, which takes an IO device (in this case the open file) and converts it into a stream that serves one line at a time. In fact, this is such a useful concept that there's a shortcut:

enum/stream3.exs
```
IO.puts File.stream!("/usr/share/dict/words") |> Enum.max_by(&String.length/1)
```

The good news is that there is no intermediate storage. The bad news is that (at least under Elixir 0.9.4) it runs about two times slower than the previous version. However, consider the case where we were reading data from a remote server, or from an external sensor (maybe temperature readings). Successive lines might arrive slowly, and potentially they might go on for ever. With the Enum implementation, we'd have to wait for all the lines to arrive before we started processing. With streams, we can process them as they arrive.

Infinite Streams

Because streams are lazy, there's no need for the whole collection to be available up-front. For example, if I write

```
iex> Enum.map(1..10_000_000, &(&1+1)) |> Enum.take(5)
[2, 3, 4, 5, 6]
```

it takes about 8 seconds before I see the result. Elixir is creating a 10 million element list, then taking the first five elements from it. If instead I write:

```
iex> Stream.map(1..10000000, &(&1+1)) |> Enum.take(5)
[2, 3, 4, 5, 6]
```

the result comes back instantaneously. The take call just needs 5 values, which it gets from the stream. Once it has them, there's no more processing.

Creating Your Own Streams

Streams are actually implemented solely in Elixir libraries—there is no specific runtime support. However, this doesn't mean you want to drop down to the very lowest level and create your own streamable types. The actual implementation is complex (in the same way that string theory and dating rituals are complex). Instead, you probably want to use some helpful wrapper functions to do the heavy lifting for you. There are a number of these, including cycle,

Prepared exclusively for Richard Bishop

iterate, repeatedly, resource, and unfold. (If you needed proof that the internal implementation is tricky, the fact that these last two names give you almost no hint of their power is something of a giveaway.)

Let's start with the three simplest: cycle, repeatedly, and iterate.

Stream.cycle

Stream.cycle takes an enumerable and returns an infinite stream containing that enumerable's elements. When it gets to the end, it repeats from the beginning, indefinitely. Here's an example that generates the rows in an HTML table with alternating *green* and *white* classes:

```
iex> Stream.cycle(~w{ green white }) |>
...> Stream.zip(1..5) |>
...> Enum.map(fn {class, value} ->
...>      ~s{<tr class="#{class}"><td>#{value}</td></tr>\n} end) |>
...> IO.puts
<tr class="green"><td>1</td></tr>
<tr class="white"><td>2</td></tr>
<tr class="green"><td>3</td></tr>
<tr class="white"><td>4</td></tr>
<tr class="green"><td>5</td></tr>
```

Stream.repeatedly

Stream.repeatedly takes a function and invokes it each time a new value is wanted.

```
iex> Stream.repeatedly(fn -> true end) |> Enum.take(3)
[true, true, true]
iex> Stream.repeatedly(&:random.uniform/0) |> Enum.take(3)
[0.7230402056221108, 0.94581636451987, 0.5014907142064751]
```

Stream.iterate

Stream.iterate(start_value, next_fun) generates an infinite stream. The first value is start_value. The next value is generated by applying next_fun to this value. This continues for as long as the stream is being used, with each value being the result of applying next_fun to the previous value.

Here are some examples:

```
iex> Stream.iterate(0, &(&1+1)) |> Enum.take(5)
[0, 1, 2, 3, 4]
iex> Stream.iterate(2, &(&1*&1)) |> Enum.take(5)
[2, 4, 16, 256, 65536]
iex> Stream.iterate([], &[&1]) |> Enum.take(5)
[[], [[]], [[[]]], [[[[]]]], [[[[[]]]]]]
```

Prepared exclusively for Richard Bishop

Stream.unfold

Now we can get a little more adventurous. Stream.unfold is related to iterate, but you can be more explicit both about the values output to the stream and about the values passed to the next iteration. You supply an initial value and a function. The function uses the argument to create two values, returned as a tuple. The first is the value to be returned by this iteration of the stream and the second is the value to be passed to the function on the next iteration of the stream. If the function returns nil, the stream terminates.

This sounds pretty abstract, but unfold is actually quite useful—it is a general way of creating a potentially infinite stream of values where each value is some function of the previous state.

The key is the generating function. It's general form is

```
fn state -> { stream_value, new_state } end
```

For example, here's a stream of Fibonacci numbers:

```
iex> Stream.unfold({0,1}, fn {f1,f2} -> {f1, {f2, f1+f2}} end) |> Enum.take(15)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

Here the *state* is a tuple containing the current and the next number in the sequence. We seed it with the initial state of {0,1}. The value returned by each iteration of the stream is the first of the state values. The new state moves one down the sequence, so an initial state of {f1,f2} becomes a new state of {f2,f1+f2}.

Stream.resource

At this point, you might be wondering how streams can interact with external resources. We've already seen how you can turn the contents of a file into a stream of lines, but how could you implement this yourself? You'd need to open the file when the stream first starts, then return successive lines, and then close the file at the end. Or maybe you wanted to turn a database result set cursor into a stream of values. Here you'd have to execute the query when the stream starts, return each row as stream values, and finally close the query at the end. And that's where Stream.resource comes in.

Stream.resource builds upon Stream.unfold. It makes two changes.

The first argument to unfold is the initial value to be passed to the iteration function. But if that value is a resource, we don't want to open it until the stream actually starts delivering values, and that might not happen until long after we first create the stream. To get around this, resource takes not a value, but a function that returns the value. That's the first difference.

Prepared exclusively for Richard Bishop

Second, when the stream is done with the resource, we may need to close it. That's what the third argument to Stream.resource does—it takes the final accumulator value and does whatever is needed to deallocate the resource.

Here's an example from the library documentation:

```
Stream.resource(fn -> File.open("sample") end,
                fn file ->
                  case IO.read(file, :line) do
                    line when is_binary(line) -> { line, file }
                    _ -> nil
                  end
                end,
                fn file -> File.close!(file) end)
```

The first function opens the file when the stream becomes active, passing it to the second funtion. This reads the file, line by line, returning either the line and the file as a tuple, or nil at end of file. The third function closes the file.

Let's finish with a different kind of resource: time. We'll implement a timer that counts down the number of seconds until the start of the next minute. It uses a stream resource to do this. The allocation function returns the number of seconds left until the next minute starts. It will do this each time the stream is evaluated, so we'll get an countdown that varies depending on when it is called.

The iteration function looks at the time left. If zero, it returns nil, otherwise it sleeps for a second, and returns the current countdown as a string, along with the decremented counter.

In this case, there's no resource deallocation, so the third function does nothing.

Here's the code:

enum/countdown.exs

```
defmodule Countdown do

  def sleep(seconds) do
    receive do
      after seconds*1000 -> nil
    end
  end

  def say(text) do
    spawn fn -> :os.cmd('say #{text}') end
  end
```

Prepared exclusively for Richard Bishop

```
  def timer do
    Stream.resource(
      # return the number of seconds to the start of the next minute
      fn ->
         {_h,_m,s} = :erlang.time
         60 - s - 1
      end,

      # wait for the next second, then return its countdown
      fn
        count when count == 0 -> nil;

        count ->
          sleep(1)
          { inspect(count), count - 1 };
      end,

      # nothing to deallocate
      fn _ ->  end
    )
  end
end
```

(The eagle-eyed among you will have noticed a function called `say` in the `Countdown` module. This executes the shell command `say`, which on OS X speaks its argument. You could substitute `espeak` on Linux and `ptts` on Windows.)

Let's play with the code.

```
$ iex countdown.exs
iex> counter = Countdown.timer
#Function<17.133702391 in Stream.resource/3>
iex> printer = counter |> Stream.each(&IO.puts/1)
Stream.Lazy[enum: #Function<17.133702391 in Stream.resource/3>,
 funs: [#Function<0.133702391 in Stream.each/2>], accs: [], after: [],
 last: nil]
iex> speaker = printer |> Stream.each(&Countdown.say/1)
Stream.Lazy[enum: #Function<17.133702391 in Stream.resource/3>,
 funs: [#Function<0.133702391 in Stream.each/2>,
  #Function<0.133702391 in Stream.each/2>], accs: [], after: [], last: nil]
```

So far, we've built a stream that creates time events, prints the countdown value, and also speaks it. But there's been no output, as we haven't yet asked the stream for any values. Let's do that now:

```
iex> speaker |> Enum.take(5)
37     ** numbers are output once
36     ** per second. Even cooler,the
35     ** computer says
34     ** "thirty seven", "thirty six"…
33
```

Prepared exclusively for Richard Bishop

```
["37", "36", "35", "34", "33"]
```

Cool—we must have started it around 22 seconds into a minute, so the countdown starts at 37. Let's use the same stream again, a few seconds later:

```
iex> speaker |> Enum.take(5)
29
28
27
26
25
["29", "28", "27", "26", "25"]
```

Wait some more seconds, and this time let it run to the top of the minute:

```
iex> speaker |> Enum.to_list
6
5
4
3
2
1
["6", "5", "4", "3","2", "1"]
```

This is clearly not great code, as it fails to correct the sleep time for any delays introduced by our code.

But it illustrates a very cool point. Lazy streams let you deal with resources that are asychronous to your code, and the fact that they are initialized every time they are used means they are effectively side-effect free. Every time we pipe our stream to an Enum function, we get a fresh set of values, computed at that time.

Streams in Practice

In the same way that functional programming requires you to look at problems in a new way, streams ask you to look at iteration and collections afresh. Not every situation where you are iterating requires a stream. But for those times where you want to defer processing until you actually need the data, and those times where you need to deal with large numbers of things without necessarily generating them all at once, consider using a stream.

The Collectable Protocol

The Enumerable protocol lets you iterate over the elements in a type—given a collection, you can get the elements. Collectable is in some sense the opposite—it allows you to build a collection by inserting elements into it.

Prepared exclusively for Richard Bishop

Not all collections are collectable. Ranges, for example, cannot have new entries added to them.

In practice, the collectable API is pretty low-level, so you'll typically access it via Enum.into and when using comprehensions (which we cover in the next section'.

For example, we can inject the elements of a range into an empty list using

```
iex> Enum.into 1..5, []
[1, 2, 3, 4, 5]
```

If the list is not empty, the new elements are tacked on to the end:

```
iex> Enum.into 1..5, [100, 101 ]
[100, 101, 1, 2, 3, 4, 5]
```

Output streams are collectable, so the following code lazily copies standard input to standard output:

```
iex> Enum.into IO.stream(:stdio, :line), IO.stream(:stdio, :line)
```

Comprehensions

When you're writing functional code, you often find yourself mapping and filtering collections of things. To make your life easier (and your code easier to read), Elixir provides a general purpose shortcut for this, the *comprehension*.

The idea of a comprehension is fairly simple: given one or more collections, extract all combinations of values from each, optionally filter the values, and then generate a new collection using the values that remain.

The general syntax for comprehensions is deceptively simple:

result = for *generator or filter...* [, into: *value*], do: *expression*

Let's see a couple of basic examples before we get into the details.

```
iex> for x <- [ 1, 2, 3, 4, 5 ], do: x * x
[1, 4, 9, 16, 25]
iex> for x <- [ 1, 2, 3, 4, 5 ], x < 4, do: x * x
[1, 4, 9]
```

A generator is a clause that specifies how you want to extract values from a collection.

```
pattern <- list
```

Any variables matched in the pattern are available for use in the rest of the comprehension (including the block). For example, x <- [1,2,3] says that we want to first run the rest of the comprehension with x set to 1. Then we run

Prepared exclusively for Richard Bishop

it with x set to 2, and so on. If we have two generators, their operations are nested, so

```
x <- [1,2], y <- [5,6]
```

will run the rest of the comprehension with x=1, y=5; x=1, y=6; x=2, y=5; and x=2, y=6. We can use those values of x and y in the do block.

```
iex> for x <- [1,2], y <- [5,6], do:  x * y
[5, 6, 10, 12]
iex> for x <- [1,2], y <- [5,6], do:  {x,  y}
[{1, 5}, {1, 6}, {2, 5}, {2, 6}]
```

You can use variables from generators in later generators:

```
iex> mix_maxes = [{1,4}, {2,3}, {10, 15}]
[{1, 4}, {2, 3}, {10, 15}]
iex> for {min,max} <- mix_maxes, n <- min..max, do: n
[1, 2, 3, 4, 2, 3, 10, 11, 12, 13, 14, 15]
```

A filter is a predicate. It acts as a gatekeeper for the rest of the comprehension—if the condition is false, then the comprehension moves on to the next iteration without generating an output value.

For example, the code that follows uses an comprehension to list pairs of numbers from 1 to 8 whose product is a multiple of 10. It uses two generators (to cycle through the pairs of numbers) and two filters. The first filter only allows pairs where the first number is at least the value of the second. The second filter checks to see if the product is a multiple of 10.

```
iex> first8 = [ 1,2,3,4,5,6,7,8 ]
[1, 2, 3, 4, 5, 6, 7, 8]
iex> for x <- first8, y <- first8, x >= y, rem(x*y, 10)==0, do: { x, y }
[{5, 2}, {5, 4}, {6, 5}, {8, 5}]
```

This comprehension iterates 64 times, with x=1, y=1; x=1, y=2, and so on. However, the first filter cuts the iteration short when x is less than y. This means the second filter only runs 36 times.

Because the first term in a generator is a pattern, we can use it to deconstruct structured data. Here's a comprehension that swaps the keys and values in a keyword list.

```
iex> reports = [ dallas: :hot, minneapolis: :cold, dc: :muggy, la: :smoggy ]
[dallas: :hot, minneapolis: :cold, dc: :muggy, la: :smoggy]
iex> for { city, weather } <- reports, do: { weather, city }
[hot: :dallas, cold: :minneapolis, muggy: :dc, smoggy: :la]
```

Prepared exclusively for Richard Bishop

Comprehensions Work on Bits, Too

When you think about it, a bitstring (and, by extension, a binary or a string) is simply a collection of ones and zeroes. So it's probably no surprise that comprehensions work on them, too. What might be surprising is the syntax:

```
iex> for << ch <- "hello" >>, do: ch
'hello'
iex> for << ch <- "hello" >>, do: <<ch>>
["h", "e", "l", "l", "o"]
```

Here the generator is enclosed in << and >>, indicating a binary. In the first case, the do block returns the integer code for each character, so the resulting list is [104, 101, 108, 108, 111], which iex displays as 'hello'.

In the second case, we convert the code back into a string, and the result is a list of those one-character strings.

Again, the thing to the left of the <- is a pattern, and so we can use binary pattern matching. Let's convert a string into the octal representation of its characters:

```
iex> for << << b1::size(2), b2::size(3), b3::size(3) >> <- "hello" >>,
...> do: "0#{b1}#{b2}#{b3}"
["0150", "0145", "0154", "0154", "0157"]
```

Scoping and Comprehensions

All variable assignments inside a comprehension are local to that comprehension—you will not affect the value of a variable in the outer scope.

```
iex> name = "Dave"
"Dave"
iex> for name <- ~w{ cat, dog }, do: String.upcase(name)
["CAT,", "DOG"]
iex> name
"Dave"
iex>
```

The Value Returned by a Comprehension

In the examples we've seen so far, the comprehension has returned a list. The list contains the values returned by the do expression for each iteration of the comprehension.

This behavior can be changed with the into: parameter. This takes a collection which is to receive the results of the comprehension. For example, we can populate a map using

```
iex> for x <- ~w{ cat dog }, into: %{}, do: { x, String.upcase(x) }
```

Prepared exclusively for Richard Bishop

```
%{"cat" => "CAT", "dog" => "DOG"}
```

It might be clearer to use Map.new in this case:

```
iex> for x <- ~w{ cat dog }, into: Map.new, do: { x, String.upcase(x) }
%{"cat" => "CAT", "dog" => "DOG"}
```

The collection doesn't have to be empty:

```
iex> for x <- ~w{ cat dog }, into: %{"ant" => "ANT"}, do: { x, String.upcase(x) }
%{"ant" => "ANT", "cat" => "CAT", "dog" => "DOG"}
```

In the next section we'll look at protocols, which specify common behaviors across different types. The into: option takes values that implement the Collectable protocol. These include lists, binaries, functions, maps, files, hash dicts, hash sets, and IO streams, so we can write things such as:

```
iex> for x <- ~w{ cat dog }, into: IO.stream(:stdio,:line), do: "<<#{x}>>\n"
<<cat>>
<<dog>>
%IO.Stream{device: :standard_io, line_or_bytes: :line, raw: false}
```

Your turn…

➤ *Exercise: ListsAndRecursion-7*

In the last exercise of the *Lists and Recursion* chapter, you wrote a span funtion. Use it and list comprehensions to return a list of the prime numbers from 2 to n.

➤ *Exercise: ListsAndRecursion-8*

Pragmatic Bookshelf has offices in Texas (TX) and North Carolina (NC), so we have to charge sales tax on orders shipped to these states. The rates can be expressed as a keyword list[2]

```
tax_rates = [ NC: 0.075, TX: 0.08 ]
```

Here's a list of orders:

```
orders = [
    [ id: 123, ship_to: :NC, net_amount: 100.00 ],
    [ id: 124, ship_to: :OK, net_amount:  35.50 ],
    [ id: 125, ship_to: :TX, net_amount:  24.00 ],
    [ id: 126, ship_to: :TX, net_amount:  44.80 ],
    [ id: 127, ship_to: :NC, net_amount:  25.00 ],
    [ id: 128, ship_to: :MA, net_amount:  10.00 ],
    [ id: 129, ship_to: :CA, net_amount: 102.00 ],
    [ id: 120, ship_to: :NC, net_amount:  50.00 ] ]
```

2. I wish it were that simple....

Prepared exclusively for Richard Bishop

Write a function that takes both lists and returns a copy of the orders, but with an extra field, `total_amount` which is the net plus sales tax. If a shipment is not to NC or TX, there's no tax applied.

Prepared exclusively for Richard Bishop

CHAPTER 11

Strings and Binaries

We've been happily using strings without really discussing them. Let's rectify that.

String Literals

Elixir has two kinds of string: single quoted and double quoted. They differ significantly in their internal representation. But they also have many things in common.

- Strings can hold characters in UTF-8 encoding.
- They may contain escape sequences:

\a	BEL (0x07)	\b	BS (0x08)	\d	DEL (0x7f)
\e	ESC (0x1b)	\f	FF (0x0c)	\n	NL (0x0a)
\r	CR (0x0d)	\s	SP (0x20)	\t	TAB (0x09)
\v	VT (0x0b)	\ddd	Octal	\xhhh	1–6 hex digits

- They allow interpolation on Elixir expressions using the syntax #{...}.

```
iex> name = "dave"
"dave"
iex> "Hello, #{String.capitalize name}!"
"Hello, Dave!"
```

- Characters that would otherwise have special meaning can be escaped with a backslash
- They support *heredocs*

Prepared exclusively for Richard Bishop

Heredocs

Any string can span several lines. To illustrate this, we'll use both IO.puts and IO.write. We use write for the multiline string because puts always appends a newline, and we want to see the contents without this.

```
IO.puts "start"
IO.write "
   my
   string
"
IO.puts "end"
```

produces:

```
start

   my
   string
end
```

Notice how the multiline string retains the leading and trailing newlines, and the leading spaces on the intermediate lines.

The *heredoc* notation fixes this. Triple the string delimiter (''' or """), and indent the trailing delimiter to the same margin as your string contents, and you get:

```
IO.puts "start"
IO.write """
   my
   string
   """
IO.puts "end"
```

produces:

```
start
my
string
end
```

Heredocs are used extensively to add documentation to functions and modules.

Sigils

Like Ruby, Elixir has an alternative syntax for some literals. We've already seen it with regular expressions, where we wrote ~r{...}. In Elixir, these ~-style literals are called *sigils* (a symbol with magical powers).

Prepared exclusively for Richard Bishop

A sigil starts with a tilde, followed by an upper or lowercase letter, some delimited content, and perhaps some options. The delimiters can be <...>, {...}, [...], (...), |...|, /.../, "...", and '...'

The letter determines the sigil's type:

~C a character list with no escaping or interpolation

~c a character list, escaped and interpolated just like a single quoted string

~R a regular expression with no escaping or interpolation

~r a regular expression, escaped and interpolated

~S string with no escaping or interpolation

~s string, escaped and interpolated just like a double quoted string

~W a list of whitespace-delimited words, with no escaping or interpolation

~w a list of whitespace-delimited words, with escaping and interpolation

Here are some examples, using a variety of delimiters.

```
iex> ~C[1\n2#{1+2}]
'1\\n2\#{1+2}'
iex> ~c"1\n2#{1+2}"
'1\n23'
iex> ~S[1\n2#{1+2}]
"1\\n2\#{1+2}"
iex> ~s/1\n2#{1+2}/
"1\n23"
iex> ~W[the c#{'a'}t sat on the mat]
["the", "c\#{'a'}t", "sat", "on", "the", "mat"]
iex> ~w[the c#{'a'}t sat on the mat]
["the", "cat", "sat", "on", "the", "mat"]
```

The ~W and ~w sigils take an optional type specifier, a, c, or s, which determines whether it returns atoms, a list, or strings of characters. (We've already seen the ~r options.)

```
iex> ~w[the c#{'a'}t sat on the mat]a
[:the, :cat, :sat, :on, :the, :mat]
iex> ~w[the c#{'a'}t sat on the mat]c
['the', 'cat', 'sat', 'on', 'the', 'mat']
iex> ~w[the c#{'a'}t sat on the mat]s
["the", "cat", "sat", "on", "the", "mat"]
```

The delimiter can be any nonword character. If it is one of the characters (, [, {, or <, then the terminating delimiter is the corresponding closing character. Otherwise the terminating delimiter is the next nonescaped occurrence of the opening delimiter.

Prepared exclusively for Richard Bishop

Elixir does not check the nesting of delimiters, so the sigil ~s{a{b} is the three character string a{b.

If the opening delimiter is three single or three double quotes, the sigil is treated as a heredoc.

```
iex> ~w"""
...> the
...> cat
...> sat
...> """
["the", "cat", "sat"]
```

If you want to specify modifiers with heredoc sigils (most commonly you'd do this with ~r), add them after the trailing delimiter.

```
iex> ~r"""
...> hello
...> """i
~r"hello\n"i
```

One of the interesting things about sigils is that you can define your own. We talk about this in Part 3 on page 279.

The Name "strings"

So, before we get further into this, I need to explain something. In most other languages, you'd call 'cat' and "cat" both *strings*. And that's what I've been doing so far. But Elixir has a different convention.

In Elixir, the convention is that we only call double-quoted strings "strings." The single-quoted form is a character list.

This is important because the single and double quoted forms are radically different internally, and libraries that work on strings only work on the double quoted form.

Let's explore the differences in more detail.

Single Quoted Strings—Lists of Character Codes

Single quoted strings are represented as a list of integer values, each value corresponding to a codepoint in the string. For this reason, we refer to them as *character lists* (or *char lists*).

```
iex> str = 'olé'
'olé'
iex> is_list str
true
```

Prepared exclusively for Richard Bishop

```
iex> length str
3
iex> Enum.reverse str
'élo'
```

This is confusing: iex *says* it is a list, but it shows the value as a string. That's because iex prints a list of integers as a string if it believes each number in the list is a printable character. You can try this for yourself.

```
iex> [ 67, 65, 84 ]
'CAT'
```

You can look at the internal representation in a number of ways:

```
iex> str = 'olé'
'olé'
iex> :io.format "~w~n", [ str ]
[111,108,233]
:ok
iex> list_to_tuple str
{111, 108, 233}
iex> str ++ [0]
[111, 108, 233, 0]
```

The `~w` in the format string forces `str` to be written as an Erlang term—the underlying list of integers. The `~n` is a newline.

The last example creates a new character list with a null byte at the end. iex no longer thinks that all the bytes are printable, and so returns the underlying character codes.

If a character list contains characters Erlang considers nonprintable, you'll see the list representation.

```
iex> '∂x/∂y'
[8706, 120, 47, 8706, 121]
```

Because a character list is a list, we can use the usual pattern matching and List functions.

```
iex> 'pole' ++ 'vault'
'polevault'
iex> 'pole' -- 'vault'
'poe'
iex> List.zip [ 'abc', '123' ]
[{97, 49}, {98, 50}, {99, 51}]
iex> [ head | tail ] = 'cat'
'cat'
iex> head
99
iex> tail
```

Prepared exclusively for Richard Bishop

```
'at'
iex> [ head | tail ]
'cat'
```

Why is the head of 'cat' 99, and not *c*?. Remember that a char list is just a list of integer character codes, so each individual entry is a number. It happens that 99 is the code for a lower case *c*.

In fact, the notation `?c` returns the integer code for the character `c`. This is often useful when using patterns to extract information from character lists. Here's a simple module that parses the character list representation of an optionally signed decimal number.

strings/parse.exs

```
defmodule Parse do

  def number([ ?- | tail ]), do: _number_digits(tail, 0) * -1
  def number([ ?+ | tail ]), do: _number_digits(tail, 0)
  def number(str),           do: _number_digits(str,  0)

  defp _number_digits([], value), do: value
  defp _number_digits([ digit | tail ], value)
  when digit in '0123456789' do
    _number_digits(tail, value*10 + digit - ?0)
  end
  defp _number_digits([ non_digit | _ ], _) do
    raise "Invalid digit '#{[non_digit]}'"
  end
end
```

Let's try it in iex.

```
iex> c("parse.exs")
[Parse]
iex> Parse.number('123')
123
iex> Parse.number('-123')
-123
iex> Parse.number('+123')
123
iex> Parse.number('+9')
9
iex> Parse.number('+a')
** (RuntimeError) Invalid digit 'a'
```

Your turn…

➤ *Exercise: StringsAndBinaries-1*

Write a function that returns true if a single-quoted string contains only printable ASCII characters (space through tilde).

Prepared exclusively for Richard Bishop

➤ *Exercise: StringsAndBinaries-2*

Write anagram?(word1, word2) that returns true if its parameters are anagrams.

➤ *Exercise: StringsAndBinaries-3*

Try the following in iex:

```
iex> [ 'cat' | 'dog' ]
['cat',100,111,103]
```

Why does iex print 'cat' as a string, but 'dog' as individual numbers?

➤ *Exercise: StringsAndBinaries-4*

(Harder) Write a function that takes a single-quoted string of the form *number* [+-*/] *number* and returns the result of the calculation. The individual numbers do not have leading plus or minus signs.

```
calculate('123 + 27') # => 150
```

Binaries

The binary type represents a sequence of bits.

A binary literal looks like << *term,...* >>.

The simplest term is just a number from 0 to 255. The numbers are stored as successive bytes in the binary.

```
iex> b = << 1, 2, 3 >>
<<1, 2, 3>>
iex> byte_size b
3
iex> bit_size b
24
```

You can specify modifiers to set the size (in bits) of any term. This is useful when working with binary formats such as media files and network packets.

```
iex> b = << 1::size(2), 1::size(3) >>
<<9::size(5)>>
iex> byte_size b
1
iex> bit_size b
5
```

You can store integers, float, and other binaries in binaries.

```
iex> int = << 1 >>
<<1>>
iex> float = << 2.5 :: float >>
<<64, 4, 0, 0, 0, 0, 0, 0>>
```

Prepared exclusively for Richard Bishop

```
iex> mix = << int :: binary, float :: binary >>
<<1, 64, 4, 0, 0, 0, 0, 0, 0>>
```

Let's finish an initial look at binaries with an example of some bit extraction. An IEEE-754 float has a sign bit, 11 bits of exponent, and 52 bits of mantissa. The exponent is biased by 1023, and the mantissa is a fraction with the top bit assumed to be 1. So we can extract the fields and then use :math.pow, which performs exponentiation, to reassemble the number:

```
iex> << sign::size(1), exp::size(11), mantissa::size(52) >> = << 3.14159::float >>
iex> (1 + mantissa / :math.pow(2, 52)) * :math.pow(2, exp-1023)
3.14159
```

Double Quoted Strings are Binaries

Whereas single quoted strings are stored as char lists, the contents of a double quoted string (*dqs*) are stored as a consecutive sequence of bytes in UTF-8 encoding. Clearly this is more efficient in terms of memory and certain forms of access, but it does have two implications.

First, because UTF-8 characters can take more than a single byte to represent, the size of the binary is not necessarily the length of the string.

```
iex> dqs = "∂x/∂y"
"∂x/∂y"
iex> String.length dqs
5
iex> byte_size dqs
9
iex> String.at(dqs, 0)
"∂"
iex> String.codepoints(dqs)
["∂", "x", "/", "∂", "y"]
iex> String.split(dqs, "/")
["∂x", "∂y"]
```

Second, because you're no longer using lists, you need to learn and work with the binary syntax alongside the list syntax in your code.

Strings and Elixir Libraries

When Elixir library documentation uses the word *string* (and most of the time it uses the word *binary*), it means double quoted strings.

The String module defines a number of functions that work with double quoted strings.

Prepared exclusively for Richard Bishop

at(str, offset)

Returns the grapheme at the given offset (starting at 0). Negative offsets count from the end of the string.

```
iex> String.at("∂og", 0)
"∂"
iex> String.at("∂og", -1)
"g"
```

capitalize(str)

Converts str to lowercase, and then capitalize the first character

```
iex> String.capitalize "école"
"École"
iex> String.capitalize "ÎÎÎÎÎ"
"Îîîîî"
```

codepoints(str)

Returns the codepoints in str.

```
iex> String.codepoints("José's ∂øg")
["J", "o", "s", "é", "'", "s", " ", "∂", "ø", "g"]
```

downcase(str)

Converts str to lowercase

```
iex> String.downcase "ØRSteD"
"ørsted"
```

duplicate(str, n)

Returns a string containing n copies of str.

```
iex> String.duplicate "Ho! ", 3
"Ho! Ho! Ho! "
```

ends_with?(str, suffix | [suffices])

True if str ends with any of the given suffices.

```
iex> String.ends_with? "string", ["elix", "stri", "ring"]
true
```

first(str)

Returns the first grapheme from str.

```
iex> String.first "∂og"
"∂"
```

graphemes(str)

Returns the graphemes in the string. This is different to the codepoints function, which lists combining characters separately.

Prepared exclusively for Richard Bishop

```
iex>String.codepoints "Ā`stute"
["Ā", "`", "s", "t", "u", "t", "e"]
iex> String.graphemes "Ā`stute"
["Ā`", "s", "t", "u", "t", "e"]
```

last(str)

Returns the last grapheme from str.

```
iex> String.last "∂og"
"g"
```

length(str)

Returns the number of graphemes in str.

```
iex> String.length "∂x/∂y"
5
```

ljust(str, new_length, padding \\ " ")

Returns a new string, at least new_length characters long, containing str left justified and padded with padding.

```
iex> String.ljust("cat", 5)
"cat  "
```

lstrip(str)

Removes leading whitespace from str.

```
iex> String.lstrip "\t\f     Hello\t\n"
"Hello\t\n"
```

lstrip(str, character)

Removes leading copies of character (an integer codepoint) from str.

```
iex> String.lstrip "!!!SALE!!!", ?!
"SALE!!!"
```

next_codepoint(str)

Splits str into its leading codepoint and the rest, or :no_codepoint if str is empty. This may be used as the basis of an iterator

strings/nextcodepoint.ex

```
defmodule MyString do
  def each(str, func), do: _each(String.next_codepoint(str), func)

  defp _each({codepoint, rest}, func) do
    func.(codepoint)
     _each(String.next_codepoint(rest), func)
  end

  defp _each(nil, _), do: []
end
```

Prepared exclusively for Richard Bishop

```
MyString.each "∂og", fn c -> IO.puts c end
```

produces:

```
∂
o
g
```

next_grapheme(str)

Same as next_codepoint, but returns graphemes (and :no_grapheme on completion).

printable?(str)

Returns true if str contains only printable characters.

```
iex> String.printable? "José"
true
iex> String.printable? "\x{0000} a null"
false
```

replace(str, pattern, replacement, options \\ [global: true, insert_replaced: nil])

Replaces pattern with replacement in str under control of options.

If the :global option is true, all occurrences of pattern are replaced, otherwise only the first is replaced.

If :insert_replaced is set to a number, the pattern is inserted into the replacement at that offset. If the option is a list, it is inserted multiple times.

```
iex> String.replace "the cat on the mat", "at", "AT"
"the cAT on the mAT"
iex> String.replace "the cat on the mat", "at", "AT", global: false
"the cAT on the mat"
iex> String.replace "the cat on the mat", "at", "AT", insert_replaced: 0
"the catAT on the matAT"
iex> String.replace "the cat on the mat", "at", "AT", insert_replaced: [0,2]
"the catATat on the matATat"
```

reverse(str)

Reverses the graphemes in a string.

```
iex> String.reverse "pupils"
"slipup"
iex> String.reverse "∑f÷∂"
"∂÷f∑"
```

Prepared exclusively for Richard Bishop

rjust(str, new_length, padding \\ " ")

Returns a new string, at least new_length characters long, containing str right justified and padded with padding.

```
iex> String.rjust("cat", 5, ?>)
">>cat"
```

rstrip(str)

Removes trailing whitespace from str

```
iex> String.rstrip(" line \r\n")
" line"
```

rstrip(str, character)

Removes trailing occurrences of character from str.

```
iex> String.rstrip "!!!SALE!!!", ?!
"!!!SALE"
```

slice(str, offset, len)

Returns a len character substring starting at offset (measured from the end of str if negative.

```
iex> String.slice "the cat on the mat", 4, 3
"cat"
iex> String.slice "the cat on the mat", -3, 3
"mat"
```

split(str, pattern \\ nil, options \\ [global: true])

Splits str into substrings delimited by pattern. If :global is false, only one split is performed. pattern can be a string, a regular expression, or nil. In the latter case, the string is split on whitespace.

```
iex> String.split "   the cat on the mat   "
["the", "cat", "on", "the", "mat"]
iex> String.split "the cat on the mat", "t"
["", "he ca", " on ", "he ma", ""]
iex> String.split "the cat on the mat", ~r{[ae]}
["th", " c", "t on th", " m", "t"]
iex> String.split "the cat on the mat", ~r{[ae]}, parts: 2
["th", " cat on the mat"]
```

starts_with?(str, prefix | [prefixes])

True if str starts with any of the given prefixes.

```
iex> String.starts_with? "string", ["elix", "stri", "ring"]
true
```

strip(str)

Strips leading and trailing whitespace from str

Prepared exclusively for Richard Bishop

```
iex> String.strip "\t  Hello   \r\n"
"Hello"
```

strip(str, character)

Strips leading and trailing instances of character from str

```
iex> String.strip "!!!SALE!!!", ?!
"SALE"
```

upcase(str)

Returns an uppercase version of str

```
iex> String.upcase "José Ørstüd"
"JOSÉ ØRSTÜD"
```

valid_character?(str)

Returns true if str is a single character string containing a valid codepoint.

```
iex> String.valid_character? "∂"
true
iex> String.valid_character? "∂og"
false
```

Your turn…

➤ *Exercise: StringsAndBinaries-5*

Write a function that takes a list of dqs and prints each on a separate line, centered in a column which is the width of the longest. Make sure it works with UTF characters.

```
iex> center(["cat", "zebra", "elephant"])
  cat
 zebra
elephant
```

Binaries and Pattern Matching

The first rule of binaries is "if in any doubt, specify the type of each field." Available types are binary, bits, bitstring, bytes, float, integer, utf8, utf16, or utf32. You can also add the qualifiers:

- size(*n*): the size in bits of the field
- signed or unsigned: for integer fields, should it be interpreted as signed
- endianness: big, little, or native

Use hyphens to separate multiple attributes for a field:

Prepared exclusively for Richard Bishop

```
<< length::unsigned-integer-size(12), flags::bitstring-size(4) >>  = data
```

However, unless you're doing a lot of work with binary file or protocol formats, the most common use of all this scary stuff is to process UTF-8 strings.

String Processing with Binaries

When we process lists, we use patterns that split the head from the rest of the list. With binaries that hold strings, we can do the same kind of trick. We have to specify the type of the head (UTF-8), and make sure the tail remains a binary.

strings/utf-iterate.ex

```
defmodule Utf8 do
  def each(str, func) when is_binary(str), do: _each(str, func)

  defp _each(<< head :: utf8, tail :: binary >>, func) do
    func.(head)
    _each(tail, func)
  end

  defp _each(<<>>, _func), do: []
end

Utf8.each "∂og", fn char -> IO.puts char end
```

produces:

```
8706
111
103
```

The parallels with list processing are clear, but the differences are significant. Rather than use [head | tail], we use << head::utf8, tail::binary >>. And rather than terminate when we reach the empty list, [], we look for an empty binary <<>>.

Your turn…

➤ *Exercise: StringsAndBinaries-6*

Write a function to capitalize the sentences in a string. Each sentence is terminated by a period and a space. Right now, the case of the characters in the string is random.

```
iex> capitalize_sentences("oh. a DOG. woof. ")
"Oh. A dog. Woof. "
```

Prepared exclusively for Richard Bishop

➤ *Exercise: StringsAndBinaries-7*

The Lists chapter had an exercise about calculating sales tax on page 106. We now have the sales information in a file of comma-separated id, ship _to, and amount values. The file looks like this:

```
id,ship_to,net_amount
123,:NC,100.00
124,:OK,35.50
125,:TX,24.00
126,:TX,44.80
127,:NC,25.00
128,:MA,10.00
129,:CA,102.00
120,:NC,50.00
```

Write a function that reads and parses this file, and then passes the result to the sales tax function. Remember that the data should be formatted into a keyword list, and that the fields need to be the correct types (so the id field is an integer, and so on).

You'll need the library functions `File.open`, `IO.read(file, :line)`, and `IO.stream(file)`.

Prepared exclusively for Richard Bishop

CHAPTER 12

Control Flow

We're quite a long way into our exploration of Elixir. But so far we haven't seen many if statement, or anything else that looks like a conventional control flow statement.

There's a good reason for that. In Elixir, we write lots of small functions, and a combination of pattern matching of parameters and guard clauses takes the place of most of the control flow seen in other languages.

Elixir code tries to be declarative, not imperative.

Elixir does have a small set of control flow constructs. The reason I've waited so long to introduce them is that I want you to try not to use them too much. You definitely will, and should, drop the occasional if or case into your code. But, before you do, consider more functional alternatives. The benefit will become obvious as you write more code—functions written without explicit control flow tend to be shorter and more focused. They're easier to read, easier to test, and easier to reuse. If you end up with a 10 or 20 line function in an Elixir program, it is pretty much guaranteed that it will contain one of the constructs in this chapter, and it is pretty much guaranteed you can simplify it.

So, forewarned, let's go.

if and unless

In Elixir, if and its evil twin, unless, take 2 parameters: a condition and a keyword list, which can contain the keys do: and/or else:. If the condition is truthy, the if expression evaluates the code associated with the do: key, otherwise it evaluates the else: code. Either branch may be absent.

```
iex> if 1 == 1, do: "true part", else: "false part"
"true part"
```

Prepared exclusively for Richard Bishop

```
iex> if 1 == 2, do: "true part", else: "false part"
"false part"
```

Just as it does with functions, Elixir provides some syntactic sugar. You can write the first of the previous examples as:

```
iex> if 1 == 1 do
...>   "true part"
...> else
...>   "false part"
...> end
true part
```

Unless is similar.

```
iex> unless 1 == 1, do: "true part", else: "false part"
false part
iex> unless 1 == 2, do: "true part", else: "false part"
true part
iex> unless 1 == 2 do
...>   "true part"
...> else
...>   "false part"
...> end
true part
```

The value of `if` and `unless` is the value of expression that was evaluated.

cond

The `cond` macro lets you list out a series of conditions and it executes the code corresponding to the first of those conditions that is truthy.

In the game of Fizz Buzz, children count up from 1. If the number is a multiple of three, they say "Fizz". For multiples of five, say "Buzz". For multiples of both, say "FizzBuzz". Otherwise, say the number.

In Elixir, we could code this as

control/fizzbuzz.ex

```
defmodule FizzBuzz do

  def upto(n) when n > 0, do: _upto(1, n, [])

  defp _upto(_current, 0, result),  do: Enum.reverse result

  defp _upto(current, left, result) do
    next_answer =
      cond do
        rem(current, 3) == 0 and rem(current, 5) == 0 ->
          "FizzBuzz"
```

Prepared exclusively for Richard Bishop

```
-         rem(current, 3) == 0 ->
-           "Fizz"
-         rem(current, 5) == 0 ->
15          "Buzz"
-         true ->
-           current
-       end
-     _upto(current+1, left-1, [ next_answer | result ])
20   end
- end
```

First, look at the use of cond starting on line 8. We assign the value of the cond expression to next_answer. Inside the cond, we have 4 alternatives—the current number is a multiple of 3 and 5, just 3, just 5, or neither. Elixir examines each in turn and returns the value of the expression following the -> for the first true one. The _upto function then recurses to find the next value. Also note the use of true -> to handle the case where none of the previous conditions match. This is the equivalent of the else or default stanza of a more traditional case statement.

There's a minor problem, though. The result list we build always has the most recent value as its head. When we finish, we'll end up with a list that has the answers in reverse order. That's why in the anchor case (when left is zero), we reverse the result before returning it. This is a very common pattern. And don't worry about performance—list reversal is highly optimized.

Let's try the code in iex:

```
iex> c("fizzbuzz.ex")
[FizzBuzz]
iex> FizzBuzz.upto(20)
[1, 2, "Fizz", 4, "Buzz", "Fizz", 7, 8, "Fizz", "Buzz", 11, "Fizz",
.. 13, 14, "FizzBuzz", 16, 17, "Fizz", 19, "Buzz"]
```

In this particular case, we could do something different and remove the call to reverse. If we process the numbers in reverse order (so we start at n and end at 1), the resulting list will be in the correct order.

control/fizzbuzz1.ex

```
defmodule FizzBuzz do

  def upto(n) when n > 0, do: _downto(n, [])

  defp _downto(0, result),  do: result

  defp _downto(current, result) do
    next_answer =
      cond do
        rem(current, 3) == 0 and rem(current, 5) == 0 ->
```

Prepared exclusively for Richard Bishop

```
          "FizzBuzz"
        rem(current, 3) == 0 ->
          "Fizz"
        rem(current, 5) == 0 ->
          "Buzz"
        true ->
          current
      end
    _downto(current-1, [ next_answer | result ])
  end
end
```

This code is quite a bit cleaner than the previous version. However, it is also slightly less idiomatic—readers will expect to traverse the numbers in a natural order and reverse the result.

But there's a third alternative. The FizzBuzz code transforms a number into some value. And, where possible, we like to code things as transformations. We can use `Enum.map` to transform the range of number from 1 upto `n` to the corresponding FizzBuzz values.

control/fizzbuzz2.ex

```
defmodule FizzBuzz do

  def upto(n) when n > 0 do
    1..n |> Enum.map(&fizzbuzz/1)
  end

  defp fizzbuzz(n) do
    cond do
      rem(n, 3) == 0 and rem(n, 5) == 0 ->
        "FizzBuzz"
      rem(n, 3) == 0 ->
        "Fizz"
      rem(n, 5) == 0 ->
        "Buzz"
      true ->
        n
    end
  end
end
```

Now this section is intended to show you how `cond` works, but you'll often find that it's better not to use it, and to take advantage of pattern matching in function calls instead:

control/fizzbuzz3.ex

```
defmodule FizzBuzz do

  def upto(n) when n > 0 do
```

Prepared exclusively for Richard Bishop

```
    1..n |> Enum.map(&fizzbuzz/1)
  end

  defp fizzbuzz(n) when rem(n, 3) == 0 and rem(n, 5) == 0, do: "FizzBuzz"
  defp fizzbuzz(n) when rem(n, 3) == 0, do: "Fizz"
  defp fizzbuzz(n) when rem(n, 5) == 0, do: "Buzz"
  defp fizzbuzz(n), do: n

end
```

The choice is yours.

case

case lets you test a value against a set of patterns, executes the code associated with the first one that matches, and returns the value of that code. The patterns may include guard clauses.

For example, File.open function returns a two element tuple. If the open is successful, it returns {:ok, file}, where file is an identifier for the open file. If the open fails, it returns {:error, reason}. We can use case to take the appropriate action when we open a file (in this case the code opens its own source file).

control/case.ex

```
case File.open("case.ex")  do

{ :ok, file } ->
  IO.puts "First line: #{IO.read(file, :line)}"

{ :error, reason } ->
  IO.puts "Failed to open file: #{reason}"

end
```

produces:

```
First line: case File.open("case.ex")  do
```

If we change the file name to something that doesn't exist and rerun the code, we get

```
Failed to open file: enoent
```

The full power of nested pattern matches can be used:

control/case1.exs

```
defmodule Users do
  dave = %{ name: "Dave", state: "TX", likes: "programming" }

  case dave do
    %{state: some_state} = person ->
```

Prepared exclusively for Richard Bishop

```
      IO.puts "#{person.name} lives in #{some_state}"

    _ ->
      IO.puts "No matches"
  end
end
```

We've seen how we can use guard clauses to refine the pattern used when matching functions. We can do the same with case.

control/case2.exs

```
defmodule Bouncer do

  dave = %{name: "Dave", age: 27}

  case dave do
    person = %{age: age} when is_number(age) and age >= 21 ->
      IO.puts "You are cleared to enter the Foo Bar, #{person.name}"

    _ ->
      IO.puts "Sorry, no admission"
  end
end
```

Raising Exceptions

First, the official warning. Exceptions in Elixir are *not* control flow structures. Instead, Elixir exceptions are intended for things that should never happen in normal operation. So, the database going down, or a name server failing to respond could be considered exceptional. Failing to open a file whose name was entered by the user is not (you could anticipate that they might mistype it every now and then). Failing to open a configuration file whose name is fixed could be seen as exceptional.

Raise an exception with the raise function. At its simplest, you pass it a string, and it generates an exception of type RuntimeError.

```
iex> raise "Giving up"
** (RuntimeError) Giving up
```

You can also pass the type of the exception, along with other optional attributes. All exceptions implement at least the message attribute.

```
iex> raise RuntimeError
** (RuntimeError) runtime error
iex> raise RuntimeError, message: "override message"
** (RuntimeError) override message
```

Prepared exclusively for Richard Bishop

You use exceptions far less in Elixir that in other languages—the design philosophy is that errors should propagate back up to an external, supervising process. We'll cover this when we talk about OTP Supervisors on page 211.

Elixir has all the usual exception catching mechanisms. To emphasize how little you should use them, I've described them in an appendix on page 289.

Designing With Exceptions

If File.open succeeds, it returns {:ok, file}, where file is the service that gives you access to the file. If it fails, it returns {:error, reason}. So, for code that knows that a file open might not succeed and that wants to handle the fact, you might write:

```
case File.open(user_file_name) do
{:ok, file} ->
  process(file)
{:error, message} ->
  IO.puts :stderr, "Couldn't open #{user_file_name}: #{message}"
end
```

If instead you expect the file to open successfully every time, then you could raise an exception on failure.

```
case File.open("config_file") do
{:ok, file} ->
  process(file)
{:error, message} ->
  raise "Failed to open config file: #{message}"
end
```

Or, you could let Elixir raise an exception for you and write

```
{ :ok, file } = File.open("config_file")
process(file)
```

If the pattern match on the first line fails, Elixir will raise a MatchError exception. It won't be as informative as our version that handled the error explicitly, but if the error should never happen, this form is probably good enough (at least until it triggers the first time and the operations folks say they'd like more information).

So an even better way to handle this is to use File.open!. The trailing exclamation point in the method name is an Elixir convention—if you see it, you know the function will raise an exception on an error, and that exception will be meaningful. So we could simply write

```
file = File.open!("config_file")
```

Prepared exclusively for Richard Bishop

and get on with the rest of our lives.

What we've seen

Elixir has just a few forms of control flow: `if`, `unless`, `cond`, `case`, and (perhaps) `raise`. But, surprisingly, this doesn't matter in practice. Elixir programs are rich and expressive without a lot of branching code. And they're easier to work with as a result.

That finishes our basic tour of Elixir. Now let's start putting it all together, and implement a full project.

Your turn…

➤ *Exercise: ControlFlow-1*

Rewrite the FizzBuzz example using `case`.

➤ *Exercise: ControlFlow-2*

We now have three different implementations of FizzBuzz. One uses `cond`, one uses `case`, and one uses separate functions with guard clauses.

Take a minute to look at all three. Which do you feel best expresses the problem. Which will be easiest to maintain?

The `case` style and the one using guard clauses are somewhat different to control structures in most other languages. If you feel that one of these was the best implementation, can you think of ways of reminding yourself to investigate these options as you write more Elixir code in the future?

➤ *Exercise: ControlFlow-3*

Many built-in functions have two forms. The *xxx* form returns the tuple `{:ok, data}` and the *xxx!* form returns data on success but raises an exception otherwise. However, there are some functions that don't have the *xxx!* form.

Write a function `ok!` that takes an arbitary parameter. If the parameter is the tuple `{:ok, data}` return the data. Otherwise raise an exception containing information from the parameter.

You could use your function like this:

```
file = ok! File.open("somefile")
```

Prepared exclusively for Richard Bishop

In this chapter, we'll see
- Project structure
- The Mix build tool
- ExUnit testing framework
- DocTests

CHAPTER 13

Organizing a Project

Let's stop hacking and get serious.

You'll want to organize your source code, write tests, and handle any dependencies. And you'll want to follow Elixir conventions, because that way you'll get support from the tools.

In this chapter we'll look at mix, the Elixir build tool. We'll investigate the directory structure it uses, and see how to manage external dependencies. And we'll end up using ExUnit to write tests for our code (and to validate the examples in our code's documentation). To motivate this, we'll write a tool that downloads and lists the *n* oldest issues from a GitHub project. Along the way, we'll need to find some libraries, and we'll be making some design decisions which are typical of an Elixir project. We'll call our project *issues*.

The Project: Fetch Issues from GitHub

GitHub provides a nice web API for fetching issues.[1] Simply issue a GET request to

```
https://api.github.com/repos/user/project/issues
```

and you'll get back a JSON list of issues. We'll reformat this, sort it, and filter out the oldest *n*, presenting the result as a table:

```
 #   | created_at           | title
-----+----------------------+----------------------------------------
889 | 2013-03-16T22:03:13Z | MIX_PATH environment variable (of sorts)
892 | 2013-03-20T19:22:07Z | Enhanced mix test --cover
893 | 2013-03-21T06:23:00Z | mix test time reports
898 | 2013-03-23T19:19:08Z | Add mix compile --warnings-as-errors
```

1. http://developer.github.com/v3/

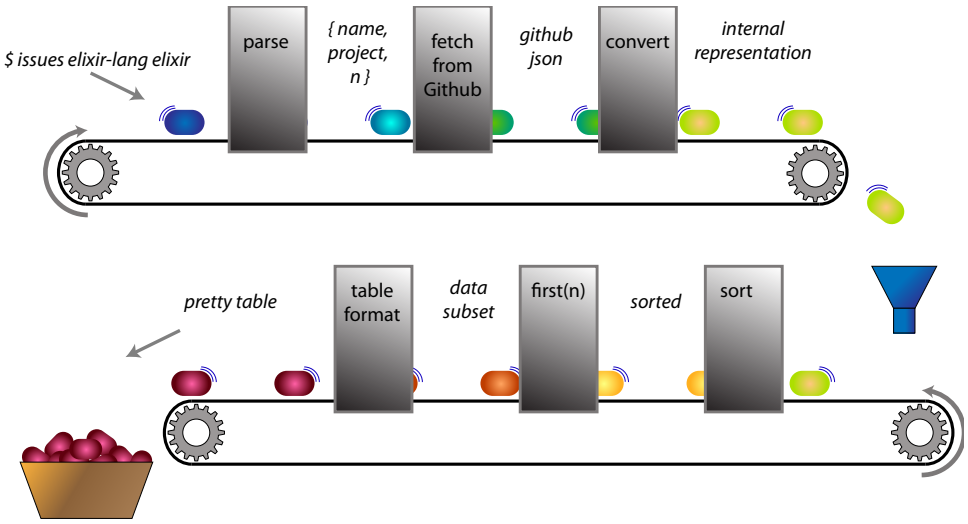
Prepared exclusively for Richard Bishop

How Our Code Will Do It

Our program will be run from the command line. We'll need to pass in a GitHub user name, a project name, and an optional count. This means we'll need some basic command line parsing.

We'll need to access GitHub as an HTTP client, so we'll need to find a library that gives us the client side of HTTP. The response that comes back will be in JSON, so we'll need a library that handles JSON, too. We'll need to be able to sort the resulting structure. And, finally, we'll need to lay out selected fields in a table.

We can think of this data transformation in terms of a production line. Raw data enters at one end, and is transformed by each of the stations in turn.



Here we see data, starting at *command line* and ending at *pretty table*. At each stage, it undergoes a transformation (parse, fetch, and so on). These transformations are the functions we write. We'll cover each one in turn.

Task: Use Mix to Create our New Project

Mix is a command line utility that manages Elixir projects. Use it to create new projects, manage a project's dependencies, run tests, and run your code. If you have Elixir installed, you also have mix. Try running it now:

```
$ mix help
mix                   # Run the default task (current: mix run)
mix archive           # List all archives
mix archive.build     # Archive this project into a .ez file
mix archive.install   # Install an archive locally
```

Prepared exclusively for Richard Bishop

```
mix archive.uninstall # Uninstall archives
mix clean             # Clean generated application files
mix cmd               # Executes the given command
mix compile           # Compile source files
mix compile.protocols # Consolidates all protocols in all paths
mix deps              # List dependencies and their status
mix deps.clean        # Remove the given dependencies' files
mix deps.compile      # Compile dependencies
mix deps.get          # Get all out of date dependencies
mix deps.unlock       # Unlock the given dependencies
mix deps.update       # Update the given dependencies
mix do                # Executes the tasks separated by comma
mix escript.build     # Builds an escript for the project
mix help              # Print help information for tasks
mix hex.config        # Read or update hex config
mix hex.info          # Print hex information
mix hex.key           # Hex API key tasks
mix hex.owner         # Hex package ownership tasks
mix hex.publish       # Publish a new package version
mix hex.search        # Search for package names
mix hex.user          # Hex user tasks
mix loadconfig        # Loads and persists the given configuration
mix local             # List local tasks
mix local.hex         # Install hex locally
mix local.rebar       # Install rebar locally
mix new               # Create a new Elixir project
mix run               # Run the given file or expression
mix test              # Run a project's tests
iex -S mix            # Start IEx and run the default task
```

This is a list of the standard *tasks* that come with mix.[2] (Your list may be a little different, depending on your version of Elixir.) For more information on a particular task, use mix help *taskname*.

```
$ mix help deps

List all dependencies and their status.

Dependencies must be specified in the `mix.exs` file in one of
the following formats:
 . . .
```

Create the project tree

Each Elixir project lives in its own directory tree. If you use mix to manage this tree, then you'll need to follow the mix conventions (which are also the

2. You can also write your own mix tasks, both for a project and to share between projects. Have a look at http://elixir-lang.org/getting_started/mix/3.html if you're interested.

Prepared exclusively for Richard Bishop

conventions of the Elixir community). These conventions are what we'll use in the rest of this chapter.

We'll call our project issues, and so it will go in a directory named issues. We'll create this directory using mix.

At the command line, navigate to a place where you want this new project to live, and type

```
$ mix new issues
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/issues.ex
* creating test
* creating test/test_helper.exs
* creating test/issues_test.exs

Your mix project was created successfully.
You can use mix to compile it, test it, and more:

    cd issues
    mix test

Run `mix help` for more commands.
```

In tree form, the newly created files and directories look like this:

```
issues
├── .gitignore
├── README.md
├── config
│   └── config.exs
├── lib
│   └── issues.ex
├── mix.exs
└── test
    ├── issues_test.exs
    └── test_helper.exs
```

Change into the issues/ directory. Before we start having a look around, this would be a good time to set up version control. I use git, so I do[3]

```
$ git init
```

3. I don't want to clutter the book with lots of version control stuff, so that's the last time I'll mention it. But you'll want to make sure you follow your own VC practices as we go along.

Prepared exclusively for Richard Bishop

```
$ git add .
$ git commit -m "Initial commit of new project"
```

Our new project contains 3 directories and 7 files.

.gitignore
: Lists the files and directories generated as by-products of the build, and which are not to be saved in the repository.

README.md
: A place to put a description of your project (in Markdown) format. If you store your project on GitHub, the contents of this file will appear on the project's home page.

config/
: Eventually we'll put some application-specific configuration here.

lib/
: This is where the source of our project lives. Mix has already added a top-level module (`issues.ex` in our case).

mix.exs
: This source file contains the configuration options for our project. We will be adding stuff to this as our project progresses.

test/
: A place to store our tests. Mix has already created a helper file and a stub for unit tests of the `issues` module.

So, now our job is to add our code. But before we do, let's think a little about the implementation.

Transformation: Parse the Command Line

Let's start with the command line. We really don't want to couple the handling of command line options into the main body of our program, so let's write a separate module to interface between what the user types and what our program does. By convention, this module is called *project*`.CLI` (so our code would be in `Issues.CLI`). And, also by convention, the main entry point to this module will be a function called `run` that takes an array of command line arguments.

Where should we put this module?

Elixir has a convention. Inside the `lib/` directory, create a subdirectory with the same name as the project (so we'd create the directory `lib/issues/`). This directory will contain the main source for our application, one module per

Prepared exclusively for Richard Bishop

file. And each module will be namespaced inside the Issues module—the module naming follows the directory naming.

In this case, the module we want to write is Issues.CLI—it is the CLI module nested inside the Issues module. Let's reflect that in the directory structure and put cli.ex in the lib/issues directory:

```
lib
├── issues
│   └── cli.ex
└── issues.ex
```

Elixir comes bundled with an option parsing library,[4] so we will use that. We'll tell it that -h and --help are possible switches, and anything else is an argument. It returns a tuple, where the first element is a keyword list of the options, and the second is a list of the remaining arguments. So our initial CLI module looks like this:

project/0/issues/lib/issues/cli.ex

```
defmodule Issues.CLI do

  @default_count 4

  @moduledoc """
  Handle the command line parsing and the dispatch to
  the various functions that end up generating a
  table of the last _n_ issues in a github project
  """

  def run(argv) do
    parse_args(argv)
  end

  @doc """
  `argv` can be -h or --help, which returns :help.

  Otherwise it is a github user name, project name, and (optionally)
  the number of entries to format.

  Return a tuple of `{ user, project, count }`, or `:help` if help was given.
  """
  def parse_args(argv) do
    parse = OptionParser.parse(argv, switches: [ help: :boolean],
                                     aliases:  [ h:    :help   ])
    case  parse  do

    { [ help: true ], _, _ }
```

4. http://elixir-lang.org/docs/stable/elixir/OptionParser.html

Prepared exclusively for Richard Bishop

```
      -> :help

    { _, [ user, project, count ], _ }
      -> { user, project, count }

    { _, [ user, project ], _ }
      -> { user, project, @default_count }

    _ -> :help

    end
  end
end
```

Step: Write Some Basic Tests

At this point, I get a little nervous if I don't have some tests. Fortunately, Elixir comes with a wonderful (and simple) testing framework called ExUnit.

Have a look at the file test/issues_test.exs.

project/0/issues/test/issues_test.exs

```
defmodule IssuesTest do
  use ExUnit.Case

  test "the truth" do
    assert(true)
  end
end
```

It acts as a template for all the test files you write. I personally just copy and paste the boilerplate into separate test files as I need them. So let's write tests for our CLI module, putting those tests into the file test/cli_test.exs. We'll test that the option parser successfully detects the -h and --help options, and that it returns the arguments otherwise. We'll also check that it supplies a default value for the count if only two arguments are given.

project/1/issues/test/cli_test.exs

```
defmodule CliTest do
  use ExUnit.Case

  import Issues.CLI, only: [ parse_args: 1 ]

  test ":help returned by option parsing with -h and --help options" do
    assert parse_args(["-h",     "anything"]) == :help
    assert parse_args(["--help", "anything"]) == :help
  end

  test "three values returned if three given" do
    assert parse_args(["user", "project", "99"]) == { "user", "project", 99 }
```

Prepared exclusively for Richard Bishop

```
  end

  test "count is defaulted if two values given" do
    assert parse_args(["user", "project"]) == { "user", "project", 4 }
  end
end
```

These tests all use the basic assert macro provided by ExUnit. This macro is clever—if an assertion fails, it can extract the values from the expression you pass it, giving you a nice error message.

To run our tests, use the mix test task.

```
issues % mix test
Compiled lib/issues.ex
Compiled lib/issues/cli.ex
Generated issues.app
..

Failures:

  1) test three values returned if three given (CliTest)
     Assertion with == failed
     code: Issues.CLI.parse_args(["user", "project", "99"]) ==
                                 {"user", "project", 99}
     lhs:  {"user", "project", "99"}
     rhs:  {"user", "project", 99}
     test/cli_test.exs:10
.
Finished in 0.01 seconds
4 tests, 1 failures
```

Three of the four tests ran successfully. However, when we pass a count as the third parameter, it blows up. See how the assertion shows you the type of assertion (== in this case), the line of code that failed, and the two values that we compared. You can see the difference between the left hand side (lhs), which is the value returned by parse_args and the expected value (the rhs). We were expecting to get a number as the count, but we got a string.

That's easily fixed. The built-in function String.to_integer converts a binary (a string) into an integer.

project/1/issues/lib/issues/cli.ex

```
def parse_args(argv) do
  parse = OptionParser.parse(argv, switches: [ help: :boolean],
                                   aliases:  [ h:    :help   ])
  case  parse  do

  { [ help: true ], _,           _ } -> :help
➤ { _, [ user, project, count ], _ } -> { user, project, String.to_integer(count) }
```

Prepared exclusively for Richard Bishop

```
  { _, [ user, project ],        _ } -> { user, project, @default_count }
  _                                 -> :help
  end
end
```

Your turn…

➤ *Exercise: OrganizingAProject-1*

Do what I did. Honest. Create the project, and write and test the option parser. It's one thing to read about it, but you'll be doing this a lot, so you may as well start now.

Transformation: Fetch from GitHub

Now let's continue down our data transformation chain. Having parsed our arguments, we need to transform them by fetching data from GitHub. So we'll extend our run function to call a function called process, passing it the value returned from the parse_args function. We could have written this

```
process(parse_args(argv))
```

But to understand this code, you have to read it right to left. I prefer to make the chain more explicit using the Elixir pipe operator:

project/1/issues/lib/issues/cli.ex

```
def run(argv) do
  argv
    |> parse_args
    |> process
end
```

We need two variants of the process function. One handles the case where the user asked for help, and parse_args returned :help. The second handles the case where a user, project, and count are returned.

project/1/issues/lib/issues/cli.ex

```
def process(:help) do
  IO.puts """
  usage:  issues <user> <project> [ count | #{@default_count} ]
  """
  System.halt(0)
end

def process({user, project, _count}) do
  Issues.GithubIssues.fetch(user, project)
end
```

Prepared exclusively for Richard Bishop

At this point, we can use mix to run our function. Let's see if help gets displayed.

```
$ mix run -e 'Issues.CLI.run(["-h"])'
usage:  issues <user> <project> [ count | 4 ]
```

You pass mix run an Elixir expression, which gets evaluated in the context of your application. Mix will recompile your application as it is out of date before executing the expression.

If we pass it user and project names, however, we'll blow up, because we haven't written that code yet.

```
% mix run -e 'Issues.CLI.run(["elixir-lang", "elixir"])'
** (UndefinedFunctionError) undefined function: Issues.GithubIssues.fetch/2
    GithubIssues.fetch("elixir-lang", "elixir")
```

Let's write that code now. Our program is going to act as an HTTP client, accessing GitHub through its web API. So, it looks like we'll need an external library.

Task: Use External Libraries

If you come from a world with a single accepted package management system (such as Ruby with its gems), you'll be disappointed by the lack of something similar in the Elixir world. But, remember, it's early days. There were no drive-throughs on the frontier.

At the same time, I think you'll be pleasantly surprised at how easy it is to integrate libraries into your project once you find them.

Finding a Library

The first port of call is http://elixir-lang.org/docs/, the Elixir documentation. Often, you'll find an existing library that does what you want.

Next, have a look to see if any standard Erlang libraries do what you need. This isn't a simple task. You need to visit http://erlang.org/doc/ and look in the left-hand sidebar for *Application Groups*. Here's you'll find libraries sorted by top-level category.

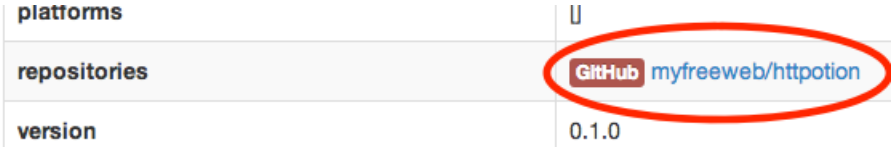
If you find what you're looking for in either of these two places, you can stop, because all these libraries are already available to your application. But if they don't have what you need, you'll have to add an external dependency.

Prepared exclusively for Richard Bishop

The next place to look is http://expm.co, the Elixir/Erlang Package Manager. This is a (small, but growing) list of packages that integrate nicely with a mix-based project.

If all else fails, Google and GitHub are your friends. Search for terms such as *elixir http client* or *erlang distributed logger* and you're likely to turn up the libraries you need.

In our case, we need an HTTP client. Step 1 shows Elixir has nothing built in. Step 2 does find an HTTP client library called `httpc` listed under the `inets` category the Erlang distribution. However, it looks complex, and probably does more than we need, so we'll move on. Next stop is expm.co, where we find two HTTP client libraries, *hackney* and *httpotion*. If you click on either, you get an expm summary page. To get documentation and examples, you need to click the *repository* link.



To me, HTTPotion looks like the simpler option. So how do we include it in our project?

Adding a Library to your Project

Mix takes the view that all external libraries should be copied into the project directory structure. The good news is that it handles all this for you—you just need to list the dependencies, and it does the rest. Remember the `mix.exs` file at the top level of our project?

project/0/issues/mix.exs

```
defmodule Issues.Mixfile do
  use Mix.Project


  def project do
    [ app:     :issues,
      version: "0.0.1",
      elixir:  ">= 0.0.0",
      deps:    deps ]
  end

  # Configuration for the OTP application
  def application do
  end
```

Prepared exclusively for Richard Bishop

```
  # Returns the list of dependencies in the format:
  # { :foobar, git: "https://github.com/elixir-lang/foobar.git", tag: "0.1" }
  #
  # To specify particular versions, regardless of the tag, do:
  # { :barbat, "~> 0.1", github: "elixir-lang/barbat.git" }
  defp deps do
    []
  end
end
```

We add new dependencies to the deps function. As the project is on GitHub, that's very straighforward.

project/1/issues/mix.exs

```
defp deps do
  [
    { :httpotion,  github: "myfreeweb/httpotion" }
  ]
end
```

Once you've done this, you're ready to have mix manage your dependencies.

Use mix deps to list the dependencies and their status:

```
$ mix deps
* httpotion [git: "https://github.com/pragdave/httpotion.git"]
  the dependency is not available, run `mix deps.get`
```

Download the dependencies with mix deps.get

```
$ mix deps.get
* Getting httpotion [git: "https://github.com/pragdave/httpotion.git"]
Cloning into 'deps/httpotion'...
remote: Counting objects: 67, done.
Unpacking objects: 100% (67/67), done.   1% (1/67)
* Getting ibrowse [git: "https://github.com/cmullaparthi/ibrowse.git"]
Cloning into '.../deps/ibrowse'...
remote: Counting objects: 807, done.
Receiving objects:  91% (735/807), 156.00 KiB | * Compiling ibrowse
```

That was slightly surprising. HTTPotion had an internal dependency on a library called ibrowse, and mix was smart enough to download and install it, too.

Run mix deps again:

```
$ mix deps
* ibrowse (git://github.com/cmullaparthi/ibrowse.git)
  locked at 7871e2e (tag: v4.1.0)
  the dependency build is outdated, please run `mix deps.compile`
* httpotion (git://github.com/myfreeweb/httpotion.git)
  locked at c20be2e
```

Prepared exclusively for Richard Bishop

```
the dependency build is outdated, please run `mix deps.compile`
```

This shows that the two libraries are installed but not yet compiled. Mix also remembers the Git hash of each library (it stores them in the file mix.lock). This means that at any point in the future you can get the exact version of the library you use now.

We don't worry about the fact they aren't compiled—mix will automatically compile them the first time we need them.

If you look at your project tree, you'll find a new directory called deps containing your two dependencies. Note that these are themselves just projects, so you can browse their source and read their documentation.

Your turn…

➤ *Exercise: OrganizingAProject-2*

Add the dependency to your project and install it.

Back to the Transformation…

So, back to our problem. We have to write the function GithubIssues.fetch that transforms a user name and project into a data structure containing that project's issues. The HTTPotion page on GitHub gives us a clue,[5] and we write a new module, Issues.GithubIssues:

project/1/issues/lib/issues/github_issues.ex

```
defmodule Issues.GithubIssues do

  alias HTTPotion.Response

  @user_agent  [ "User-agent": "Elixir dave@pragprog.com"]

  def fetch(user, project) do
    response = HTTPotion.get(issues_url(user, project), @user_agent)
    return_code = if Response.success?(response), do: :ok, else: :error
    { return_code, response.body }
  end

  def issues_url(user, project) do
    "https://api.github.com/repos/#{user}/#{project}/issues"
  end

end
```

5. https://github.com/myfreeweb/httpotion

Prepared exclusively for Richard Bishop

We simply call get on the GitHub URL. (We also have to pass in a user agent header to keep the GitHub API happy.) What comes back is a structure. If we have a successful response, so we return a tuple whose first element is :ok, along with the body. Otherwise we return an :error tuple. We still return the body, because the GitHub API encodes an error message in it.

But there's one more thing. If you look at the HTTPotion GitHub page, you'll see that their example code calls HTTPotion.start. That's because HTTPotion actually runs as a separate application, outside your main process.[6] Now, a lot of developers will copy this code, and call start inline like this. (I did myself, until José Valim put me straight.) But there's a better way. Back in our mix.exs file, there's a function called application.

project/0/issues/mix.exs
```
# Configuration for the OTP application
def application do
end
```

OTP is the framework that manages suites of running applications. The app function is used to configure the contents of these suites. By default, this app function does nothing. But we can use it to start extra applications. We tell mix about starting HTTPotion here.[7]

project/1/issues/mix.exs
```
def application do
  [
    applications: [ :httpotion ]
  ]
end
```

Don't worry about the details here—we'll be talking about this extensively in the second part of this book.

We can play with this in iex. Use the -S mix option to run mix before dropping into interaction mode. Because this is the first time we've tried to run our code since installing the dependencies, you'll see them get compiled first.

```
$ iex -S mix
==> ibrowse (compile)
Compiled src/ibrowse_sup.erl
```

6. Actually, you've been doing this already whenever you run code such as mix or iex. But it has been behind the scenes. From here on in, we're going to start opening the curtain.
7. This is something that I found counter-intuitive at first. Erlang (and by extension Elixir) programs are often structured as suites of cooperating subapplications. Often, the code that would be a library in another language is instead a subapplication in Elixir. It might help to think of these as components or services.

Prepared exclusively for Richard Bishop

```
Compiled src/ibrowse_lb.erl
Compiled src/ibrowse_lib.erl
Compiled src/ibrowse_app.erl
Compiled src/ibrowse.erl
Compiled src/ibrowse_http_client.erl
* Compiling httpotion
Compiled lib/httpotion.ex
Generated httpotion.app

iex> Issues.GithubIssues.fetch("elixir-lang", "elixir")
{:ok,
"[{"url":"https://api.github.com/repos/elixir-lang/elixir/issues/970",
labels_url":"https://api.github.com/repos/elixir-lang/elixir/issues/970
/labels{/name}","comments_url":"https://api.github.com/repos/elixir-lang/
elixir/issues/970/comments","events_url":"https://api.github.com/repos/
elixir-lang/elixir/issues/970/events",  …
{"html_url":null,"diff_url":null,"patch_url":null},"body":""}]
```

This is the body of the Git response. It's a tuple with the first element set to :ok. The second element is a single long string containing the data encoded in JSON format.

Transformation: Convert Response

We'll need a JSON library to convert it into a data structure. We'll use the Erlang library jsx,[8] so let's add its dependency to our mix.exs file.

project/2/issues/mix.exs

```
defp deps do
  [
    { :httpotion,  github: "myfreeweb/httpotion" },
    { :jsx,        github: "talentdeficit/jsx"  }
  ]
end
```

Run mix deps.get and you'll end up with jsx installed.

To convert the body from a string, we call the jsx decode function when we return the message from the Github API:

project/3/issues/lib/issues/github_issues.ex

```
def fetch(user, project) do
  response = HTTPotion.get(issues_url(user, project), @user_agent)
  return_code = if Response.success?(response), do: :ok, else: :error
➤ { return_code, :jsx.decode(response.body) }
end
```

8. https://github.com/talentdeficit/jsx

Prepared exclusively for Richard Bishop

We also have to deal with a possible error response from the fetch, so back in the CLI module we write a function that decodes the body and returns it on a success response, and which extracts the error from the body and displays it otherwise.

```
def process({user, project, _count}) do
  Issues.GithubIssues.fetch(user, project)
    |> decode_response
end

def decode_response({:ok, body}), do: body
def decode_response({:error, msg}) do
  {_, message} = List.keyfind(error, "message", 0)
  IO.puts "Error fetching from Github: #{message}"
  System.halt(2)
end
```

The json that Github returns for a successful response is a list with one element per GitHub issue. That element is itself a list of key/value tuples. To make these easier (and more efficient) to work with, we'll convert our list of lists into a list of Elixir HashDicts.[9]

We'll do that by piping our data through this function:

```
def convert_to_list_of_hashdicts(list) do
  list |> Enum.map(&Enum.into(&1, HashDict.new))
end
```

Application Configuration

Before we move on from the GithubFetch module, there's one little tweak I'd like to make. The issues_url function hardcodes the Github URL. Now, in practice, this isn't going to change, but let's use this to illustrate how you can add configuration to your application.

Remember that when we created the project using mix new, it added a config/ directory containing confix.exs. That file is used to store application-level configuration.

It should start with the line

```
use Mix.Config
```

You then write configuration information for each of the applications in your project. In our case, we're configuring the Issues application, so we write:

9. The HashDict library gives you fast access by key to a list of key/value pairs. http://elixir-lang.org/docs/stable/elixir/HashDict.html

Prepared exclusively for Richard Bishop

project/3/issues/config/config.exs
```
use Mix.Config

config :issues, github_url: "https://api.github.com"
```

Each config line adds one or more key/value pairs to the given application's _ environment'. If you have multiple lines for the same application they accumulate, with duplicate keys in later lines overriding values from earlier ones.

In our code, we use Application.get_env function to return a value from the environment.

project/3/issues/lib/issues/github_issues.ex
```
# use a module attribute to fetch the value at compile time
@github_url Application.get_env(:issues, :github_url)

def issues_url(user, project) do
  "#{@github_url}/repos/#{user}/#{project}/issues"
end
```

Because the application environment is commonly used in Erlang code, you'll find yourself using the configuration facility to configure code you import, as well as code you write.

Sometimes you may want to vary the configuration, perhaps depending on your application's environment. One way is to use the import_config function, which reads configuration from a file. If your config.exs contains

```
use Mix.Config
import_config "#{Mix.env}.exs"
```

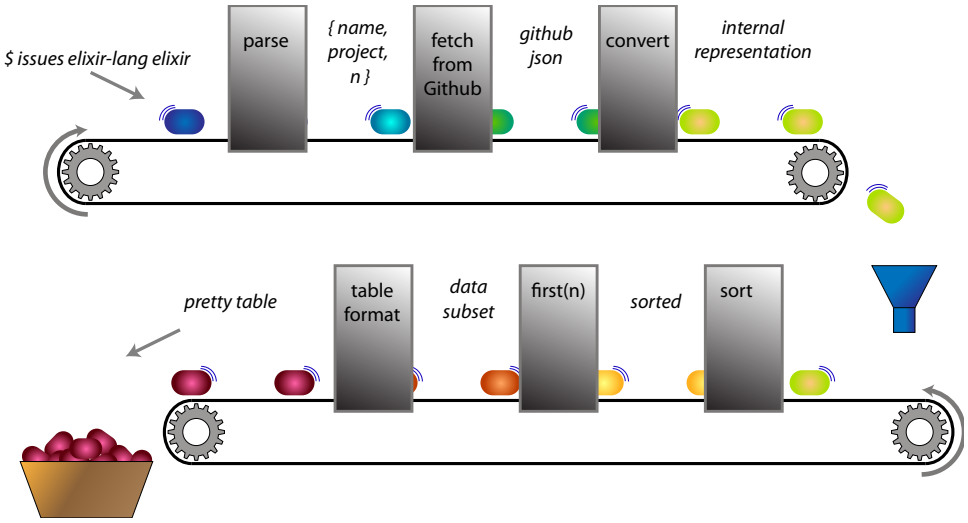
then Elixir will read dev.exs, test.exs, or prod.exs depending on your environment.

You can also pass mix the name of a configuration file using the --config option.

Transformation: Sort Data

Look at our original "design."

Prepared exclusively for Richard Bishop



We're making good progress—we've coded all of the top conveyor belt. Our next transformation is to sort the data on its created_at field. And this can just use a standard Elixir library function, sort/2. We *could* create a new module for this, but it would be pretty lonely. For now we'll put the function in CLI, and keep an eye out for opportunities to move it out if we add other, related functions later.

So now our CLI module contains:

```
def process({user, project, count}) do
  Issues.GithubIssues.fetch(user, project)
    |> decode_response
    |> convert_to_list_of_hashdicts
    |> sort_into_ascending_order
end

def sort_into_ascending_order(list_of_issues) do
  Enum.sort list_of_issues,
            fn i1, i2 -> i1["created_at"] <= i2["created_at"] end
end
```

That sort_into_ascending_order function worries me a little—I get the comparison the wrong way around about 50% of the time, so let's write a little CLI test.

project/3/issues/test/cli_test.exs

```
test "sort ascending orders the correct way" do
  result = sort_into_ascending_order(fake_created_at_list(["c", "a", "b"]))
  issues = for issue <- result, do: issue["created_at"]
  assert issues == ~w{a b c}
end

defp fake_created_at_list(values) do
```

Prepared exclusively for Richard Bishop

```
  data = for value <- values,
         do: [{"created_at", value}, {"other_data", "xxx"} ]
  convert_to_list_of_hashdicts data
end
```

Update the imports line at the top of the test:

```
import Issues.CLI, only: [ parse_args: 1,
                           sort_into_ascending_order: 1,
                           convert_to_list_of_hashdicts: 1 ]
```

and run it.

```
$ mix test
.....

Finished in 0.00 seconds
5 tests, 0 failures
```

Mmmmmm, lookin' fine.

Transformation: Take First N Items

Our next transformation is to extract the first *count* entries from the list. Rather than write a function, we'll use the built-in `Enum.take`.

```
def process({user, project, count}) do
  Issues.GithubIssues.fetch(user, project)
      |> decode_response
      |> convert_to_list_of_hashdicts
      |> sort_into_ascending_order
      |> Enum.take(count)
end
```

Your turn…

➤ *Exercise: OrganizingAProject-3*

Bring your version of this project in line with the code here.

➤ *Exercise: OrganizingAProject-4*

(Tricky) Before reading the next section, see if you can write the code to format the data into columns, like the sample output at the start of the chapter. This is probably the longest piece of Elixir code you'll have written. Try to do it without using `if` or `cond`.

Transformation: Format the Table

All that's left from our design is creating the formatted table. A nice interface would be

Prepared exclusively for Richard Bishop

```
def process({user, project, count}) do
  Issues.GithubIssues.fetch(user, project)
      |> decode_response
      |> convert_to_list_of_hashdicts
      |> sort_into_ascending_order
      |> Enum.take(count)
➤     |> print_table_for_columns(["number", "created_at", "title"])
end
```

We pass the formatter the list of columns to include in the table, and it writes the table to standard output.

The formatter doesn't actually add any new project or design related techniques, so we'll just list it here.

project/4/issues/lib/issues/table_formatter.ex

```
defmodule Issues.TableFormatter do

  import Enum, only: [ each: 2, map: 2, map_join: 3, max: 1 ]

  def print_table_for_columns(rows, headers) do
    data_by_columns = split_into_columns(rows, headers)
    column_widths   = widths_of(data_by_columns)
    format          = format_for(column_widths)

    puts_one_line_in_columns headers, format
    IO.puts                  separator(column_widths)
    puts_in_columns          data_by_columns, format
  end

  def split_into_columns(rows, headers) do
    for header <- headers do
      for row <- rows, do: printable(row[header])
    end
  end

  def printable(str) when is_binary(str), do: str
  def printable(str), do: to_string(str)

  def widths_of(columns) do
    for column <- columns, do: column |> map(&String.length/1) |> max
  end

  def format_for(column_widths) do
    map_join(column_widths, " | ", fn width -> "~-#{width}s" end) <> "~n"
  end

  def separator(column_widths) do
    map_join(column_widths, "-+-", fn width -> List.duplicate("-", width) end)
  end
```

Prepared exclusively for Richard Bishop

```
  def puts_in_columns(data_by_columns, format) do
    data_by_columns
    |> List.zip
    |> map(&Tuple.to_list/1)
    |> each(&puts_one_line_in_columns(&1, format))
  end

  def puts_one_line_in_columns(fields, format) do
    :io.format(format, fields)
  end

end
```

And here are the tests for it.

project/4/issues/test/table_formatter_test.exs

```
defmodule TableFormatterTest do

  use ExUnit.Case          # bring in the test functionality
  import ExUnit.CaptureIO  # And allow us to capture stuff sent to stdout

  alias Issues.TableFormatter, as: TF

  def simple_test_data do
    [
     [ c1: "r1 c1", c2: "r1 c2",  c3: "r1 c3", c4: "r1+++c4" ],
     [ c1: "r2 c1", c2: "r2 c2",  c3: "r2 c3", c4: "r2 c4" ],
     [ c1: "r3 c1", c2: "r3 c2",  c3: "r3 c3", c4: "r3 c4" ],
     [ c1: "r4 c1", c2: "r4++c2", c3: "r4 c3", c4: "r4 c4" ],
    ]
  end

  def headers, do: [ :c1, :c2, :c4 ]

  def split_with_three_columns,
    do: TF.split_into_columns(simple_test_data, headers)

  test "split_into_columns" do
    columns = split_with_three_columns
    assert     length(columns) == length(headers)
    assert List.first(columns) == ["r1 c1", "r2 c1", "r3 c1", "r4 c1"]
    assert  List.last(columns) == ["r1+++c4", "r2 c4", "r3 c4", "r4 c4"]
  end

  test "column_widths" do
    columns = split_with_three_columns
    widths = TF.widths_of(columns)
    assert widths == [ 5, 6, 7 ]
  end

  test "correct format string returned" do
```

Prepared exclusively for Richard Bishop

```
    assert TF.format_for([9, 10, 11]) == "~-9s | ~-10s | ~-11s~n"
  end

  test "Output is correct" do
    result = capture_io fn ->
      TF.print_table_for_columns(simple_test_data, headers)
    end
    assert result == """
    c1    | c2     | c4
    ------+--------+--------
    r1 c1 | r1 c2  | r1+++c4
    r2 c1 | r2 c2  | r2 c4
    r3 c1 | r3 c2  | r3 c4
    r4 c1 | r4++c2 | r4 c4
    """
  end
end
```

Rather than clutter the process function in the CLI module with a long module name, I chose to use import to make the print function available without a module qualifier. This goes near the top of cli.ex.

project/5/issues/lib/issues/cli.ex

```
import Issues.TableFormatter, only: [ print_table_for_columns: 2 ]
```

This code also uses a wonderful Elixir testing feature. By importing ExUnit.CaptureIO, we get access to the capture_io function. This runs the code passed to it, but captures anything written to standard output, returning it as a string.

Task: Make a command line executable

Although we can run our code by calling the run function via mix, it isn't really friendly for other users. So let's create something we can run from the command line.

Mix can package our code, along with its dependencies, into a single file that can be run on any Unix-based platform. This makes use of Erlang's escript utility, which can run precompiled programs stored as a zip archive. In our case, the program will be run as issues.

When escript runs a program, it looks in your mix.exs file for the option escript. This should return a keyword list of escript configuration settings. The most important of these is main_module:, which must be set to the name of a module containing a main function. It passes the command line arguments to this main function as a list of character lists (not binaries). As this seems to be a command-line concern, we'll put the main function in Issues.CLI. Here's the update to mix.exs:

Prepared exclusively for Richard Bishop

project/4/issues/mix.exs

```
defmodule Issues.Mixfile do
  use Mix.Project

  def project do
    [ app:     :issues,
      version: "0.0.1",
      elixir:  ">= 0.0.0",
➤     escript: escript_config,
      deps:    deps ]
  end

  # Configuration for the OTP application
  def application do
    [
      applications: [ :httpotion, :jsx ]
    ]
  end

  defp deps do
    [
      { :httpotion,  github: "myfreeweb/httpotion" },
      { :jsx,        github: "talentdeficit/jsx"  }
    ]
  end

➤   defp escript_config do
➤     [ main_module: Issues.CLI ]
➤   end
end
```

Now let's add a `main` function to our CLI. If fact, all we need to do is rename the existing `run` function:

project/4/issues/lib/issues/cli.ex

```
def main(argv) do
  argv
    |> parse_args
    |> process
end
```

Then we package our program using mix:

```
$ mix escript.build
Generated escript issues
```

Now we can run the app locally. We can also send it to run on any computer that has Erlang installed.

```
$ ./issues dynamo dynamo 3
nu | created_at           | title
```

Prepared exclusively for Richard Bishop

```
---+---------------------+--------------------------------------------
7  | 2012-09-15T10:48:11Z | Should have a websocket demo ?
45 | 2013-02-23T14:40:56Z | Raise an error if a layout can't be found.
46 | 2013-02-24T21:42:23Z | Force mix dynamo CamelCase to raise exception
```

Task: Test The Comments

When I document my functions, I like to include examples of the function being used—comments saying things such as "feed it these arguments, and you'll get this result." In the Elixir world, a common way to do this is to show the function being used in an iex session.

Here's an example. Our TableFormatter contains a number of self-contained functions that we can document.

project/5/issues/lib/issues/table_formatter.ex

```
defmodule Issues.TableFormatter do

  import Enum, only: [ each: 2, map: 2, map_join: 3, max: 1 ]

  @doc """
  Takes a list of row data, where each row is a HashDict, and a list of
  headers. Prints a table to STDOUT of the data from each row
  identified by each header. That is, each header identifies a column,
  and those columns are extracted and printed from the rows.

  We calculate the width of each column to fit the longest element
  in that column.
  """
  def print_table_for_columns(rows, headers) do
    data_by_columns = split_into_columns(rows, headers)
    column_widths   = widths_of(data_by_columns)
    format          = format_for(column_widths)

    puts_one_line_in_columns headers, format
    IO.puts                  separator(column_widths)
    puts_in_columns          data_by_columns, format
  end

  @doc """
  Given a list of rows, where each row contains a keyed list
  of columns, return a list containing lists of the data in
  each column. The `headers` parameter contains the
  list of columns to extract

  ## Example

      iex> list = [Enum.into([{"a", "1"},{"b", "2"},{"c", "3"}], HashDict.new),
      ...>         Enum.into([{"a", "4"},{"b", "5"},{"c", "6"}], HashDict.new)]
      iex> Issues.TableFormatter.split_into_columns(list, [ "a", "b", "c" ])
```

Prepared exclusively for Richard Bishop

```
    [ ["1", "4"], ["2", "5"], ["3", "6"] ]

"""
def split_into_columns(rows, headers) do
  for header <- headers do
    for row <- rows, do: printable(row[header])
  end
end

@doc """
Return a binary (string) version of our parameter.
## Examples
    iex> Issues.TableFormatter.printable("a")
    "a"
    iex> Issues.TableFormatter.printable(99)
    "99"
"""
def printable(str) when is_binary(str), do: str
def printable(str), do: to_string(str)

@doc """
Given a list containing sublists, where each sublist contains the data for
a column, return a list containing the maximum width of each column

## Example
    iex> data = [ [ "cat", "wombat", "elk"], ["mongoose", "ant", "gnu"]]
    iex> Issues.TableFormatter.widths_of(data)
    [ 6, 8 ]
"""
def widths_of(columns) do
  for column <- columns, do: column |> map(&String.length/1) |> max
end

@doc """
Return a format string that hard codes the widths of a set of columns.
We put `" | "` between each column.

## Example
    iex> widths = [5,6,99]
    iex> Issues.TableFormatter.format_for(widths)
    "~-5s | ~-6s | ~-99s~n"
"""
def format_for(column_widths) do
  map_join(column_widths, " | ", fn width -> "~-#{width}s" end) <> "~n"
end

@doc """
Generate the line that goes below the column headings. It is a string of
hyphens, with + signs where the vertical bar between the columns goes.
```

Prepared exclusively for Richard Bishop

```
  ## Example
        iex> widths = [5,6,9]
        iex> Issues.TableFormatter.separator(widths)
        "------+--------+----------"
  """
  def separator(column_widths) do
    map_join(column_widths, "-+-", fn width -> List.duplicate("-", width) end)
  end

  @doc """
  Given a list containing rows of data, a list containing the header selectors,
  and a format string, write the extracted data under control of the format string.
  """
  def puts_in_columns(data_by_columns, format) do
    data_by_columns
    |> List.zip
    |> map(&Tuple.to_list/1)
    |> each(&puts_one_line_in_columns(&1, format))
  end

  def puts_one_line_in_columns(fields, format) do
    :io.format(format, fields)
  end

end
```

Note how some of the documentation contains sample iex sessions.

Now we write a test that validates that each of the iex sessions actually returns the values shown in the @doc string. We create a new test file, test/doc_test.exs containing:

project/5/issues/test/doc_test.exs

```
defmodule DocTest do
  use ExUnit.Case

  doctest Issues.TableFormatter

end
```

We can now run this:

```
$ mix test test/doc_test.exs
......

Finished in 0.00 seconds
5 tests, 0 failures
```

And, of course, these tests are integrated into the overall test suite:

```
$ mix test
..............
```

Prepared exclusively for Richard Bishop

```
Finished in 0.01 seconds
13 tests, 0 failures
```

Let's force an error to see what happens:

```
@doc """
Return a binary (string) version of our parameter.

## Examples

    iex> Issues.TableFormatter.printable("a")
    "a"
    iex> Issues.TableFormatter.printable(99)
    "99.0"
"""
def printable(str) when is_binary(str), do: str
def printable(str), do: to_string(str)
```

And run the tests again:

```
$  mix test test/doc_test.exs
.........

  1) test doc at Issues.TableFormatter.printable/1 (3) (DocTest)
     Doctest failed
     code: " Issues.TableFormatter.printable(99) should equal \"99.0\""
     lhs:  "\"99\""
     stacktrace:
       lib/issues/table_formatter.ex:52: Issues.TableFormatter (module)
....
Finished in 0.01 seconds
6 tests, 1 failures
```

Task: Create Project Documentation

Java has JavaDoc, Ruby has RDoc, and Elixir has ExDoc, a documentation tool that describes your project, showing the modules, the things defined in them, along with any documentation you've written for them.

Using it is easy. First, add the ExDoc dependency to your mix.exs file.

project/5/issues/mix.exs

```
defp deps do
  [
    { :httpotion,  github: "myfreeweb/httpotion" },
    { :jsx,        github: "talentdeficit/jsx"  },
    { :ex_doc,     github: "elixir-lang/ex_doc" }
  ]
end
```

Prepared exclusively for Richard Bishop

While you're in the mix.exs, you can add a project name and (if your project is in GitHub) a URL. The latter allows ExDoc to provide live links to your source code. These parameters go in the project function:

```
def project do
  [ app: :issues,
    version: "0.0.1",
    name: "Issues",
    source_url: "https://github.com/pragdave/issues",
    deps: deps ]
end
```

Then run mix deps.get.

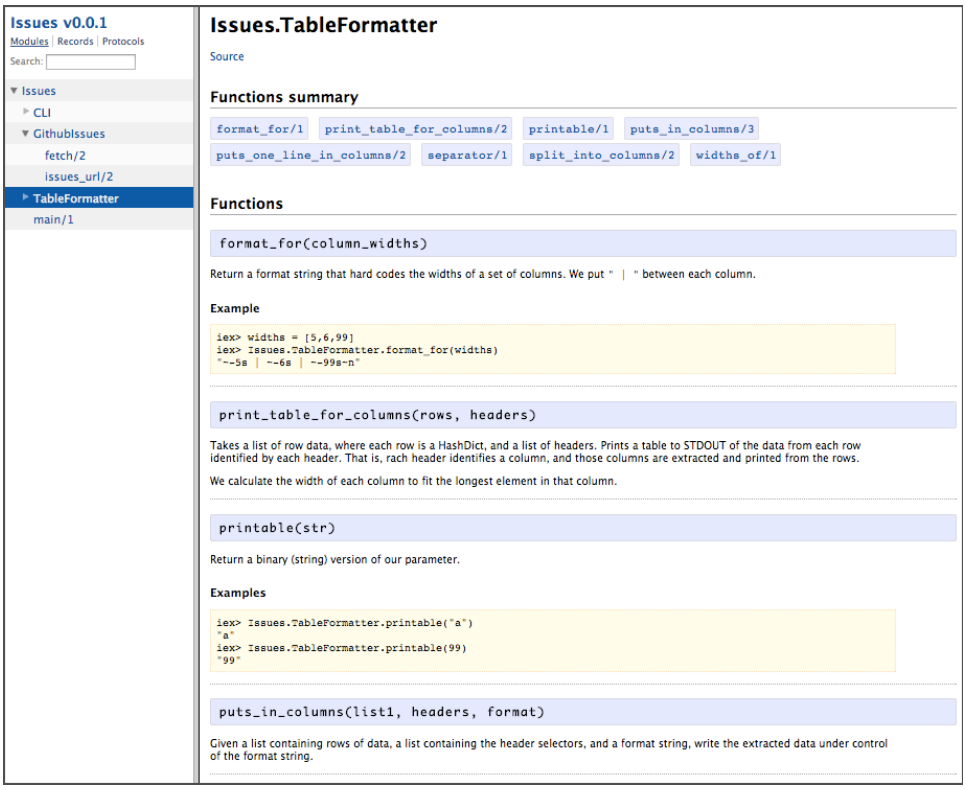
To generate the documentation, just run

```
$ mix docs
Docs generated with success.
Open up docs/index.html in your browser to read them.
```

The first time you run it, this will install ExDoc. That involves compiling some C code, so you'll need a development environment on your machine.

Open docs/index.html in your browser, then use the sidebar on the left to search or drilldown through your modules. Here's what I see for the start of the documentation for TableFormatter.

Prepared exclusively for Richard Bishop



And that's it. The full project is in the source download at project/5/issues.

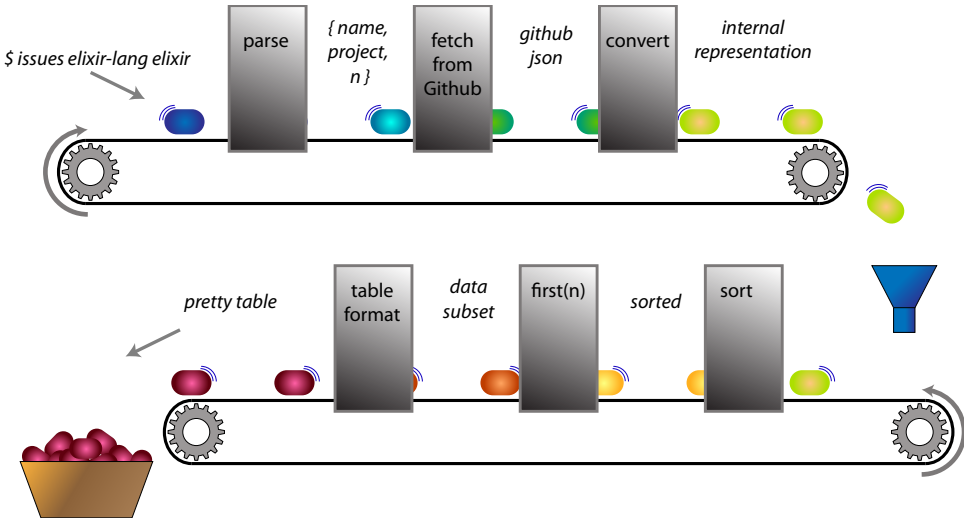
What We've Just Seen

I wanted to show you how Elixir projects are written—the tools we use and the processes we follow. I wanted to show how lots of small functions can transform data, and how specifying that transformation acts as an outline for the program. I also wanted to show how easy testing can be in Elixir.

But, mostly, I wanted to show how enjoyable Elixir development is, and how thinking about the world in terms of data and its transformation is a productive way to code.

Let's look at our original outline:

Prepared exclusively for Richard Bishop



And then at the CLI.process function.

```
def process({user, project, count}) do
  Issues.GithubIssues.fetch(user, project)
    |>  decode_response
    |>  convert_to_list_of_hashdicts
    |>  sort_into_ascending_order
    |>  Enum.take(count)
    |>  print_table_for_columns(["number", "created_at", "title"])
end
```

This is a cool way to code.

Your turn…

➤ *Exercise: OrganizingAProject-6*

In the US, NOAA provides hourly XML feeds[10] of conditions at 1,800 locations. For example, the feed for a small airport close to where I'm writing this is at http://w1.weather.gov/xml/current_obs/KDTO.xml.

Write an application that fetches this data, parses it, and displays it in a nice format.

(Hint: you might not have to download a library to handle XML parsing)

10. http://w1.weather.gov/xml/current_obs

Prepared exclusively for Richard Bishop

Part II

Concurrent Programming

You want to write concurrent programs. That's probably why you're reading this book.

Let's look at Elixir's actor-based concurrency model. And then we'll dig in to OTP, the Erlang management architecture that helps you structure your code from humble colo'd hobby project to massive data centered web scale.

Prepared exclusively for Richard Bishop

CHAPTER 14

Working With Multiple Processes

One of the key features of Elixir is the idea of packaging code into small chunks that can be run independently and concurrently.

If you've come from a conventional programming language, this may worry you. Concurrent programming is "known" to be difficult, and there's a performance penalty to pay when you create lots of processes.

Elixir doesn't have these issues, thanks to the architecture of the Erlang VM on which it runs.

Elixir uses the *actor* model of concurrency. An actor is an independent process that shares nothing with any other process. You can `spawn` new processes, send them messages, and `receive` messages back. And that's it.[1]

In the past, you may have had to use threads or operating system processes to achieve concurrency. Each time, you probably felt you were opening Pandora's box—there was so much that could go wrong. But that worry just evaporates in Elixir. In fact, Elixir developers are so comfortable creating new processes, they'll often do it at times where you'd have created an *object* in a language such as Java.

One more thing—when we talk about processes in Elixir, we are not talking about native operating system processes. These are too slow and bulky. Instead, Elixir uses process support in Erlang. These processes will run across all your CPUs (just like native processes), but they have very little overhead. As we'll see a bit later, it's very easy to create hundreds of thousands of Elixir processes on even a modest computer.

1. Apart from some details about error handling and monitoring, which we cover later.

Prepared exclusively for Richard Bishop

A Simple Process

Here's a module that defines a function that we'd like to run as a separate process.

spawn/spawn-basic.ex

```
defmodule SpawnBasic do
  def greet do
    IO.puts "Hello"
  end
end
```

Yup, that's it. There's nothing special—it's just regular code.

Let's fire up iex and play.

```
iex> c("spawn-basic.ex")
[SpawnBasic]
```

First, let's call it as a regular function:

```
iex> SpawnBasic.greet
Hello
:ok
```

Now, let's run it in a separate process:

```
iex> spawn(SpawnBasic, :greet, [])
Hello
#PID<0.42.0>
```

The `spawn` function kicks off a new process. It comes in many forms, but the two simplest ones let you run an anonymous function and to run a named function in a module, passing a list of arguments (that's the one we used here).

The `spawn` returns a *Process Identifier*, normally called a pid. This uniquely identifies the process it creates, potentially from any other process in the world (although here it's just unique in our application).

When we call `spawn`, it creates a new process to run the code we specify. We don't know exactly when it will execute—we only know that it is eligible to run.

In this example, we can see that our function ran and output "Hello" prior to iex reporting the pid returned by `spawn`. But you can't rely on this. Instead, you'll use messages to synchronize the activity of your processes.

Prepared exclusively for Richard Bishop

Sending Messages Between Processes

Let's rewrite our example to use messages. The top level will send greet a message containing a string, and the greet function will respond with a greeting containing that message.

In Elixir, we send a message using the send function. It takes a pid and the message to send (an Elixir value, which we also call a *term*) on the right. You can send anything you want, but most Elixir developers seem to use atoms and tuples.

We wait for messages using receive. In a way, this acts just like case, with the message body as the parameter. Inside the block associated with the receive call you can specify any number of patterns and associated actions. Just as with case the action associated with the first pattern that matches the function is run.

Here's the updated version of our greet function.

spawn/spawn1.ex

```
defmodule Spawn1 do
  def greet do
    receive do
      {sender, msg} ->
        send sender, { :ok, "Hello #{msg}" }
    end
  end
end

# here's a client
pid = spawn(Spawn1, :greet, [])
send pid, {self, "World!"}

receive do
  {:ok, message} ->
    IO.puts message
end
```

The function uses receive to wait for a message, and then matches the message in the block. In this case, the only pattern is a two-element tuple, where the first element is the pid of the original sender and the second the message. In the corresponding action, we use send sender, ... to send a formatted string back to the original message sender. We package that string into a tuple, with :ok as its first element.

Outside the module, we call spawn to create a process, and send it a tuple:

```
send pid, { self, "World!" }
```

Prepared exclusively for Richard Bishop

The function self returns the pid of its caller. Here we use it to pass our pid to the greet function, so it will know where to send the response.

We then wait for a response. Notice that we do a pattern match on {:ok, message}, extracting the second element of the tuple, which contains the actual text.

We can run this in iex:

```
iex> c("spawn1.ex")
Hello World!
[Spawn1]
```

Very cool. The text was sent, and greet responded back with the full greeting.

Let's try sending a second message.

spawn/spawn2.ex

```
defmodule Spawn2 do
  def greet do
    receive do
      {sender, msg} ->
        send sender, { :ok, "Hello #{msg}" }
    end
  end
end

# here's a client
pid = spawn(Spawn2, :greet, [])

send pid, {self, "World!"}

receive do
  {:ok, message} ->
    IO.puts message
end

send pid, {self, "Kermit!"}
receive do
  {:ok, message} ->
    IO.puts message
end
```

Run it in iex:

```
iex> c("spawn2.ex")
Hello World!
.... just sits there ....
```

The first message is sent back, but the second is nowhere to be seen. What's worse, iex just hangs, and we have to use ^C to get out of it.

Prepared exclusively for Richard Bishop

That's because our greet function only handles a single message. Once it has processed the receive, it exits. As a result, the second message we send it is never processed. The second receive at the top level then just hangs, waiting for a response that will never come.

Let's at least fix the hanging part. We can tell receive that we want to time out if a response is not received in so many milliseconds. This uses a pseudo-pattern called after.

spawn/spawn3.ex

```
defmodule Spawn3 do
  def greet do
    receive do
      {sender, msg} ->
        send sender, { :ok, "Hello #{msg}" }
    end
  end
end

# here's a client
pid = spawn(Spawn3, :greet, [])

send pid, {self, "World!"}
receive do
  {:ok, message} ->
    IO.puts message
end

send pid, {self, "Kermit!"}
receive do
  {:ok, message} ->
    IO.puts message
➤ after 500 ->
➤   IO.puts "The greeter has gone away"
end

iex> c("spawn3.ex")
Hello World!
... short pause ...
The greeter has gone away
[Spawn3]
```

But how would we make our greet function handle multiple messages? Our natural reaction is to make it loop, doing a receive on each iteration. Elixir doesn't have loops, but it does have recursion.

spawn/spawn4.ex

```
defmodule Spawn4 do
  def greet do
    receive do
```

Prepared exclusively for Richard Bishop

```
      {sender, msg} ->
        send sender, { :ok, "Hello #{msg}" }
➤       greet
    end
  end
end

# here's a client
pid = spawn(Spawn4, :greet, [])

send pid, {self, "World!"}
receive do
  {:ok, message} ->
    IO.puts message
end

send pid, {self, "Kermit!"}
receive do
  {:ok, message} ->
    IO.puts message
  after 500 ->
    IO.puts "The greeter has gone away"
end
```

Run this, and both messages are processed.

```
iex> c("spawn4.ex")
Hello World!
Hello Kermit!
[Spawn4]
```

Recursion, Looping, and the Stack

The recursive greet function might have worried you a little. Every time it receives a message, it ends up calling itself. In many languages, that adds a new frame to the stack. After a large number of messages, you might run out of memory.

This doesn't happen in Elixir, as it implements something called *tail call optimization*. If the very last thing a function does is to call itself, then there's actually no need to make the call. Instead, the runtime can simply jump back to the start of the function. If the recursive call has arguments, then these replace the original parameters as the loop occurs.

But beware—the recursive call *must* be the very last thing executed. For example, the following code is not tail recursive.

```
def factorial(0), do: 1
def factorial(n), do: n * factorial(n-1)
```

Although the recursive call is physically the last thing in the function, it is not the last thing executed. The function has to multiply the value it returns by 'n'.

Prepared exclusively for Richard Bishop

To make it tail recursive, we need to move the multiplication into the recursive call, and this means adding an accumulator:

spawn/fact_tr.ex

```
defmodule TailRecursive do

  def factorial(n), do: _fact(n, 1)

  defp _fact(0, acc), do: acc
  defp _fact(n, acc), do: _fact(n-1, acc*n)

end
```

Process Overhead

At the start of the chapter, I somewhat cavalierly said that Elixir processes were very low overhead. Now it is time to back that up. Let's write some code that creates *n* processes. The first will send a number to the second. It will increment that number and pass it to the third. This will continue until we get to the last process, which will pass the number back to the top level.

Here's the code.

spawn/chain.exs

```
defmodule Chain do

  def counter(next_pid) do
    receive do
      n ->
        send next_pid, n + 1
    end
  end

  def create_processes(n) do
    last = Enum.reduce 1..n, self,
             fn (_,send_to) ->
               spawn(Chain, :counter, [send_to])
             end

    # start the count by sending
    send last, 0

    # and wait for the result to come back to us
    receive do
      final_answer when is_integer(final_answer) ->
        "Result is #{inspect(final_answer)}"
    end
  end
```

Prepared exclusively for Richard Bishop

```
25
-   def run(n) do
-     IO.puts inspect :timer.tc(Chain, :create_processes, [n])
-   end
-
30 end
```

The counter function on line 3 is the code that will be run in separate processes. It is passed the pid of the next process in the chain. When it receives a number, it increments it and sends it on to that next process.

The create_processes function is probably the densest piece of Elixir we've encountered so far. Let's break it down.

It is passed the number of processes to create. Each process has to be passed the pid of the previous process, so that it knows who to send the updated number to. All this is done on line 12.

The reduce call will iterate over the range 1..n. Each time around, it will pass an accumulator as the second parameter to its function. We set the initial value of that accumulator to self, our pid.

In the function, we spawn a new process that runs the counter function, using the third parameter of spawn to pass in the current value of the accumulator (initially self). The value returned by spawn is the pid of the newly created process, which becomes the value of the accumulator for the next iteration.

Putting it another way, each time we spawn a new process, we pass it the previous process's pid in the send_to parameter.

The value returned by the reduce function is the final value of the accumulator, which is the pid of the last process created.

On the next line we set the ball rolling by passing zero to the last process. It will increment it and pass 1 to the second to last process. This goes on until the very first process we created passes the result back to us. We use the receive block to capture this, and format it into a nice message.

Our receive block contains a new feature. We've already seen how guard clauses can constrain pattern matching and function calling. The same guard clauses can be used to qualify the pattern in a receive block.

Why do we need this, though? It turns out there's a bug in some versions of Elixir.[2] When you compile and run a program using iex -S mix, there's a residual message left lying around from the compilation process (it records the

2. https://github.com/elixir-lang/elixir/issues/1050

Prepared exclusively for Richard Bishop

termination of a process). We ignore that message by telling the receive clause that we're only interested in simple integers.

The run function starts the whole thing off. It uses a built-in Erlang library, tc, which can be used to time the execution of a function. We pass it the module, name, and parameters, and it responds with a tuple. The first element is the execution time in microseconds and the second is the result returned by the function.

We'll run this code from the command line, rather from iex. (You'll see why in a second.) These results are on my 2011 Macbook Air (2.13Ghz Core 2 Duo and 4Gb ram).

```
$ elixir -r chain.exs -e "Chain.run(10)"
{3175,"Result is 10"}
```

We asked it to run 10 processes, and it came back in 3.175 ms. The answer looks correct. Let's try 100 processes.

```
$ elixir -r chain.exs -e "Chain.run(100)"
{3584,"Result is 100"}
```

Only a small increase in the time. There's probably some startup latency on the first process creation. Onward! Let's try 1000.

```
$ elixir -r chain.exs -e "Chain.run(1000)"
{8838,"Result is 1000"}
```

Now 10,000

```
$ elixir -r chain.exs -e "Chain.run(10000)"
{76550,"Result is 10000"}
```

Ten thousand processes created and executed in 77 ms. Let's push the boat out and try for 400,000.

```
$ elixir -r chain.exs -e "Chain.run(400_000)"
=ERROR REPORT==== 25-Apr-2013::15:16:14 ===
Too many processes
** (SystemLimitError) a system limit has been reached
```

It looks like the virtual machine won't support 400,000 processes. Fortunately this is not a hard limit—we just bumped into a default value. We can increase this using the VM's +P parameter. And we pass this parameter to the VM using the --erl parameter to elixir. (This is why I chose to run from the command line.)

```
$ elixir --erl "+P 1000000"  -r chain.exs -e "Chain.run(400_000)"
{3210704,"Result is 400000"}
```

Prepared exclusively for Richard Bishop

One last run, this time with 1,000,000 processes.

```
$ elixir --erl "+P 1000000"  -r chain.exs -e "Chain.run(1_000_000)"
{7225292,"Result is 1000000"}
```

We ran one million processes (sequentially) in about 7 seconds. This kind of performance is stunning, and it changes the way we design code. We can now create hundreds of little helper processes. And each process can contain its own state—in a way, processes in Elixir are like objects in an object-oriented system (but they have a better sense of humor).

Your turn…

➤ *Exercise: WorkingWithMultipleProcesses-1*
Run this code on your machine. See if you get comparable results.

➤ *Exercise: WorkingWithMultipleProcesses-2*
Write a program that spawns two processes, and then passes each a unique token (for example “fred” and “betty”). Have them send the tokens back.

- Is the order that the replies are received deterministic in theory? In practice?
- If either answer is no, how could you make it so?

When Processes Die

Who gets told when a process dies? By default, no one. Obviously, the VM knows, and can report it to the console, but your code will be oblivious unless you explicitly tell Elixir that you want to get involved.

First, here’s the default case. We spawn a function that uses the Erlang timer library to sleep for 500 ms. It then exits with a status of 99.

The code that spawns it sits in a receive. If it receives a message, it reports the fact, otherwise after one second it lets us know that nothing happened.

spawn/link1.exs
```
import :timer, only: [ sleep: 1 ]

defmodule Link1 do
  def sad_function do
    sleep 500
    exit(:boom)
  end

  def run do
```

Prepared exclusively for Richard Bishop

```
    Process.flag(:trap_exit, true)

    spawn(Link1, :sad_function, [])

    receive do
      msg ->
        IO.puts "MESSAGE RECEIVED: #{inspect msg}"
    after 1000 ->
        IO.puts "Nothing happened as far as I am concerned"
    end
  end
end

Link1.run
```

(You might want to think about how you'd have written this in your old programming language.)

We can run this from the console:

```
$ elixir -r link1.exs  -e Link1.run
Nothing happened as far as I am concerned
```

As far as the top level was concerned, there was no activity caused by the spawned process exiting.

Linking Two Processes

If we want two processes to share in each other's pain, we can *link* them. When processes are linked, each can receive information when the other exits. The spawn_link call spawns a process and links it to the caller in one operation.

spawn/link2.exs

```
import :timer, only: [ sleep: 1 ]

defmodule Link2 do
  def sad_function do
    sleep 500
    exit(:boom)
  end

  def run do
➤   spawn_link(Link2, :sad_function, [])

    receive do
      msg ->
        IO.puts "MESSAGE RECEIVED: #{inspect msg}"
    after 1000 ->
        IO.puts "Nothing happened as far as I am concerned"
    end
```

Prepared exclusively for Richard Bishop

```
  end
end

Link2.run
```

The runtime reports the abnormal termination:

```
$ elixir -r link2.exs  -e Link2.run
** (EXIT from #PID<0.35.0>) :boom
```

So our child process died, and it killed the entire application. That'sthe default bahviour of linked processes—when one exits abnormally, it kills the other.

What if you want to handle the death of another process? Well, you probably don't want to do this. Elixir uses the OTP framework for constructing process trees, and OTP includes the concept of process supervision. An incredible amount of effort has been spent getting this right, so I recommend using it most of the time. (We cover this a few chapters from now.)

However, you can tell Elixir to convert the exit signals from a linked process into a message that you can handle. Do this by *trapping the exit*.

spawn/link3.exs

```
import :timer, only: [ sleep: 1 ]

defmodule Link3 do
  def sad_function do
    sleep 500
    exit(:boom)
  end

  def run do
➤   Process.flag(:trap_exit, true)
    spawn_link(Link2, :sad_function, [])

    receive do
      msg ->
        IO.puts "MESSAGE RECEIVED: #{inspect msg}"
    after 1000 ->
        IO.puts "Nothing happened as far as I am concerned"
    end
  end
end

Link3.run
```

This time we see an :EXIT message when the spawned process terminates:

```
$ elixir -r link2.exs  -e Link3.run
MESSAGE RECEIVED: {:EXIT, #PID<0.41.0>, :boom}
```

Prepared exclusively for Richard Bishop

It doesn't matter why a process exits—it may simply finish processing, it may explicitly exit, or it may raise an exception—the same :EXIT message is received. Following an error, however, it contains details of what went wrong.

Monitoring a Process

Linking joins the calling process and another process—each receives notifications about the other. By contrast, *monitoring* lets a process spawn another and be notified of its termination, but without the reverse notification—it is one way only.

When you monitor a process, you receive a :DOWN message when it exits, fails, or if it doesn't exist.

You can turn on monitoring when you spawn a process using spawn_monitor, or you can monitor an existing process using Process.monitor.[3]

spawn/monitor1.exs

```
import :timer, only: [ sleep: 1 ]

defmodule Monitor1 do
  def sad_method do
    sleep 500
    exit(:boom)
  end

  def run do
➤   res = spawn_monitor(Monitor1, :sad_method, [])
    IO.puts inspect res

    receive do
      msg ->
        IO.puts "MESSAGE RECEIVED: #{inspect msg}"
    after 1000 ->
        IO.puts "Nothing happened as far as I am concerned"
    end
  end
end

Monitor1.run
```

Run it, and the results are similar to the spawn_link version:

```
$ elixir -r monitor1.exs -e Monitor1.run
```

3. If you monitor (or link to) an existing process, there is a potential race condition—if the other process dies before your monitor call completes, you may not receive a notification. The spawn_link and spawn_monitor versions are atomic, however, so you'll always catch a failure.

Prepared exclusively for Richard Bishop

```
{#PID<0.37.0>,#Reference<0.0.0.53>}
MESSAGE RECEIVED: {:DOWN,#Reference<0.0.0.53>,:process,#PID<0.37.0>,:boom}
```

(The Reference record in the message is the identity of the monitor that was created. It is also returned, along with the pid, by the spawn_monitor call.)

So, when do you use links, and when should you choose monitors?

It depends on the semantics of your processes. If the intent is that a failure in one process should also terminate another, then you need links. If, instead, you need to know when another process exits for any reason, then choose monitors.

Your turn…

The Erlang function timer.sleep(time_in_ms) suspends the current process for a given time. You might want to use it to force some scenarios in the following exercises. The key with the exercises is to get used to the different reports that you'll see when you're developing code.

- *Exercise: WorkingWithMultipleProcesses-3*
  Use spawn_link to start a process, and have that process send a message to the parent and then exit immediately. Meanwhile, sleep for 500ms in the parent, then receive as many messages as there are waiting. Trace what you receive. Does it matter that you weren't waiting for the notification from the child at the time it exited?

- *Exercise: WorkingWithMultipleProcesses-4*
  Do the same, but have the child raise an exception. What difference do you see in the tracing.

- *Exercise: WorkingWithMultipleProcesses-5*
  Repeat the two, changing spawn_link to spawn_monitor.

Parallel Map—The Hello World of Erlang

Devin Torres reminded me that every book in the Erlang space must, by law, include a definition of a parallel map function. Regular map returns the list that results from applying a function to each element of a collection. The parallel version does the same, but it applies the function to each element in a separate process.

spawn/pmap.exs

```
defmodule Parallel do
  def pmap(collection, fun) do
```

Prepared exclusively for Richard Bishop

```
    me = self

    collection
  |>
    Enum.map(fn (elem) ->
       spawn_link fn -> (send me, { self, fun.(elem) }) end
     end)
  |>
    Enum.map(fn (pid) ->
      receive do { ^pid, result } -> result end
    end)
  end
end
```

Our method contains two transformations (look for the |> operator). The first transformation takes the collection and maps it into a list of pids, where each pid in the list runs the given function on an individual element of the list. If the collection contains 1,000 items, we'll run 1,000 processes.

The second transformation takes the list of pids and converts it into the results returned by the processes corresponding to each pid in the list. Note how it uses ^pid in the receive block to get the result for each pid in turn. Without this we'd get back the results in random order.

But does it work?

```
iex> c("pmap.exs")
[Parallel]
iex> Parallel.pmap 1..10, &(&1 * &1)
[1,4,9,16,25,36,49,64,81,100]
```

That's pretty sweet.

But it actually gets better, as we'll see when we look at tasks and agents on page 233.

Your turn…

➤ *Exercise: WorkingWithMultipleProcesses-6*

In the pmap code, I assigned the value of self to the variable me at the top of the method, and then used me as the target of the message returned by the spawned processes. Why use a separate variable here?

➤ *Exercise: WorkingWithMultipleProcesses-7*

Change the ^pid in pmap to _pid. This means that the receive block will take responses in the order the processes send them. Now run the code again. Do you see any difference in the output? If you're like me, you don't, but the program clearly contains a bug. Are you scared by this? Can you find

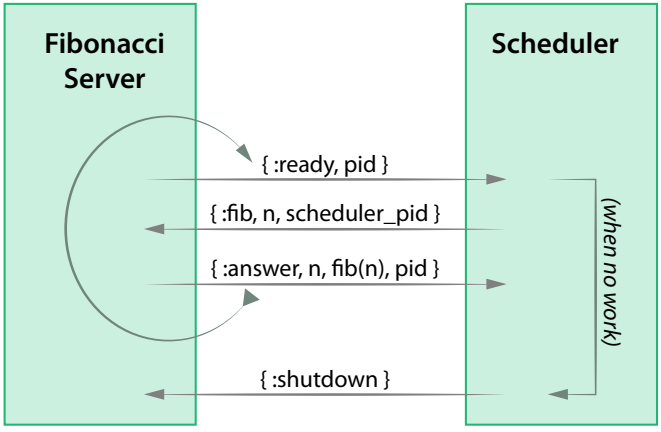
Prepared exclusively for Richard Bishop

a way to reveal the problem (perhaps by passing in a different function, or by sleeping, or increasing the number of processes)? Change it back to `^pid` and make sure the order is now correct.

A Fibonacci Server

Let's round out this chapter with an example program. Its task is to calculate *fib(n)* for a list of *n*, where *fib(n)* is the n^{th} Fibonacci number.[4] I chose this not because it is something we all do every day, but because the naive calculation of Fibonacci numbers 10 through 37 takes a measurable number of seconds on typical computers.

The twist is that we'll write our program to calculate different Fibonacci numbers in parallel. To do this, we'll write a trivial server process that does the calculation, and a scheduler that assigns work to a calculation process when it becomes free. The following diagram shows the message flow.



When the calculator is ready for the next number, it sends a `:ready` message to the scheduler. If there is still work to do, the scheduler sends it to the calculator in a `:fib` message, otherwise it sends the calculator a `:shutdown`. When a calculator receives a `:fib` message, it calculates the given Fibonacci number and returns it in an `:answer`. If it gets a `:shutdown`, it simply exits.

Here's the Fibonacci calculator module:

spawn/fib.exs

```
defmodule FibSolver do
```

4. The Fibonacci sequence starts 0, 1. Each subsequent number is the sum of the preceding two numbers in the sequence. http://en.wikipedia.org/wiki/Fibonacci_number

Prepared exclusively for Richard Bishop

```
  def fib(scheduler) do
    send scheduler, { :ready, self }
    receive do
      { :fib, n, client } ->
        send client, { :answer, n, fib_calc(n), self }
        fib(scheduler)
      { :shutdown } ->
        exit(:normal)
    end
  end

  # very inefficient, deliberately
  defp fib_calc(0), do: 1
  defp fib_calc(1), do: 1
  defp fib_calc(n), do: fib_calc(n-1) + fib_calc(n-2)
end
```

The public API is the fib function, which takes the scheduler pid. When it starts, it sends a :ready message to the scheduler, and then waits for a message back.

If it gets a :fib message, it calculates the answer and sends it back to the client. It then loops by calling itself recursively. This will send another :ready message, telling the client it is ready for more work.

If it gets a :shutdown it simply exits.

The Task Scheduler

The scheduler is a little more complex, as it is designed to handle both a varying number of server processes and an unknown amount of work.

spawn/fib.exs

```
defmodule Scheduler do

  def run(num_processes, module, func, to_calculate) do
    (1..num_processes)
    |> Enum.map(fn(_) -> spawn(module, func, [self]) end)
    |> schedule_processes(to_calculate, [])
  end

  defp schedule_processes(processes, queue, results) do
    receive do
      {:ready, pid} when length(queue) > 0 ->
        [ next | tail ] = queue
        send pid, {:fib, next, self}
        schedule_processes(processes, tail, results)

      {:ready, pid} ->
```

Prepared exclusively for Richard Bishop

```
        send pid, {:shutdown}
        if length(processes) > 1 do
          schedule_processes(List.delete(processes, pid), queue, results)
        else
          Enum.sort(results, fn {n1,_}, {n2,_}  -> n1 <= n2 end)
        end

      {:answer, number, result, _pid} ->
        schedule_processes(processes, queue, [ {number, result} | results ])

    end
  end
end
```

The public API for the scheduler is the run function. It receives the number of processes to spawn, the module and function to spawn, and a list of things to process. The scheduler is pleasantly ignorant of the actual task being performed.

Let's emphasize that last point. Nowhere does our scheduler know anything about Fibonacci numbers. Exactly the same code will happily manage processes working on DNA sequencing or cracking passwords.

The run function spawns the correct number of processes, and records their pids. It then calls the workhorse function, schedule_processes.

This function is basically a receive loop. If it gets a :ready message from a server, it sees if there is more work in the queue. If there is, it passes the next number to the calculator, and then recurses with one less number in the queue.

If the work queue is empty when it receives a :ready message, it sends a shutdown to the server. If this is the last process, then we're done, and it sorts the accumulated results. If it isn't the last process, it removes it from the list of processes and recurses to handle another message.

Finally, if it gets an :answer message, it records the answer in the result accumulator and recurses to handle the next message.

We drive the scheduler with the following code:

spawn/fib.exs

```
to_process = [ 37, 37, 37, 37, 37, 37 ]

Enum.each 1..10, fn num_processes ->
  {time, result} = :timer.tc(Scheduler, :run,
                             [num_processes, FibSolver, :fib, to_process])
  if num_processes == 1 do
    IO.puts inspect result
```

Prepared exclusively for Richard Bishop

```
    IO.puts "\n #   time (s)"
  end
  :io.format "~2B     ~.2f~n", [num_processes, time/1000000.0]
end
```

The to_process list contains the numbers we'll be passing to our fib servers. In our case, we give it the same number, 37, ten times. The intent here is to load each of our processors.

We run the code a total of 10 times, varying the number of spawned processes from 1 to 10. We use :timer.tc to determine the elapsed time of each iteration, reporting the result in seconds. The first time around the loop, we also display the numbers we calculated.

```
$ elixir fib.exs
[{37, 39088169}, {37, 39088169}, {37, 39088169}, {37, 39088169},
 {37, 39088169}, {37, 39088169}, {37, 39088169}, {37, 39088169},
 {37, 39088169}, {37, 39088169}]

 #   time (s)
 1     10.64
 2     6.05
 3     5.67
 4     4.97
 5     4.79
 6     5.28
 7     5.47
 8     5.24
 9     5.29
10     5.20
```

On my four-core system, we see a dramatic reduction in elapsed time when we increase the concurrency from one to two, and then small decreases until we hit 4 processes, then fairly flat performance after that. If you want to see similar results on systems with more cores, you'll need to increase the number of entries in the to_process list.

Your turn…

➤ *Exercise: WorkingWithMultipleProcesses-8*

Run the Fibonacci code on your machine. Do you get comparable timings. If your machine has multiple cores and/or processors, do you see improvements in the timing as we increase the concurrency of the application?

Prepared exclusively for Richard Bishop

➤ *Exercise: WorkingWithMultipleProcesses-9*

Use the same server code, but instead run a function that finds the number of times the word "cat" appears in each file in a given directory. Run one server process per file. The function File.ls! returns the names of files in a directory, and File.read! reads the contents of a file as a binary.

Run your code on a directory with a reasonable number of files (maybe 100 or so) so you can experiment with the effects of concurrency.

Agents—A Teaser of Things to Come

Our Fibonacci code is pathetically inefficient. To calculate fib(5), we calculate:

```
fib(5)
=  fib(4)                                    + fib(3)
=  fib(3)                    + fib(2)        + fib(2)          + fib(1)
=  fib(2)          + fib(1) + fib(1) + fib(0) + fib(1) + fib(0) + fib(1)
=  fib(1) + fib(0) + fib(1) + fib(1) + fib(0) + fib(1) + fib(0) + fib(1)
```

Look at all that duplication. If only we could cache all the intermediate values.

As you know, Exlir modules are basically buckets of functions—they cannot hold state. But processes can hold state. And Elixir comes with a library module called Agent that makes it easy to wrap a process containing state behind a nice module interface. Don't worry about the details of the code that follows—we cover agents and tasks on page 233, later in the book. For now, just see how processes are one of the tools we use to add persistence to Elixir code.[5]

spawn/fib_agent.exs

```
defmodule FibAgent do
  def start_link do
    cache = Enum.into([{0, 0}, {1, 1}], HashDict.new)
    Agent.start_link(fn -> cache end)
  end

  def fib(pid, n) when n >= 0 do
    Agent.get_and_update(pid, &do_fib(&1, n))
  end

  defp do_fib(cache, n) do
    if cached = cache[n] do
      {cached, cache}
    else
```

5. This code comes from a mailing list post by José Valim, written in response to some really ugly code I wrote.

Prepared exclusively for Richard Bishop

```
      {val, cache} = do_fib(cache, n - 1)
      result = val + cache[n-2]
      {result, Dict.put(cache, n, result)}
    end
  end
end

{:ok, agent} = FibAgent.start_link()
IO.puts FibAgent.fib(agent, 2000)
```

Let's run it:

```
$ elixir fib_agent.exs
42246963333923048787067256023414827825798528402506810980102801373143085843701
30707224123599639141511088446087538909603607640194711643596029271983312598737
32625355580260699158591522949245390499872225679531698287448247299226390183371
67780606070116154978867198798583114688708762645973690867228840236544222952433
47964480139515349562972087652656069529806499841977448720155612802665404554171
71788193032402520431208251681712 5
```

If we'd tried to calulate fib(2000) using the noncached version, the sun would grow to engulf the Earth while we were waiting for it to finish.

What's Next

So far we've been running our processes in the same VM. But if we're planning on taking over the world, we're going to need to be able to scale. And that means running on more than one machine.

The abstraction for this is the *node*, and that's the subject of the next chapter.

Prepared exclusively for Richard Bishop

CHAPTER 15

Nodes—The Key To Distributing Services

There's nothing mysterious about a *node*. It is simply a running Erlang VM. Throughout this book we've been running our code on a node.

The Erlang VM, called *Beam*, is more than a simple interpreter. It's like its own little operating system, running on top of your host operating system. It handles its own events, process scheduling, memory, naming services, and interprocess communication. And, on top of all that, a node can connect to other nodes—in the same computer, across a local lan, or across the Internet—and provide many of the same services across these connections that it provides to the processes it hosts locally.

Naming Nodes

So far, we haven't needed to give our node a name—we've only had one. If we ask Elixir what the current node is called, it'll give you a made-up name:

```
iex> Node.self
:nonode@nohost
```

We can set the name of a node when we start it. With iex, use either the `--name` or `--sname` options. The former sets a fully qualified name:

```
$ iex --name wibble@light-boy.local
iex(wibble@light-boy.local)> Node.self
:"wibble@light-boy.local"
```

and the latter sets a short name.

Note that the name that's returned is an atom—it's in quotes because it contains characters not allowed in a literal atom.

```
$ iex --sname wobble
iex(wobble@light-boy)> Node.self
:"wobble@light-boy"
```

Prepared exclusively for Richard Bishop

Note that in both cases the iex prompt contains the node's name along with my machine's name (light-boy).

Now I want to show you what happens when we have two nodes running. The easiest way to do this is to open two terminal windows and run a node in each. To represent these windows in the book, we'll show them stacked vertically.

So let's run a node called node_one in the top window and node_two in the lower one. We'll then use the Elixir Node module's list function to display a list of known nodes, then connect from one to the other.

Window #1

```
$ iex --sname node_one
iex(node_one@light-boy)>
```

Window #2

```
$ iex --sname node_two
iex(node_two@light-boy)> Node.list
[]
iex(node_two@light-boy)> Node.connect :"node_one@light-boy"
true
iex(node_two@light-boy)> Node.list
[:"node_one@light-boy"]
```

Initially, node_two doesn't know about any other nodes. But after we connect to node_one (notice that we pass an atom containing node one's node name), the list shows the other node. And, if we go back to node one, it will now know about node two.

```
iex(node_one@light-boy)> Node.list
[:"node_two@light-boy"]
```

Now that we have two nodes, we can try running some code. Over on node one, let's create an anonymous function that simply outputs the current node name.

```
iex(node_one@light-boy)> func = fn -> IO.inspect Node.self end
#Function<erl_eval.20.82930912>
```

We can run this with the spawn function.

```
iex(node_one@light-boy)> spawn(func)
#PID<0.59.0>
node_one@light-boy
```

Prepared exclusively for Richard Bishop

But spawn also lets you specify a node name. The process will be spawned on that node.

```
iex(node_one@light-boy)> Node.spawn(:"node_one@light-boy", func)
#PID<0.57.0>
node_one@light-boy
iex(node_one@light-boy)> Node.spawn(:"node_two@light-boy", func)
#PID<7393.48.0>
node_two@light-boy
```

We're running on node one. When we tell spawn to run on node_one@light-boy, we see two lines of output. The first is the pid returned by spawn, and the second line is the value of Node.self written by the function.

The second spawn is where it gets interesting. We pass it the name of node two and the same function. Again we get two lines of output. The first is the pid, and the second the node name. Notice the contents of the pid. The first field in a pid is the node number. When running on a local node, it's zero. But here we're running on a remote node, so that field has a positive value (7393). Then look at the output of the function. It reports that it is running on node two. I think that's pretty cool.

Now, you may have been expecting the output from the second spawn to appear in the lower window. After all, the code runs on node two. But it was created on node one, and so it inherits its process hierarchy from node one. Part of that hierarchy is something called the *group leader*, which (among other things) determines where IO.puts sends its output. So, in a way, what we're seeing is doubly impressive. We start on node one, run a process on node two, and when the process outputs something, it appears back on node one.

Your turn…

➤ *Exercise: Nodes-1*

Set up two terminal windows, and go to a different directory in each. Then start up a named node in each. Then, in one window, write a function that lists the contents of the current directory.

```
fun = fn -> IO.puts(Enum.join(File.ls!, ",")) end
```

Run it twice, once on each node.

Prepared exclusively for Richard Bishop

Nodes, Cookies, and Security

Although this is cool, it might also have rung some alarm bells. If you can run arbitrary code on any node, then anyone with a publicly accessible node has just handed over their machine to any random hacker.

But that's not the case. Before a node will let another connect, it checks that the remote node has permission. It does that by comparing that node's *cookie* with its own cookie. A cookie is just an arbitrary string (ideally fairly long and very random). As an administrator of a distributed Elixir system, you need to create a cookie, and then make sure all nodes use it.

If you are running the iex or elixir commands, you can pass in the cookie using the --cookie option.

```
$ iex --sname one --cookie chocolate-chip
iex(one@light-boy)> Node.get_cookie
:"chocolate-chip"
```

So if we repeat our two node experiment and explicitly set the cookie names to be different, what happens?

Window #1

```
$ iex --sname node_one --cookie cookie-one
iex(node_one@light-boy)> Node.connect :"node_two@light-boy"
false
```

Window #2

```
$ iex --sname node_two --cookie cookie-two
iex(node_two@light-boy)>
=ERROR REPORT==== 27-Apr-2013::21:27:43 ===
** Connection attempt from disallowed node 'node_one@light-boy' **
```

The node that attempts to connect receives false, indicating the connection was not made. And the node that it tried to connect to logs an error describing the attempt.

But why does it succeed when we don't specify a cookie? When Erlang starts, it looks for a file in your home directory called .erlang.cookie. If it doesn't exist, Erlang creates it and stores a random string in it. It uses that string as the cookie for any node started by that user. That way, all nodes you start on a particular machine are automatically given access to each other.

Be careful when connecting nodes over a public network—the cookie is transmitted in plain text.

Prepared exclusively for Richard Bishop

Naming Your Processes

Although a pid is displayed as three numbers, it just contains two fields, the first number is the node id, and the next two numbers are the low and high bits of the process id. When you run a process on your current node, its node id will always be zero. However, when you export a pid to another node, the node id is set to the number of the node on which the process lives.

That works well once a system is up and running and everything is knitted together. If you want to register a callback process on one node with an event generating process on another, just give the callback pid to the generator.

But how can the callback find the generator in the first place? One way is for the generator to register its pid, giving it a name. The callback on the other node can look up the generator by name, using the pid that comes back to send messages to it.

Here's an example. Let's write a simple server that sends a notification about every 2 seconds. To receive the notification, a client has to register with the server. And we'll arrange things so that clients on different nodes can register.

While we're at it, we'll do a little packaging, so that to start the server you run Ticker.start, and to start the client, Client.start. We'll also add an API Ticker.register to register a client with the server.

Here's the server code:

nodes/ticker.ex

```
defmodule Ticker do

  @interval 2000   # 2 seconds

  @name  :ticker

  def start do
    pid = spawn(__MODULE__, :generator, [[]])
    :global.register_name(@name, pid)
  end

  def register(client_pid) do
    send :global.whereis_name(@name), { :register, client_pid }
  end

  def generator(clients) do
    receive do
      { :register, pid } ->
        IO.puts "registering #{inspect pid}"
        generator([pid|clients])
```

Prepared exclusively for Richard Bishop

```
    after
      @interval ->
        IO.puts "tick"
        Enum.each clients, fn client ->
          send client, { :tick }
        end
        generator(clients)
    end
  end
end
```

We define a start function that spawns the server process. It then uses :global.register_name to register the pid of this server under the name :ticker.

The register function is called by clients who want to register to receive ticks. This function sends a message to the Ticker server, asking it to add the client to its list. Clients could have done this directly by sending the :register message to the server process. Instead, we give them an interface function that hides the details of registration. This helps decouple the client from the server, and gives us more flexibility to change things in the future.

Before we look at the actual tick process, let's stop to consider the start and register functions. These are not part of the ticking process—they are simply chunks of code in the Ticker module. This means they can be called directly wherever we have the module loaded—there's no message passing required. This is a common pattern—we have a module that is responsible both for spawning a process and for providing the external interface to that process.

Back to the code. The last function, generator, is the spawned process. It waits for two events. When it gets a tuple containing :register and a PID, it adds the pid to the list of clients and recurses. Alternatively, it may timeout after two seconds, in which case it sends a {:tick} message to all registered clients.

(This code has no error handling, and no means of terminating the process. I just wanted to illustrate passing pids and messages between nodes.)

The client code is simple.

nodes/ticker.ex

```
defmodule Client do

  def start do
    pid = spawn(__MODULE__, :receiver, [])
    Ticker.register(pid)
  end

  def receiver do
```

Prepared exclusively for Richard Bishop

```
    receive do
      { :tick } ->
        IO.puts "tock in client"
        receiver
    end
  end
end
```

It spawns a receiver to handle the incoming ticks, and passes the pid of the receiver to the server as an argument to the register function. Again, it's worth noting that this function call is local—it runs on the same node as the client. However, inside the Tick.register function, it locates the node containing the server, and sends a message to it. As the pid of our client is sent to the server, it becomes an external pid, pointing back to the client's node.

The spawned client process simply loops, receiving tick messages and writing a cheery message to the console when it gets one.

Let's run it. We'll start up our two nodes. We'll call Ticker.start on node one. Then we'll call Client.start on both node one and node two.

Window #1

```
nodes % iex --sname one
iex(one@light-boy)> c("ticker.ex")
[Client,Ticker]
iex(one@light-boy)> Node.connect :"two@light-boy"
true
iex(one@light-boy)> Ticker.start
:yes
tick
tick
iex(one@light-boy)> Client.start
registering #PID<0.59.0>
{:register,#PID<0.59.0>}
tick
tock in client
tick
tock in client
tick
tock in client
tick
tock in client
 :    :   :
```

Window #2

```
nodes % iex --sname two
iex(two@light-boy)> c("ticker.ex")
```

Prepared exclusively for Richard Bishop

```
[Client,Ticker]
iex(two@light-boy)> Client.start
{:register,#PID<0.53.0>}
tock in client
tock in client
tock in client
 :    :   :
```

To stop this, you'll need to exit out of iex on both nodes.

When To Name Processes

When you name something, you are recording some global state. And, as we all know, global state can be troublesome. What if two processes try to register the same name, for example.

The runtime does have some tricks to help us here. In particular, we can list the names our application will register in the app's mix.exs file. (We'll see how when we look at packaging an application on page 224. However, the general rule is: register your process names when your application starts.

Your turn…

➤ *Exercise: Nodes-2*

When I introduced the interval server, I said it sent a tick "about every 2 seconds". But in the receive loop, it has an explicit timeout of 2000mS. Why did I say "about" when it looks as if the time should be pretty accurate?

➤ *Exercise: Nodes-3*

Alter the code so that successive ticks are sent to each registered client (so the first goes to the first client, the second the next client, and so on). Once the last client receives a tick, it starts back at the first. The solution should deal with new clients being added at any time.

I/O, PIDs, and Nodes

Input and Output in the Erlang VM is performed using I/O servers. These are simply Erlang processes that implement a low-level message interface. You never have to deal with this interface directly (which is a good thing, as it is complex). Instead, you use the various Elixir and Erlang I/O libraries, and let them do the heavy lifting.

Prepared exclusively for Richard Bishop

In Elixir, you identify an open file or device by the pid of its I/O server. And these pids behave just like all other pids—you can, for example, send them between nodes.

If you look at the implementation of Elixir's IO.puts function,[1] you'll see:

```
def puts(device \\ group_leader(), item) do
    erl_dev = map_dev(device)
    :io.put_chars erl_dev, [to_iodata(item), ?\n]
end
```

The default device it uses is returned by the function :erlang.group_leader. (The group_leader function is imported from the :erlang module at the top of the IO module.) This will be the PID of an I/O server.

So, bring up two terminal windows and start a different named node in each. Connect to node one from node two, and register the pid returned by group_leader under a global name (we use :two).

Window #1

```
$ iex --sname one
iex(one@light-boy) >
```

Window #2

```
$ iex --sname two
iex(two@light-boy) > Node.connect(:"one@light-boy")
true
iex(two@light-boy) > :global.register_name(:two, :erlang.group_leader)
:yes
```

Note that we've registered the pid, we can access it from the other node. And once we've done that, we can pass it to IO.puts, the output appears in the other terminal window.

Window #1

```
iex(one@light-boy) > two = :global.whereis_name :two
#PID<7419.30.0>
iex(one@light-boy) > IO.puts(two, "Hello")
:ok
iex(one@light-boy) > IO.puts(two, "World!")
:ok
```

1. To see the source of an Elixir library module, view the online documentation at http://elixir-lang.org/docs/, navigate to the function in question and click the *Source* link.

Prepared exclusively for Richard Bishop

Window #2

```
Hello
World
iex(two@light-boy) >
```

Your turn…

➤ *Exercise: Nodes-4*

The ticker process in this chapter is a central server that sends events to registered clients. Reimplement this as a ring of clients. A client sends a tick to the next client in the ring. After 2 seconds, that next client sends a tick to its next client.

When thinking about how to add clients to the ring, remember to deal with the case where a client's receive loop times out just as you're adding a new process. What does this say about who has to be responsible for updating the links?

What's Next

It's easy to write concurrent applications with Elixir. But writing code that follows the happy path is a lot easier than writing bullet-proof, scalable, and hot-swappable world-beating apps. For that, you're going to need some help.

In the worlds of Elixir and Erlang, that help is called OTP, and is the subject of the next few chapters.

Prepared exclusively for Richard Bishop

CHAPTER 16

OTP: Servers

If you've been following Elixir or Erlang, you'll probably have come across OTP. It is often hyped as the answer to all high-availability distributed application woes. It isn't, but it certainly solves many problems that you'd otherwise need to solve yourself, including application discovery, detection and management of failure, hot code swapping, and the structure of servers.

First, the obligatory one-paragraph history. OTP stands for the *Open Telecom Platform*, but the full name is largely of historical interest, and everyone just says OTP. It was initially used to build telephone exchanges and switches. But these devices have the same characteristics we want from any large online application, so OTP is now a general purpose tool for developing and managing large systems.

OTP is actually a bundle that includes Erlang, a database (wonderfully called *Mnesia*), and an innumerable number of libraries. It also defines a structure for your applications. But, as with all large, complex, frameworks, there is a lot to learn. In this book, we'll focus on the essentials, and point you towards other sources of information.

We've been using OTP all along—mix, the elixir compiler, and even our Issue tracker followed OTP conventions. But that use was implicit. Now we're going to make it explicit, and start writing servers using OTP.

Some OTP Definitions

OTP defines systems in terms of hierarchies of *applications*. An application consists of one or more processes. These processes follow one of a small number of OTP conventions, called *behaviors* (or *behaviours*). There is a behavior used for general purpose servers, one for implementing event handlers, and one for finite-state machines. Each implementation of one of these

Prepared exclusively for Richard Bishop

behaviors will run in its own process (and may have additional associated processes). In this chapter we'll be implementing the *server* behavior, called *GenServer*.

A special behavior, called *supervisor*, monitors the health of these processes, and implements strategies for restarting them if needed.

We'll look at these components from the bottom up—this chapter and the next will look at two different servers, the next at supervisors, and finally we implement applications.

An OTP Server

When we wrote our Fibonacci server in the previous chapter on page 180, we had to do all the message handling ourselves. It wasn't difficult, but it was tedious. Our scheduler also had to keep track of three pieces of state information—the queue of numbers to process, the results generated so far, and the list of active pids.

It turns out that most servers have a similar set of needs, so OTP provides libraries that do all the low-level work for you.

When you write an OTP server, you write a module containing one or more callback functions with standard names. OTP will call the appropriate callback to handle a particular situation. For example, when someone sends a request to your server, OTP will call your `handle_call` function, passing in the request, the caller, and the current server state. Your function responds by returning a tuple containing an action to take, the return value for the request, and an updated state.

State and the Single Server

Way back when we summed the elements in a list on page 68, we came across the idea of an accumulator, a value that was passed as a parameter when a looping function calls itself recursively.

lists/sum.exs

```
defmodule MyList do
  def sum([], total),              do: total
  def sum([ head | tail ], total), do: sum(tail, head+total)
end
```

The parameter `total` maintains the state while the function trundles down the list.

In our Fibonacci code, we maintained a lot of state in the `schedule_processes` function. In fact, all three of its parameters were used for state information.

Prepared exclusively for Richard Bishop

Now think about servers. They use recursion to loop, handling one request on each call. So they can also pass state to themselves as a parameter in this recursive call. And that's one of the things OTP manages for us. Our handler functions get passed the current state (as their last parameter) and they return (among other things) a potentially updated state. Whatever state a function returns is the state that will be passed to the next request handler.

Our First OTP Server

Let's write what is possibly the simplest OTP server. You pass it a number when you start it up, and that becomes the current state of the server. When you call it with a :next_number request, it returns that current state to the caller, and at the same time increments the state, ready for the next call. Basically, each time you call it, you get an updated sequence number.

Create a new project using mix

Start by creating a new mix project in your work directory. We'll call it sequence.

```
$ mix new sequence
* creating README.md
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/sequence.ex
* creating lib/sequence
* creating lib/sequence/supervisor.ex
* creating test
* creating test/test_helper.exs
* creating test/sequence_test.exs
```

Create the basic sequence server

Now we're going to create Sequence.Server, our server module. Add the file server.ex to the sequence/ directory under lib.

otp-server/1/sequence/lib/sequence/server.ex

```
Line 1 defmodule Sequence.Server do
   2   use GenServer
   3
   4   def handle_call(:next_number, _from, current_number) do
   5     { :reply, current_number, current_number+1 }
   6   end
   7 end
```

The first thing to note is line 2. The use line effectively adds the OTP *GenServer* behavior to our module. This is what lets it handle all the callbacks. It also means we don't have to define every callback in our module—the behavior defines defaults for them all.

Prepared exclusively for Richard Bishop

The handle_call function that follows is invoked by GenServer when a client calls our server. It receives

- the information passed to the call by the client as its first parameter,
- the pid of the client as the second parameter,
- and the server state as the third

Our implementation is simple: we return a tuple to OTP:

```
{ :reply, current_number, current_number+1 }
```

The reply element tells OTP to reply to the client, passing back the value which is the second element. Finally, the third element of the tuple defines the new state. This will be passed as the last parameter to handle_call the next time it is invoked.

Fire up our server manually

We can play with our server in iex. Open it in the project's main directory, remembering the -S mix option.

```
$ iex -S mix
iex> { :ok, pid } = GenServer.start_link(Sequence.Server, 100)
{:ok,#PID<0.71.0>}
iex> GenServer.call(pid, :next_number)
100
iex> GenServer.call(pid, :next_number)
101
iex> GenServer.call(pid, :next_number)
102
```

We're using two functions from the Elixir GenServer module. The start_link function behaves like the spawn_link function we used in the previous chapter. It asks GenServer to start a new process and link to us (so we'll get notifications if it fails). We pass in the module to run as a server, the initial state (100, in this case). We could also pass GenServer options as a third parameter, but the defaults work fine here.

We get back a status (:ok) and the pid of the server. The call function takes this pid, and calls the handle_call function in the server. The second parameter of the call is passed as the first argument to handle_call.

In our case, the only value we need to pass is the identity of the action we want to perform, :next_number. If you look at the definition of handle_call in the server, you'll see that its first parameter is :next_number. When Elixir invokes the function, it pattern matches the argument in the call with this first parameter in the function. A server can support multiple actions by implementing multiple handle_call functions with different first parameters.

Prepared exclusively for Richard Bishop

If you want to pass more than one thing in the call to a server, pass a tuple. For example, our server might need a function to reset the count to a given value. We could define the handler as

```
def handle_call({:set_number, new_number}, _from, _current_number) do
  { :reply, new_number, new_number }
end
```

and call it with

```
iex> GenServer.call(pid, {:set_number, 999})
999
```

Similarly, a handler can return multiple values by packaging them into a tuple or list.

```
def handle_call({:factors, number}, _, _) do
  { :reply, { :factors_of, number, factors(number)}, [] }
end
```

Your turn…

➤ *Exercise: OTP-Servers-1*

You're going to start creating a server that implements a stack. The call that initializes your stack will pass in a list that is the initial stack contents.

For now, only implement the `pop` interface. It's acceptable for your server to crash if someone tries to pop from an empty stack.

For example, if initialized with `[5,"cat",9]`, successive calls to `pop` will return `5`, `"cat"`, and `9`.

One-Way Calls

The `call` function calls a server and waits for a reply. But sometimes you won't want to wait because there is no reply coming back. In those circumstances, use the GenServer `cast` function. (Think of it as casting your request into the sea of servers.)

Just like `call` is passed to `handle_call` in the server, a `cast` is sent to `handle_cast`. Because there's no response possible, the `handle_cast` function only takes two parameters, the call argument and the current state. And, because it doesn't want to send a reply, it will return the tuple `{:noreply, new_state}`.

Let's modify our sequence server to support an `:increment_number` function. We'll treat this as a cast, so it simply sets the new state and returns.

Prepared exclusively for Richard Bishop

otp-server/1/sequence/lib/sequence/server.ex

```
Line 1 defmodule Sequence.Server do
  -   use GenServer
  -
  -   def handle_call(:next_number, _from, current_number) do
    5     { :reply, current_number, current_number+1 }
  -   end
  -
  -   def handle_cast({:increment_number, delta}, current_number) do
  -     { :noreply, current_number + delta}
   10   end
  - end
```

Notice that the cast handler takes a tuple as its first parameter. The first element is :increment_number, and is used by pattern matching to select the handlers to run. The second element of the tuple is the delta to add to our state. The function simply returns a tuple where the state is the previous state plus this number.

To call this from our iex session, we first have to recompile our source. The r command takes a module name and recompiles the file containing that module.

```
iex> r Sequence.Server
.../sequence/lib/sequence/server.ex:2: redefining module Sequence.Server
{Sequence.Server,[Sequence.Server]}
```

Even though we've recompiled the code, the old version is still running. The VM doesn't hotswap code until you explicitly access it by module name. So, to try our new functionality, we'll simply create a new server. When it starts, it will pick up the latest version of the code.

```
iex> { :ok, pid } = GenServer.start_link(Sequence.Server, 100)
{:ok,#PID<0.60.0>}
iex> GenServer.call(pid, :next_number)
100
iex> GenServer.call(pid, :next_number)
101
iex> GenServer.cast(pid, {:increment_number, 200})
:ok
iex> GenServer.call(pid, :next_number)
302
```

Tracing the Execution of a Server

The third parameter to start_link is a set of options. A useful one during development is the debug trace, which logs message activity to the console.

You enable tracing using

Prepared exclusively for Richard Bishop

```
iex> {:ok,pid} = GenServer.start_link(Sequence.Server, 100, [debug: [:trace]])
{:ok,#PID<0.68.0>}
iex> GenServer.call(pid, :next_number)
*DBG* <0.68.0> got call next_number from <0.25.0>
*DBG* <0.68.0> sent 100 to <0.25.0>, new state 101
100
iex> GenServer.call(pid, :next_number)
*DBG* <0.68.0> got call next_number from <0.25.0>
*DBG* <0.68.0> sent 101 to <0.25.0>, new state 102
101
```

See how it traces the incoming call and the response we send back. A nice touch is that it also shows the next state.

You can also ask a server to keep some basic statistics by including :statistics in the debug list.

```
iex> {:ok,pid} = GenServer.start_link(Sequence.Server, 100,
...>                                  [debug: [:statistics]])
{:ok,#PID<0.69.0>}
iex> GenServer.call(pid, :next_number)
100
iex> GenServer.call(pid, :next_number)
101
iex> :sys.statistics pid, :get
{:ok,[start_time: {{2013,4,26},{18,17,16}}, current_time: {{2013,4,26},{18,17,28}},
      reductions: 50, messages_in: 2, messages_out: 0]}
```

Most of the fields should be fairly obvious. Timestamps are given as {{y,m,d},{h,m,s}} tuples. And the reductions value is a measure of the amount of work done by the server. It is used in process scheduling as a way of making sure all processes get a fair share of the available CPU.

The Erlang sys module is your interface to the world of *system messages*. These are sent in the background between processes—they are a bit like the backchatter in a multiplayer video game. While two players are engaged in an attack (their real work), they can also be sending each other background messages: “Where are you?”, “Stop moving” and so on.

The list associated with the debug parameter you give to GenServer is simply the names of functions to call in the sys module, If you say [debug: [:trace, :statistics]], then those functions will be called in sys, passing in the server’s pid. Have a look at the documentation for sys[1] to see what’s available.

This also means that you can turn things on and off *after* you have started a server. For example, you can enable tracing on an existing server using

1. http://www.erlang.org/documentation/doc-5.8.3/lib/stdlib-1.17.3/doc/html/sys.html

Prepared exclusively for Richard Bishop

```
iex> :sys.trace pid, true
:ok
iex> GenServer.call(pid, :next_number)
*DBG* <0.69.0> got call next_number from <0.25.0>
*DBG* <0.69.0> sent 105 to <0.25.0>, new state 106
105
iex> :sys.trace pid, false
:ok
iex> GenServer.call(pid, :next_number)
106
```

Another useful sys function is get_status

```
iex> :sys.get_status pid
{:status,#PID<0.57.0>,{:module,:gen_server},[["$ancestors": [#PID<0.25.0>],
"$initial_call":
{Sequence.Server,:init,1}],:running,#PID<0.25.0>,[],
[header: 'Status for generic server <0.57.0>',
data: [{'Status',:running},{'Parent',#PID<0.25.0>},{'Logged events',[]}],
data: [{'State',102}]]]}
```

This is the default formatting of the status message provided by GenServer. You have the option to change the 'State' part to return a more application-specific message by defining format_status. This receives an option describing why the function was called, and a list containing the server's process dictionary and the current state. (Note that in the code that follows, the string *State* in the response is in single quotes.)

otp-server/1/sequence/lib/sequence/server.ex

```
def format_status(_reason, [ _pdict, state ]) do
  [data: [{'State', "My current state is '#{inspect state}', and I'm happy"}]]
end
```

If we ask for the status in iex, we now get the new message (after restarting the server).

```
iex> :sys.get_status pid
{:status,#PID<0.61.0>,{:module,:gen_server},[["$ancestors": [#PID<0.25.0>],
"$initial_call": {Sequence.Server,:init,1}],:running,#PID<0.25.0>,
[trace: true],[header: 'Status for generic server <0.61.0>',
{'Parent',#PID<0.25.0>},{'Logged events',[]}],
data: [{'State',"My current state is '103', and I'm happy"}]]]}
```

Your turn…

➤ *Exercise: OTP-Servers-2*

Extend your stack server with a push interface which adds a single value to the top of the stack. This will be implemented as a cast.

Experiment in iex with pushing and popping values.

Prepared exclusively for Richard Bishop

GenServer Callbacks

GenServer is an OTP protocol. OTP works by assuming that your module defines a number of callback functions (six, in the case of a GenServer). If you were writing a GenServer in Erlang, your code would have to contain implementations of all six.

When you add the line `use GenServer` to a module, Elixir creates default implementations of these 6 callback functions. All we have to do is override the ones where we add our own application-specific behaviour. Our examples so far have used the two callbacks `handle_call`, and `handle_cast`. Here's a full list.

init(start_arguments)

called by GenServer when starting a new server. The parameter is the second argument passed to `start_link`. Should return `{:ok, state}` on success, or `{:stop, reason}` if the server could not be started.

You can specify an optional timeout using `{:ok, state, timeout}`, in which case GenServer will send the process a `:timeout` message whenever no message is received in a span of *timeout* ms. (The message is passed to the `handle_info` function.

The default GenServer implementation sets the server state to the argument you pass.

handle_call(request, from, state)

Invoked when a client uses `GenServer.call(pid, request)`. The `from` parameter is a tuple containing the pid of the client and a unique tag. The `state` parameter is the server state.

On success returns `{:reply, result, new_state}`. Other valid responses are shown in the list that follows this one on page 206.

The default implementation stops the server with a `:bad_call` error, so you'll need to implement `handle_call` for every call time your server implements.

handle_cast(request, state)

Called in response to `GenServer.cast(pid, request)`.

A successful response is `{:noreply, new_state}`. Can also return `{:stop, reason, new_state}`.

The default implementation stops the server with a `:bad_cast` error.

handle_info(info, state)

Called to handle incoming messages that are not call or cast requests. For example, `timeout` messages are handled here. So are termination mes-

Prepared exclusively for Richard Bishop

sages from any linked processes. In addition, messages sent to the pid using send (so they bypass GenServer) will be routed to this function.

terminate(reason, state)

Called when the server is about to be terminated. However, as we'll see in the next chapter, once we add supervision to our servers, we don't have to worry about this.

code_change(from_version, state, extra)

OTP lets us replace a running server without stopping the system. However, the new version of the server may represent its state differently to the old. The code_change callback is invoked to change from the old state format to the new.

format_status(reason, [pdict, state])

Used to customize the state display of the server. The conventional response is [data: [{'State', state_info}]].

The call and cast handlers return standardized responses. Some of these responses can contain an optional :hibernate or timeout parameter. If hibernate is returned, the server state is removed from memory, but it is recovered on the next request. This saves memory at the expense of some CPU. The timeout option can be the atom :infinite (which is the default) or a number. If the latter, a :timeout message is sent if the server is idle for that many milliseconds.

The first two responses are common between call and cast.

```
{ :noreply, new_state [ , :hibernate | timeout ] }
```

{ :stop, reason, new_state }

Signal that the server is to terminate.

The last two can only be used by handle_call.

{ :reply, response, new_state [, :hibernate | timeout] }

Send response to the client.

{ :stop, reason, reply, new_state }

Send the response and signal that the server is to terminate.

Naming A Process

The idea of referencing processes by their pid gets old quickly. Fortunately, there are a number of alternatives.

The simplest is local naming. You assign a name that is unique for all OTP processes on your server, and then use that name instead of the pid whenever

Prepared exclusively for Richard Bishop

you reference it. To create a locally named process, use the name: option when you start the server:

```
iex> { :ok, pid } = GenServer.start_link(Sequence.Server, 100, name: :seq)
{:ok,#PID<0.66.0>}
iex> GenServer.call(:seq, :next_number)
100
iex> GenServer.call(:seq, :next_number)
101
iex> :sys.get_status :seq
{:status, #PID<0.69.0>, {:module, :gen_server},
 [["$ancestors": [#PID<0.58.0>], "$initial_call": {Sequence.Server, :init, 1}],
  :running, #PID<0.58.0>, [],
  [header: 'Status for generic server seq',
   data: [{'Status', :running}, {'Parent', #PID<0.58.0>},
    {'Logged events', []}],
   data: [{'State', "My current state is '102', and I'm happy"}]]]}
```

Tidying Up The Interface

As we left it, our server works, but it is ugly to use. Our callers have to make explicit GenServer calls, and they have to know the registered name for our server process. We can do better. Let's wrap this interface in a set of three functions in our server module: start_link, next_number, and increment_number. The first of these calls the GenServer start_link method. As we'll see in a couple of chapters when we look at supervisors, the name start_link is a convention. start_link must return the correct status values to OTP, but as our code simply delegates to the GenServer module, this is taken care of.

Following the definition of start_link, the next two functions are the external API to issue call and cast requests to the running server process.

We'll also use the name of the module as the registered local name of our server (hence the name: __MODULE__ when we start it, and the __MODULE__ parameter when we use call or cast).

otp-server/2/sequence/lib/sequence/server.ex

```
defmodule Sequence.Server do
  use GenServer

  #####
  # External API

➤ def start_link(current_number) do
    GenServer.start_link(__MODULE__, current_number, name: __MODULE__)
  end

➤ def next_number do
```

Prepared exclusively for Richard Bishop

```
    GenServer.call __MODULE__, :next_number
  end

  def increment_number(delta) do
    GenServer.cast __MODULE__, {:increment_number, delta}
  end

  #####
  # GenServer implementation

  def handle_call(:next_number, _from, current_number) do
    { :reply, current_number, current_number+1 }
  end

  def handle_cast({:increment_number, delta}, current_number) do
    { :noreply, current_number + delta}
  end

  def format_status(_reason, [ _pdict, state ]) do
    [data: [{'State', "My current state is '#{inspect state}', and I'm happy"}]]
  end
end
```

When we run this code in iex, it's a lot cleaner.

```
$ iex -S mix
iex> Sequence.Server.start_link 123
{:ok,#PID<0.57.0>}
iex> Sequence.Server.next_number
123
iex> Sequence.Server.next_number
124
iex> Sequence.Server.increment_number 100
:ok
iex> Sequence.Server.next_number
225
```

This is the pattern you should use in your servers.

Your turn…

- *Exercise: OTP-Servers-3*
  Give your stack server process a name, and make sure it is accessible by that name in iex.

- *Exercise: OTP-Servers-4*
  Add the API to your stack module (the functions that wrap the gen_server calls).

Prepared exclusively for Richard Bishop

➤ *Exercise: OTP-Servers-5*

Implement the terminate callback in your stack handler. Use IO.puts to report the arguments it receives.

Try various ways of terminating your server. For example, popping an empty stack will raise an exception. You might add code that calls System.halt(n) if the push handler receives a number less than 10. (This will let you generate different return codes). Use your imagination to try different scenarios.

What We Learned

An OTP GenServer is just a regular Elixir process in which the message handling has been abstracted out. The GenServer behavior defines a message loop internally, and maintains a state variable. That message loop then calls out to various functions that you define in your server module, handle_call, handle_cast, and so on.

We also saw that GenServer provides fairly detailed tracing of the messages received and responses sent by your server modules.

Finally, we wrapped our message-based API in module functions, which gives our users a cleaner interface, and decouples them from our implementation.

But we still have an issue if our server crashes. We'll deal with this in the next chapter, when we look at supervisors.

Prepared exclusively for Richard Bishop

CHAPTER 17

OTP: Supervisors

We've said it a few times now: the *Elixir way* says not to worry too much about code that crashes; instead make sure that the overall application keeps running.

This might sound contradictory, but really it is not.

Think of a typical application. If an unhandled error causes an exception to be raised, the application stops. Nothing else gets done until it is restarted. If it's a server handling multiple requests, they all might be lost.

The issue here is that one error takes the whole application down.

But imagine instead that your application consists of hundreds or thousands of processes, each handling just a small part of a request. If one of those crashes, everything else carries on. You might lose the work that it is doing, but you can also design your applications to minimize even that risk. And when that process gets restarted, you're back running at 100%.

In the Elixir and OTP worlds, all of this process monitoring and restarting is performed by supervisors.

Supervisors And Workers

An Elixir supervisor is a process with just one purpose—it manages one or more worker processes. (As we'll see later, it can potentially also manage other supervisors.)

At its simplest, a supervisor is simply a process that uses the OTP supervisor behavior. It is given a list of processes to monitor. It is also told what to do if a process dies, and how to prevent restart loops (when a process is restarted, dies, gets restarted, dies, and so on).

Prepared exclusively for Richard Bishop

To do this, the supervisor uses the process linking and monitoring facilities of the Erlang VM. I talked about these when I covered spawn on page 175.

You write supervisors as separate modules, normally in the same application containing the processes you want to supervise. Let's extend the sequence serving application. We'll use a supervisor to make sure that the sequence generating process (in Sequence.Server) keeps running.

By default, mix new creates a supervisor for every new project. It lives in the file supervisor.ex in the lib/sequence directory. We'll modify this to create our basic supervisor.

otp-supervisor/1/sequence/lib/sequence/supervisor.ex

```
defmodule Sequence.Supervisor do
  use Supervisor

  def start_link(initial_number) do
    Supervisor.start_link(__MODULE__, initial_number)
  end

  def init(initial_number) do
    children = [ worker(Sequence.Server, [initial_number]) ]
    supervise children, strategy: :one_for_one
  end
end
```

It is similar in structure to a GenServer module, but it uses the Supervisor behavior. It contains two functions. start_link is the external API—we call this to start the supervisor (which in turn will start the sequence process). We pass in the initial state we want to pass to the sequence process's start_link function (in this case a number).

The supervisor's start_link method delegates its work to Elixir's Supervisor library. It passes in its own module name and the parameter to be passed to the supervised process. But the sequence process is not started yet. Instead, the supervisor is started in a separate process, and its init function is called. This init function is responsible for starting the supervised process.

Let's look at that init function.

otp-supervisor/1/sequence/lib/sequence/supervisor.ex

```
def init(initial_number) do
  children = [ worker(Sequence.Server, [initial_number]) ]
  supervise children, strategy: :one_for_one
end
```

Prepared exclusively for Richard Bishop

It uses two functions defined by the Elixir Supervisor behavior.[1] The worker function creates a specification of a child worker. It takes the name of the GenServer module containing the worker (Sequence.Server in our case) and the parameter to pass in. It returns a tuple in a form that can be used by the OTP libraries.

The supervise function takes a list of these worker specifications and starts them. The strategy parameter tells the supervisor how to restart children. In this case, :one_for_one means to restart a child if it dies.

So, let's look at the sequence of what will happen.

- We call Sequence.Supervisor.start_link(100).
- this calls the function Supervisor.start_link, passing it the current module's name.
  - OTP then starts our module as a supervisor process, and invokes its init function (which we write).
- Our init function creates a list of child processes using the worker function.
- It then calls the supervise function, passing in this list.
  - The supervise function in turn calls the start_link function in our Sequence.Server module. Now we're back on familiar ground, as we spawn the actual worker.

However, there's still one more layer to consider. You may have noticed that mix new creates a file called lib/*appname*.ex. This is the top-level of your application, and it is automatically invoked when you use mix to run your code. Up until now, we've been able to ignore it. But no longer. Let's see what it contains:

```
defmodule Sequence do
  def start(_type, _args) do
    Sequence.Supervisor.start_link
  end
end
```

Our top-level module defines a start function, and that function invokes our supervisor. And, as we now know, that supervisor in turn starts the actual application code.

Enough theory. Let's try it:

```
$ iex -S mix
Compiled lib/sequence.ex
```

1. http://elixir-lang.org/docs/master/Supervisor.Behaviour.html

Prepared exclusively for Richard Bishop

```
Compiled lib/sequence/supervisor.ex
Compiled lib/sequence/server.ex
Generated sequence.app
** (Mix) Could not start application sequence: {:bad_return,
{{Sequence, :start, [:normal, []]}, {:EXIT, {:undef,
[{Sequence.Supervisor, :start_link, [], []}, {:application_master,
:start_it_old, 4, [file: 'application_master.erl', line: 274]}]}}}}
```

What does this error mean? The call to Sequence.start returned a bad value. And that bad value was a process exit message. The reason for the exit was :undef, [{Sequence.Supervisor, :start_link, [], []}. We called start_link with no parameters, but the function in our supervisor takes one argument. Let's fix the call.

otp-supervisor/1/sequence/lib/sequence.ex

```
defmodule Sequence do
  use Application

  def start(_type, _args) do
    Sequence.Supervisor.start_link(123)
  end
end
```

```
$ iex -S mix
Compiled lib/sequence.ex
Generated sequence.app

iex> Sequence.Server.increment_number 3
:ok
iex> Sequence.Server.next_number
126
```

So far, so good. But the key thing with a supervisor is that it is supposed to manage our worker process. If it dies, for example, we want it to be restarted. Let's try that. If we pass increment_number something that isn't a number, the process should die trying to add it to the current number.

```
iex> Sequence.Server.increment_number "cat"
:ok
=ERROR REPORT==== 5-May-2013::19:13:18 ===
** Generic server sequence terminating
** Last message in was {'$gen_cast',{increment_number,<<"cat">>}}
** When Server state == 226
** Reason for termination ==
** {badarith,
       [{'Elixir.Sequence-Server',handle_cast,2,
            [{file,
                 "...lib/sequence/server.ex"},
             {line,32}]},
        {gen_server,handle_msg,5,[{file,"gen_server.erl"},{line,607}]},
        {proc_lib,init_p_do_apply,3,[{file,"proc_lib.erl"},{line,227}]}]}
```

Prepared exclusively for Richard Bishop

```
nil
iex> Sequence.Server.next_number
123
iex> Sequence.Server.next_number
124
```

We get a wonderful error report which shows us the badarith exception, along with a stack trace from the process. We can also see the message that we sent that triggered the problem.

But, when we then asked our server for a number, it responded as if nothing had happened. The supervisor had restarted it for us.

This is excellent, but there's a problem. When the supervisor restarted our sequence process, it restarted it with the initial parameters we passed in, the numbers started again from 123. A reincarnated process has no memory of its past lives, and no state is retained across a crash.

Your turn…

➤ *Exercise: OTP-Supervisors-1*

Add a supervisor to your stack application. Use iex to make sure it starts the server correctly. Use the server normally, and then crash it (try popping from an empty stack). Did it restart? What was the stack contents after the restart?

Managing Process State Across Restarts

Some servers are effectively stateless. If we had a server that calculated the factors of numbers, or that responded to network requests with the current time, we could simply restart them and let them run.

But our server is not stateless—it needs to remember the current number so that it can generate an increasing sequence.

There are a number of approaches to this, and they all involve storing the state outside of the process. Let's choose a simple option—we'll write a separate worker process that can store and retrieve a value. We'll call it our *stash*. The sequence server can store its current number in the stash whenever it terminates, and then we can recover the number when we restart.

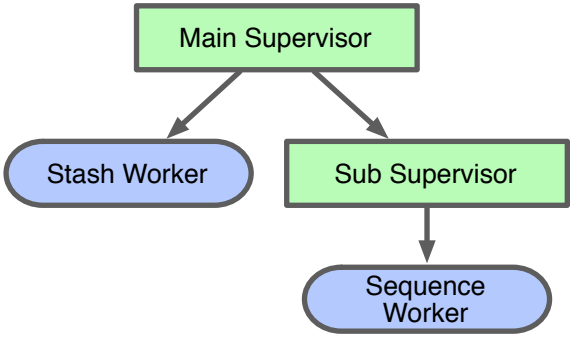
At this point, we have to think about lifetimes. Our sequence server should be fairly robust, but we've already found one thing that crashes it. So, in actuarial terms, it isn't the fittest process in the scheduler queue. But our

Prepared exclusively for Richard Bishop

stash process must be more robust—it must outlive the sequence server, at the very least.

We have to do two things to make this happen. First, we make it as simple as possible. The fewer moving parts in a chunk of code, the less likely it is to go wrong.

The second thing we have to do is to supervise it separately. In fact, we're going to create a supervision tree. It'll look like the following diagram.



Here we have a top-level supervisor that is responsible for the health of two things, the stash worker and a second supervisor. That second supervisor then manages the worker that generates the sequence.

Our sequence generator will need to know the PID of the stash in order to retrieve and store the sequence value. We *could* register the stash process under a name (just as we did with the sequence worker itself), but as this is purely a local affair, we can just pass it the PID directly. But, to do that, we need to get the stash worker spawned first. This leads to a slightly different design for the top-level supervisor. Rather than give it a list of child processes to start, we're going to initialize it with no children, and then add the stash and the subsupervisor manually. Once we start the stash worker, we'll have its pid, and we can then pass it on to the subsupervisor (which in turn will pass it to the sequence worker). Our overall supervisor looks like this:

otp-supervisor/2/sequence/lib/sequence/supervisor.ex

```
Line 1 defmodule Sequence.Supervisor do
   -   use Supervisor
   -
   -   def start_link(initial_number) do
   5     result = {:ok, sup } = Supervisor.start_link(__MODULE__, [initial_number])
   -     start_workers(sup, initial_number)
   -     result
   -   end
   -
```

```
10  def start_workers(sup, initial_number) do
-     # Start the stash worker
-     {:ok, stash} =
-        Supervisor.start_child(sup, worker(Sequence.Stash, [initial_number]))
-
15    # and then the subsupervisor for the actual sequence server
-     Supervisor.start_child(sup, supervisor(Sequence.SubSupervisor, [stash]))
-   end
-
-   def init(_) do
20    supervise [], strategy: :one_for_one
-   end
-
- end
```

On line 5 we start up the supervisor. This will automatically invoke the init callback. This in turn calls supervise, but passes in an empty list. The supervisor is now running, but has no children.

At this point, OTP returns control back to our start_link function, which then calls the start_workers function. This starts the stash worker, passing it the initial number. We get back a status (:ok) and a pid. We then pass the pid to the subsupervisor.

This subsupervisor is basically the same as our very first supervisor—it simply spawns the sequence worker. But instead of passing in a current number, it passes in the pid of the stash.

otp-supervisor/2/sequence/lib/sequence/subsupervisor.ex

```
defmodule Sequence.SubSupervisor do
  use Supervisor

  def start_link(stash_pid) do
    Supervisor.start_link(__MODULE__, stash_pid)
  end

  def init(stash_pid) do
    child_processes = [ worker(Sequence.Server, [stash_pid]) ]
    supervise child_processes, strategy: :one_for_one
  end

end
```

The sequence worker has two changes. First, when it is initialized, it must get the current number from the stash. Second, when it terminates, it stores the then current number back in the stash. To do this, we'll override two more GenServer callbacks, init and terminate.

Prepared exclusively for Richard Bishop

otp-supervisor/2/sequence/lib/sequence/server.ex

```
defmodule Sequence.Server do
  use GenServer

  #####
  # External API

  def start_link(stash_pid) do
    GenServer.start_link(__MODULE__, stash_pid, name: __MODULE__)
  end

  def next_number do
    GenServer.call __MODULE__, :next_number
  end

  def increment_number(delta) do
    GenServer.cast __MODULE__, {:increment_number, delta}
  end

  #####
  # GenServer implementation

  def init(stash_pid) do
    current_number = Sequence.Stash.get_value stash_pid
    { :ok, {current_number, stash_pid} }
  end

  def handle_call(:next_number, _from, {current_number, stash_pid}) do
    { :reply, current_number, {current_number+1, stash_pid} }
  end

  def handle_cast({:increment_number, delta}, {current_number, stash_pid}) do
    { :noreply, {current_number + delta, stash_pid}}
  end

  def terminate(_reason, {current_number, stash_pid}) do
    Sequence.Stash.save_value stash_pid, current_number
  end
end
```

The stash itself is trivial:

otp-supervisor/2/sequence/lib/sequence/stash.ex

```
defmodule Sequence.Stash do
  use GenServer

  #####
  # External API

  def start_link(current_number) do
    GenServer.start_link( __MODULE__, current_number)
```

Prepared exclusively for Richard Bishop

```
  end

  def save_value(pid, value) do
    GenServer.cast pid, {:save_value, value}
  end

  def get_value(pid) do
    GenServer.call pid, :get_value
  end

  #####
  # GenServer implementation

  def handle_call(:get_value, _from, current_value) do
    { :reply, current_value, current_value }
  end

  def handle_cast({:save_value, value}, _current_value) do
    { :noreply, value}
  end
end
```

So let's work through what is going on here.

- We start the top-level supervisor, passing it an initial value for the counter. It starts up the stash worker, giving it this number. It then starts the subsupervisor, passing it the pid of the stash.
- The subsupervisor in turn starts the sequence worker. This goes to the stash, gets the current value, and uses that value and the stash pid as its state. The next_number and increment_number functions are unchanged (except they receive the more complex state.
- But if the sequence worker terminates for any reason, GenServer will call its terminate function. It stores its current value in the stash before dying.
- The subsupervisor will notice that a child has died. It will restart it, passing in the stash pid, and the newly incarnated worker will pick up the current value that was stored when the previous instance died.

At least that's the theory. Let's try it:

```
$ iex -S mix
Compiled lib/sequence.ex
Compiled lib/sequence/server.ex
Compiled lib/sequence/stash.ex
Compiled lib/sequence/subsupervisor.ex
Compiled lib/sequence/supervisor.ex
Generated sequence.app
iex> Sequence.Server.next_number
```

Prepared exclusively for Richard Bishop

```
123
iex> Sequence.Server.next_number
124
iex> Sequence.Server.increment_number 100
:ok
iex> Sequence.Server.next_number
225
iex> Sequence.Server.increment_number "cause it to crash"
:ok
iex>
=ERROR REPORT==== 8-May-2013::13:31:28 ===
** Generic server sequence terminating
** Last message in was {'$gen_cast',
                        {increment_number,<<"cause it to crash">>}}
** When Server state == {226,<0.70.0>}
** Reason for termination ==
** {badarith,
       [{'Elixir.Sequence-Server',handle_cast,2,
            [{file,
                 ".../lib/sequence/server.ex"},
             {line,32}]},
        {gen_server,handle_msg,5,[{file,"gen_server.erl"},{line,607}]},
        {proc_lib,init_p_do_apply,3,[{file,"proc_lib.erl"},{line,227}]}]}
iex> Sequence.Server.next_number
226
iex> Sequence.Server.next_number
227
```

Even though we crashed our sequence worker, it got restarted, and the state was preserved. Now we start to see how careful supervision is critical if we want to write reliable applications.

Supervisors Are The Heart of Reliability

If you think about our previous example, it was both trivial and profound. It was trivial because there are many ways of achieving some kind of fault tolerance with a library that returns successive numbers.

But it was profound because it is a concrete representation of the idea of building rings of confidence in our code. The outer ring, where our code interacts with the world, should be as reliable as we can make it. But within that ring there are other, nested rings. And in those rings, things can be less than perfect. The trick is to ensure that the code in each ring knows how to deal with failures of the code in the next ring down.

And that's where supervisors come in to play. In this chapter we've seen only a small fraction of the capabilities of supervisors. They have different strategies for dealing with the termination of a child, different ways of terminating

Prepared exclusively for Richard Bishop

children, and different ways of restarting them. There's plenty of information online about using OTP supervisors.

But the real power of supervisors is that they exist. The fact that you use them to manage your workers means that you are forced to think about reliability and state as you design your application. And that discipline leads to applications with very high availability—in *Programming Erlang: Software for a Concurrent World [Arm13]*, Joe Armstrong says OTP has been used to build systems with 99.9999999% reliability. That's nine nines. And that ain't bad.

There's one more level in our lightning tour of OTP—the application. And that's the next chapter.

Your turn...

➤ *Exercise: OTP-Supervisors-2*

Rework your stack server to use a supervision tree with a separate stash process to hold the state. Verify it works, and that when you crash the server the state is retained across a restart.

Prepared exclusively for Richard Bishop

CHAPTER 18

OTP: Applications

So far in our quick tour of Elixir and OTP we've looked at server processes and the supervisors that monitor them. There's one more stage in our journey—the application.

Application: I do not think it means what you think it means

Because OTP comes from the Erlang world, it uses Erlang names for things. And unfortunately some of these names are not terribly descriptive. The name *application* is one of these. When most of us talk about applications, we think of a program we run to do something—maybe on our computer, phone, or via a web browser. An application is a self-contained whole.

But in the OTP world, that's not the case. Instead, an application is a bundle of code that comes with a descriptor. That descriptor tells the runtime what dependencies the code has, what global names it registers, and so on. In fact, an OTP application is more like a DLL or shared object than a conventional application.

It might help to see the word *application* and in your head pronounce it *component* or *service*.

For example, back when we were fetching GitHub issues using the HTTPotion library, what we actually installed was an independent application containing HTTPotion. Although it looked like we were just using a library, mix automatically loaded the HTTPotion application. When we then started it, HTTPoption in turn started a couple of other applications that it needed (SSL and iBrowse), which in turn kicked off their own supervisors and workers. And all of this was transparent to us.

Prepared exclusively for Richard Bishop

Having said that applications are components, there are some applications that are at the top of the tree, and that are meant to be run directly.

In this chapter we'll look at both types of application component (see what I did there?). But in reality, they're virtually the same, so let's cover the common ground first.

The Application Specification File

You probably noticed that every now and then mix will talk about a file called *name*.app, where *name* is the name of your application.

This file is called an *application specification*, and is used to define your application to the runtime environment. Mix creates this file automatically from the information in mix.exs combined with information it gleans from compiling your application.

When you run your application this file is consulted to get things loaded.

Your application does not need to use all the OTP functionality—this file will always be created and referred to. However, as we'll see, once you start using OTP supervision trees, you'll be adding stuff to mix.exs which will get copied into the .app file.

Turning Our Sequence Program into an OTP Application

So, here's the good news. The application in the previous chapter is already a full-blown OTP application. When mix created the initial project tree, it added a supervisor (which we then modified) and enough information to our mix.exs file to get the application started. In particular, it filled in the application function:

```
def application do
  [mod: { Sequence, [] }]
end
```

This says that the top-level module of our application is called Sequence. OTP assumes this module will implement a start function, and it will pass that function an empty list as a parameter.

In our previous version of the start function, we ignored the arguments, and instead hard-wired the call to start_link to pass 123 to our application. Let's change that to take the value from mix.exs instead. First, change mix.exs to pass an initial value (we'll use 456):

```
def application do
  [mod: { Sequence, 456 }]
```

Prepared exclusively for Richard Bishop

```
end
```

Then, change the sequence.ex code to use this passed-in value:

otp-app/sequence/lib/sequence.ex

```
defmodule Sequence do

  use Application

  def start(_type, initial_number) do
    Sequence.Supervisor.start_link(initial_number)
  end

end
```

We can check that this works:

```
$ iex -S mix
Compiled lib/sequence.ex
Compiled lib/sequence/subsupervisor.ex
Compiled lib/sequence/stash.ex
Compiled lib/sequence/server.ex
Compiled lib/sequence/supervisor.ex
Generated sequence.app

iex> Sequence.Server.next_number
456
```

Let's look at the application function again.

The mod: option tells OTP the module that is the main entry point for our app. If our app is a conventional runnable application, then it will need to start somewhere, so we'd write our kickoff function here. But even pure library applications may need to be initialized. (For example, a logging library may start a background logger process, or connect to a central logging server).

For the sequence app, we tell OTP that the Sequence module is the main entry point. OTP will call the start function of this module when it starts the application. The second element of the tuple is the parameter to pass to this function. In our case, it's the initial number for the sequence.

There's a second option we'll want to add to this.

The registered: option lists the names that our application will register. This can be used to ensure each name is unique across all loaded applications in a node or cluster. In our case, the sequence server registers itself under the name :sequence, so we'll update the configuration to read:

otp-app/sequence/mix.exs

```
# Configuration for the OTP application
```

Prepared exclusively for Richard Bishop

```
def application do
  [
    mod: { Sequence, 456 },
    registered: [ :sequence ]
  ]
end
```

Now that we've done the configuring in mix, we run mix compile, which both compiles the app and updates the sequence.app application specification file with information from mix.exs. (The same thing happens if you run mix using iex -S mix.)

```
$ mix compile
Compiled lib/sequence.ex
Compiled lib/sequence/server.ex
Compiled lib/sequence/stash.ex
Compiled lib/sequence/subsupervisor.ex
Compiled lib/sequence/supervisor.ex
Generated sequence.app
```

Mix tells us it has created a sequence.app file, but where is it? As of Elixir 0.11.2, it is tucked away in _build/shared/lib/sequence/ebin. Although a little obscure, the directory structure under _build is compatible with Erlangs OTP way of doing things. This will make life easier when you come to release your code.

Anyway, let's have a look at sequence.app that was generated.

otp-app/sequence/_build/shared/lib/sequence/ebin/sequence.app

```
{application,sequence,
             [{registered,[sequence]},
              {description,"sequence"},
              {vsn,"0.0.1"},
              {modules,['Elixir.Sequence.Server','Elixir.Sequence.Stash',
                        'Elixir.Sequence.SubSupervisor',
                        'Elixir.Sequence.Supervisor','Elixir.Sequence']},
              {applications,[kernel,stdlib,elixir]},
              {mod,{'Elixir.Sequence',456}}]}.
```

This file contains an Erlang tuple that defines the app. Some of the information comes from the project and application section of mix.exs. Mix also automatically added a list of the names of all the compiled modules in our app (the .beam files) and a list of the apps our app depends on (kernel, stdlib, and elixir). That's pretty smart.

More on Application Parameters

In the previous example, we passed the integer 456 to the application as an initial parameter. Although valid(ish), we really should have passed in a keyword list instead. That's because Elixir provides a function, Application.get_env

Prepared exclusively for Richard Bishop

to retrieve these values from anywhere in your code. So we probably should have set up mix.exs with

```
def application do
  [
    mod: { Sequence, [initial_number: 456] },
    registered: [ :sequence ]
  ]
end
```

and then accessed the value using gen_env:

```
defmodule Sequence do
  use Application

  def start(_type, _args) do
    Sequence.Supervisor.start_link(Application.get_env(:initial_number))
  end

end
```

Your call.

What We Just Did

Let's briefly recap. In that last example, we ran our OTP sequence application using mix. Looking at just our code, two supervisor processes and two worker processes got started. These were knitted together so that our system continued to run with no loss of state even if the worker that we talked to crashed. And any other Erlang process on this node (including iex itself) can talk to our sequence application and enjoy its stream of freshly minted ints.

You probably noticed that the start function takes two parameters. The second corresponds to the value we specified in the mod: option in the mix.exs file (in our case the initial value of the counter). The first parameter specifies the status of the restart, which we're not going to get into, because....

Your turn...

➤ *Exercise: OTP-Applications-1*
Turn your stack server into an OTP application.

➤ *Exercise: OTP-Applications-2*
So far, we haven't written any tests for the application. Is there anything you can test? See what you can do.

Prepared exclusively for Richard Bishop

Hot Code Swapping

You may have heard that OTP applications can update their code while they are running. It's true. In fact, any Elixir program can do it. It's just that OTP provides a release management framework that handles it.

However, OTP release management is complex. If you think about it, something with the potential to deal with dependencies between thousands of processes on hundreds of machines with tens of thousands of modules will, by its nature, be somewhat bigger than a breadbox.

However, I can show you the basics.

First, the real deal is not swapping code, but swapping state. In an application where everything runs as separate processes, swapping code simply means starting a process with the new code, and then sending messages to it.

However, server processes maintain state, and it is likely that changes to the server code will change the structure of the state they hold (adding a field, changing a value, or whatever). So OTP provides a standard server callback which lets a server inherit the state from a prior version of itself.

Alexei Sholik was kind enough to come up with this minimal example of what's possible.

Let's go back to our first version of the supervised sequence server. Its code looked like this.

otp-supervisor/1/sequence/lib/sequence/server.ex

```
defmodule Sequence.Server do
  use GenServer

  #####
  # External API

  def start_link(current_number) do
    GenServer.start_link(__MODULE__, current_number, name: __MODULE__)
  end

  def next_number do
    GenServer.call __MODULE__, :next_number
  end

  def increment_number(delta) do
    GenServer.cast __MODULE__, {:increment_number, delta}
  end

  #####
```

Prepared exclusively for Richard Bishop

```
  # GenServer implementation

  def handle_call(:next_number, _from, current_number) do
    { :reply, current_number, current_number+1 }
  end

  def handle_cast({:increment_number, delta}, current_number) do
    { :noreply, current_number + delta}
  end

  def format_status(_reason, [ _pdict, state ]) do
    [data: [{'State', "My current state is '#{inspect state}', and I'm happy"}]]
  end
end
```

If we want to version our code and data, we have to tell OTP the version numbers of what is running. So, at the top of our module, we'll add an @vsn directive.

otp-app/sequence_reload/lib/sequence/server.ex

```
defmodule Sequence.Server do
  use GenServer
  @vsn "0"
```

Our boss calls. We're about to go for a second round of funding on our wildly successfully sequence server business, but customers have noticed a bug. We implemented increment_number to add a delta to the current number—a one time change. But, apparently, it was instead supposed to set the difference between successive numbers we served.

Let's try the existing code in iex.

```
$ iex -S mix
iex> Sequence.Supervisor.start_link 500
{:ok,#PID<0.57.0>}
iex> Sequence.Server.next_number
500
iex> Sequence.Server.increment_number 10
:ok
iex> Sequence.Server.next_number
511
iex> Sequence.Server.next_number
512
```

Yup, we're only applying the delta once.

Well, that's an easy change to the code. We simply have to keep two things in the state—the current number and the delta to increment it by. We implement the new server code.

Prepared exclusively for Richard Bishop

otp-app/sequence_reload/updated_server.ex
```
defmodule Sequence.Server do
  use GenServer
  @vsn "1"

  # The state representation has been changed to a tuple throughout the module

  def start_link(initial) do
    GenServer.start_link(__MODULE__, initial, name: :sequence)
  end

  def next_number do
    GenServer.call :sequence, :next_number
  end

  def increment_number(delta) do
    GenServer.cast :sequence, {:increment_number, delta}
  end

  def init(initial) when is_number(initial) do
    { :ok, {initial, 1} }
  end

  def handle_call(:next_number, _from, {current_number, delta}) do
    { :reply, current_number, {current_number + delta, delta} }
  end

  def handle_cast({:increment_number, delta}, {current_number, _old_delta}) do
    { :noreply, {current_number + delta, delta}}
  end
end
```

First, notice that we've updated the version number to "1". The other big change is that we changed the state from being just the current number. It is now a tuple of {*current_number*, *delta*}. We updated the increment handler to change the value of delta, and the next number handler now adds in delta each time.

If we simply stop the old server and start the new one, we'll lose the state stored in the old. But we can't just copy the state across—the old server had a single integer, and the new one has a tuple.

Fortunately, OTP has a callback for this. In the new server, implement the `code_change` function.

otp-app/sequence_reload/updated_server.ex
```
def code_change("0", oldState, _extra) do
  IO.puts "Changing code from 0 to 1"
  { :ok, {oldState, 1} }
end
```

Prepared exclusively for Richard Bishop

The callback takes three arguments—the old version number, the old state, and an additional parameter we don't use. The callback's job is to return {:ok, *new_state*). In our case, the new state is a tuple containing the old current number, and a delta of one.

Now, this is where it gets a little unrealistic. In a big, live, application, we'd configure up application and release descriptors, and let the OTP release manager do everything for us. But there's too much fluff in all that for a simple example like this, so we'll cheat and use Erlang's sys module to demonstrate a basic upgrade.

First we suspend the existing server.

```
iex> :sys.suspend :sequence
```

We then compile and load the new version. Note that the module name is the same as the old module name.

```
iex> c("updated_server.ex")
.../sequence_reload/updated_server.ex:1: redefining module Sequence.Server
[Sequence.Server]
```

Now the fun part. We tell OTP to update the new sequence server's state. We pass it the registered name (we can also use a pid), the module name, the previous version number, and an extra argument. (That extra argument gets passed as the third parameter to the code_change callback in our server).

```
iex> :sys.change_code :sequence, Sequence.Server, "0", []
Changing code from 0 to 1
:ok
```

Our callback was indeed triggered—you can see the tracing.

Now let's resume the server, and try out the new behavior.

```
iex> :sys.resume :sequence
:ok
iex> Sequence.Server.next_number
513
iex> Sequence.Server.increment_number 10
:ok
iex> Sequence.next_number
524
iex> Sequence.next_number
534
iex> Sequence.next_number
544
```

We updated the code while the app was running, and just before the investors arrived. Users of our system would not have noticed any interruption.

Prepared exclusively for Richard Bishop

If you're planning on deploying a big Elixir app, you're going to need to think about release management. Many applications don't need it—a little downtime while you restart is acceptable. But if you're aiming for Joe Armstrong's nine-nines reliability, you'll need to work on your supervision structure and release procedures.

But, once you have it set up, you'll find that deploying and updating are an automated and repeatable process.

OTP is Big. Unbelievably Big

This book barely scratches the surface of OTP. But (I hope) it does introduce the major concepts, and give you an idea of the sort of things that are possible.

More advanced uses of OTP may include release management (including hot code swapping), handling distributed failover, automated scaling, and so on. But the chances are that if you have an application that needs such things, you will also already have or will soon need dedicated operations experts who know the low-level details of making OTP apps perform the way you need.

There is never anything simple about scaling out to the kind of size and sophistication that is possible with OTP. But at least you know that you can start small and still get there.

Prepared exclusively for Richard Bishop

In this chapter, we'll see
- Using tasks to run code in the background
- Using agents to maintain state

CHAPTER 19

Tasks and Agents

This part of the book is about processes and process distribution. So far we've covered two extremes. In the first chapters, we looked at the `spawn` primitive, along with message sending and receiving and multinode operations. We then looked at OTP, the 800 pound gorilla of process architecture.

Sometimes, though, we want something in the middle. We want to be able to run simple processes, either for background processing or for maintaining state. But we don't want to be bother with the low-level details of spawn, send, and receive, and we really don't need the extra control that writing our own GenServer gives us.

Enter tasks and agents, two simple-to-use Elixir abstractions. Underneath the covers, these use the features of OTP, but they insulate you from these details.

Tasks

An Elixir task is a function that runs in the background.

tasks/tasks1.exs
```
defmodule Fib do
  def of(0), do: 0
  def of(1), do: 1
  def of(n), do: Fib.of(n-1) + Fib.of(n-2)
end

IO.puts "Start the task"
worker = Task.async(fn -> Fib.of(20) end)
IO.puts "Do something else"
# ...
IO.puts "Wait for the task"
result = Task.await(worker)
```

Prepared exclusively for Richard Bishop

```
IO.puts "The result is #{result}"
```

The call to Task.async creates an separate process that runs the given function. The return value of async is a task descriptor (actually a pid and a ref) that is used to identify it later.

Once the task is running, the code continues with other work. When it wants to get the value of the function, it calls Task.await, passing in the task descriptor. This call waits for our background task to finish, and returns its value.

When we run this, we see:

```
$ elixir tasks1.exs
Start the task
Do something else
Wait for the task
The result is 6765
```

You can also pass Task.async the name of a module and function, along with any arguments:

tasks/tasks2.exs

```
defmodule Fib do
  def of(0), do: 0
  def of(1), do: 1
  def of(n), do: Fib.of(n-1) + Fib.of(n-1)
end

worker = Task.async(Fib, :of, [20])
result = Task.await(worker)
IO.puts "The result is #{result}"
```

Tasks and Supervision

Tasks are implemented as OTP servers, which means you can add them to your application's supervision tree. You can do this in two ways.

First, you can link a task to a currently supervised process by calling start_link instead of async. This actually has less impact than you might think. If the function running in the task crashes and you use start_link, your process will be terminated immediately. If instead you use async, your process will only be terminated when you subsequently call await on the crashed task.

The second way to supervise tasks is to run them directly from a supervisor. This is pretty mush the same as specifying any other worker:

```
import Supervisor.Spec

children = [
  worker(Task, [ fn -> do_something_extraordinary() end ])
```

Prepared exclusively for Richard Bishop

```
]

supervise children, strategy: :one_for_one
```

Agents

An agent is a background process which maintains state. This state can be accessed at different places within a process, node, or across multiple nodes.

The initial state is set by a function you pass in when you start the agent.

You can interrogate the state using Agent.get, passing it the agent descriptor and a function. The agent runs the function on its current state and returns the result.

You can also use Agent.update to change the state held by an agent. As with the get operator, you pass in a function. Unlike get, the result of the function becomes the new state.

Here's a bare-bones example. We start an agent whose state is the integer 0. We then use the identity function to return that state. Calling Agent.update with &(&1+1) increments the state, as verified by a subsequent get.

```
iex> { :ok, count } = Agent.start(fn -> 0 end)
{:ok, #PID<0.69.0>}
iex> Agent.get(count, &(&1))
0
iex> Agent.update(count, &(&1+1))
:ok
iex> Agent.update(count, &(&1+1))
:ok
iex> Agent.get(count, &(&1+1))
2
```

In the previous example, the variable count holds the pid of the agent process. You can also give agents a local or global name, and access them using this name:[1]

```
iex> Agent.start(fn -> 1 end, name: Sum)
{:ok, #PID<0.78.0>}
iex> Agent.get(Sum, &(&1))
1
iex> Agent.update(Sum, &(&1+99))
:ok
iex> Agent.get(Sum, &(&1))
100
```

1. This example exploits the fact that an uppercase bareword in Elixir is converted into an atom with the prefix "Elixir."

Prepared exclusively for Richard Bishop

The following example shows a more typical use. The Frequency module maintains a list of word/frequency pairs in a HashDict. The dictionary itself is stored in an agent, which is named after the module.

This is all initialized with the start_link function, which is presumably invoked during application initialization.

tasks/agent_dict.exs

```
defmodule Frequency do

  def start_link do
    Agent.start_link(fn -> HashDict.new end, name: __MODULE__)
  end

  def add_word(word) do
    Agent.update(__MODULE__,
                 fn dict ->
                      Dict.update(dict, word, 1, &(&1+1))
                 end)
  end

  def count_for(word) do
    Agent.get(__MODULE__, fn dict -> Dict.get(dict, word) end)
  end

  def words do
    Agent.get(__MODULE__, fn dict -> Dict.keys(dict) end)
  end

end
```

We can play with this code in iex.

```
iex> c "agent_dict.exs"
[Frequency]
iex> Frequency.start_link
{:ok, #PID<0.101.0>}
iex> Frequency.add_word "dave"
:ok
iex> Frequency.words
["dave"]
iex(41)> Frequency.add_word "was"
:ok
iex> Frequency.add_word "here"
:ok
iex> Frequency.add_word "he"
:ok
iex> Frequency.add_word "was"
:ok
iex> Frequency.words
["he", "dave", "was", "here"]
```

Prepared exclusively for Richard Bishop

```
iex> Frequency.count_for("dave")
1
iex> Frequency.count_for("was")
2
```

A Bigger Example

Let's rewrite our anagram code to use both tasks and an agent.

We'll load words in parallel from a number of separate dictionaries. Each dictionary is handled by a separate task. We'll use an agent to store the resulting list of words and signatures.

tasks/anagrams.exs

```
defmodule Dictionary do

  @name __MODULE__

  ##
  # External API

  def start_link,
  do: Agent.start_link(fn -> HashDict.new end, name: @name)

  def add_words(words),
  do: Agent.update(@name, &do_add_words(&1, words))

  def anagrams_of(word),
  do: Agent.get(@name, &Dict.get(&1, signature_of(word)))

  ##
  # Internal implementation

  defp do_add_words(dict, words),
  do: Enum.reduce(words, dict, &add_one_word(&1, &2))

  defp add_one_word(word, dict),
  do: Dict.update(dict, signature_of(word), [word], &[word|&1])

  defp signature_of(word),
  do: word |> to_char_list |> Enum.sort |> to_string

end

defmodule WordlistLoader do
  def load_from_files(file_names) do
    file_names
    |> Stream.map(fn name -> Task.async(fn -> load_task(name) end) end)
    |> Enum.map(&Task.await/1)
  end
```

Prepared exclusively for Richard Bishop

```
  defp load_task(file_name) do
    File.open!(file_name)
    |> IO.stream(:line)
    |> Enum.map(&String.strip/1)
    |> Dictionary.add_words
  end
end
```

Our four wordlist files contain:

list1	list2	list3	list4
angor	ester	palet	rogan
argon	estre	patel	ronga
caret	goran	pelta	steer
carte	grano	petal	stere
cater	groan	pleat	stree
crate	leapt	react	terse
creat	nagor	recta	tsere
creta	orang	reest	tepal

Let's run it:

```
$ iex anagrams.exs
iex> Dictionary.start_link
iex> Enum.map(1..4, &"words/list#{&1}") |> WordlistLoader.load_from_files
iex> Dictionary.anagrams_of "organ"
["orang", "nagor", "groan", "grano", "goran", "argon", "angor"]
```

Making it Distributed

Because agents and tasks run as OTP servers, they can already be distributed. All be need to do is to give our agent a globally-accessible name. And that's a one line change:

tasks/anagrams_dist.exs

```
@name {:global, __MODULE__}
```

Now we'll load our code into two separate nodes, and connect them. (Remember that we have to specify names for the nodes in order for them to be able to talk.)

Window #1

```
$ iex --sname one anagrams_dist.exs
iex(node_one@FasterAir)>
```

Window #2

```
$ iex --sname one anagrams_dist.exs
```

Prepared exclusively for Richard Bishop

```
iex(node_two@FasterAir)> Node.connect :node_one@FasterAir
true
iex(node_two@FasterAir)> Node.list
[:node_one@FasterAir]
```

We'll start the dictionary agent in node one, and then load up the dictionary from both nodes one and two:

Window #1

```
iex(node_one@FasterAir)> WordlistLoader.load_from_files(~w{words/list1 words/list2})
[:ok, :ok]
```

Window #2

```
iex(node_one@FasterAir)> WordlistLoader.load_from_files(~w{words/list3 words/list2})
[:ok, :ok]
```

Finally, we'll query the agent from both nodes:

Window #1

```
iex(node_one@FasterAir)> Dictionary.anagrams_of "argon"
["ronga", "rogan", "orang", "nagor", "groan", "grano", "goran", "argon",
 "angor"]
```

Window #2

```
iex(node_two@FasterAir)> Dictionary.anagrams_of "crate"
["recta", "react", "creta", "creat", "crate", "cater", "carte",
"caret"]
```

Prepared exclusively for Richard Bishop

Part III

More Advanced Elixir

One of the joys of Elixir is that it laughs at the concept of "what you see is what you get." Instead, you can extend it in many different ways. This allows you to add layers of abstraction to your code, which in turn makes your code easier to work with.

This part covers macros (which let you extend the syntax of the language), protocols (which let you add behaviors to existing modules), and 'use' (which lets you add capabilities to a module. We finish with a grab bag chapter of miscellaneous other Elixir tricks and tips.

Prepared exclusively for Richard Bishop

CHAPTER 20

Macros And Code Evaluation

Have you ever felt frustrated that the language you are using didn't have just the right feature for some code you were writing? Or did you find yourself repeating chunks of code that weren't amenable to factoring into functions? Or did you just wish that you could program closer to your problem domain?

If so, then you'll love this chapter.

Let's start by seeing why we need macros, and then implement some examples.

But, before we do, here's a warning. Macros can easily make your code harder to understand, because you're essentially rewriting parts of the language. So never use a macro when you could use a function. Let's repeat that:

> Never use a macro when you can use a function

In fact, you'll probably not use a macro in regular application code. But if you're writing a library, and want to use some of the other metaprogramming techniques that we show in later chapters, you'll need to know how macros work.

Implementing an if Statement

Let's imagine Elixir didn't have an `if` statement—all it has is `case`. Although we're prepared to abandon our old friend the `while` loop, not having an `if` statement is just too much to bear, so we set about implementing one.

We'll want to call it using something like

```
myif «condition» do
  «evaluate if true»
else
```

Prepared exclusively for Richard Bishop

```
  «evaluate if false»
end
```

Now, we know that blocks in Elixir are converted into keyword parameters, so this is equivalent to

```
myif «condition»,
  do:   «evaluate if true»,
  else: «evaluate if false»
```

Here's a sample call:

```
My.myif 1==2, do: IO.puts "1 == 2", else: IO.puts "1 != 2"
```

Let's try to implement myif as a function:

```
def myif(condition, clauses) do
  do_clause   = Keyword.get(clauses, :do, nil)
  else_clause = Keyword.get(clauses, :else, nil)

  case condition do
      _ in [false, nil] -> else_clause
      _                 -> do_clause
  end
end
```

But, when we run it, we're (mildly) surprised to get the following output:

```
1 == 2
1 != 2
```

When we call the myif function, Elixir has to evaluate all of its parameters before passing them in. So both the do: and else: clauses are evaluated, and we see their output. Because IO.puts returns :ok on success, what actually gets passed to my_if is

```
my_if 1==2, do: :ok, else: :ok
```

Clearly we need a way of delaying the execution of these clauses. And this is where macros come in. But before we implement our myif macro, we need a little background.

Macros Inject Code

Let's pretend we're the Elixir compiler. We read the source of a module, top-to-bottom, and generate a representation of the code we find. That representation is actually a nested Elixir tuple.

Prepared exclusively for Richard Bishop

If we want to support macros, we need a way to tell the compiler that we'd like to manipulate a part of that tuple. We do that using defmacro, quote, and unquote.

In the same way that def defines a function, we use defmacro to define a macro. We'll see what that looks like shortly. However, the real magic starts not when you define a macro, but when you use one.

When you pass parameters to a macro, Elixir doesn't evaluate them. Instead, it passes them as tuples representing their code. We can examine this behavior using a simple macro definition that prints out its parameter.

macros/dumper.exs

```
defmodule My do
  defmacro macro(param) do
    IO.inspect param
  end
end

defmodule Test do
  require My

  # These values represent themselves
  My.macro :atom        #=> :atom
  My.macro 1            #=> 1
  My.macro 1.0          #=> 1.0
  My.macro [1,2,3]      #=> [1,2,3]
  My.macro "binaries"   #=> "binaries"
  My.macro { 1, 2 }     #=> {1,2}
  My.macro do: 1        #=> [do: 1]

  # And these are represented by 3-element tuples
  My.macro { 1,2,3,4,5 } #=> {:"{}",[line: 20],[1,2,3,4,5]}

  My.macro do: ( a = 1; a+a ) #=>
  #   [do:
  #     {:__block__,[],
  #        [{:=,[line: 22],[{:a,[line: 22],nil},1]},
  #         {:+,[line: 22],[{:a,[line: 22],nil},{:a,[line: 22],nil}]}]}]

  My.macro do #=> [do: {:+,[line: 24],[1,2]}, else: {:+,[line: 26],[3,4]}]
    1+2
  else
    3+4
  end
end
```

This shows us that atoms, numbers, lists (including keyword lists), binaries, and tuples with two elements are represented internally as themselves. All

Prepared exclusively for Richard Bishop

other Elixir code is represented by a three element tuple. Right now, the internals of that representation aren't important.

Load Order

You may be wondering about the structure of the above. We put the macro definition in one module, and the usage of that macro in another. And that second module included a require call.

Macros are expanded before a program executes. That means that the macro definitions must be available as Elixir is compiling a module. The require function tells Elixir that the current module depends on the content of the named module. In practice, it is used to make the macros defined in one module available in another.

But the reason for the two modules is less clear. It's to do with the fact that Elixir first compiles source files into separate modules, and then runs them.

If you have one module per source file, and you reference a module in file A from file B, Elixir will load the module from A, and everything just works. But if you have a module and the code that uses it in the same file, and the module is defined in the same scope in which you use it, Elixir will not know to load the module's code. You'll get the error:

```
** (CompileError)
   .../dumper.ex:7:
   module My is not loaded but was defined. This happens because you
   are trying to use a module in the same context it is defined. Try
   defining the module outside the context that requires it.
```

By placing the code that uses module My in a separate module, we force My to get loaded.

The quote function

We've seen that when we pass parameters to a macro, they are not evaluated. The language also comes with a function, quote that does the same kind of thing. It takes a block and returns the internal representation of that block. We can play with it in iex:

```
iex> quote do: :atom
:atom
iex> quote do: 1
1
iex> quote do: 1.0
1.0
iex> quote do: [1,2,3]
[1,2,3]
```

Prepared exclusively for Richard Bishop

```
iex> quote do: "binaries"
"binaries"
iex> quote do: {1,2}
{1,2}
iex> quote do: [do: 1]
[do: 1]
iex> quote do: {1,2,3,4,5}
{:"{}",[],[1,2,3,4,5]}
iex> quote do: (a = 1; a + a)
{:__block__, [],
 [{:=, [], [{:a, [], Elixir}, 1]},
  {:+, [context: Elixir, import: Kernel],
   [{:a, [], Elixir}, {:a, [], Elixir}]}]}
iex> quote do: [ do: 1 + 2, else: 3 + 4]
[do: {:+, [context: Elixir, import: Kernel], [1, 2]},
 else: {:+, [context: Elixir, import: Kernel], [3, 4]}]
```

There's another way to think about quote. When we write "abc", we're creating a binary containing the bytes 65, 66, and 67. The double quotes say "interpret what follows as a string of characters and return the appropriate representation."

quote is the same: it says "interpret the content of the block that follows as code, and return the internal representation."

Using the representation as code

When we extract the internal representation of some code (either via a macro parameter or using quote), we stop Elixir from adding it automatically to the tuples of code it is building during compilation—we've effectively created a free-standing island of code. How do we get that code injected back into the internal representation of our program?

There are two ways.

The first is our old friend the macro. Just like a function, the value returned by a macro is the last expression evaluated in that macro. That expression is expected to be a fragment of code in Elixir's internal representation. But Elixir does not return this representation to the code that invoked the macro. Instead, it injects this code back into the internal representation of our program, and returns the result of *executing* that code to the caller. But that execution only takes place if needed.

We can demonstrate this in two steps. First, here's a macro that simply returns its parameter (after printing it out). The code we pass it when we invoke the macro is passed as an internal representation, and when the macro returns

Prepared exclusively for Richard Bishop

that code, that representation is injected back into the compile tree, and as a result we see the output.

macros/eg.exs

```
defmodule My do
  defmacro macro(code) do
    IO.inspect code
    code
  end
end

defmodule Test do
  require My

  My.macro(IO.puts("hello"))
end
```

When we run this, we see

```
{{:.,[line: 11],[{:__aliases__,[line: 11],[:IO]},:puts]},
     [line: 11],["hello"]}
hello
```

Now we'll change that file to return a different piece of code. We use quote to generate the internal form:

macros/eg1.exs

```
defmodule My do
  defmacro macro(code) do
    IO.inspect code
    quote do: IO.puts "Different code"
  end
end

defmodule Test do
  require My

  My.macro(IO.puts("hello"))
end
```

This generates:

```
{{:.,[line: 11],[{:__aliases__,[line: 11],[:IO]},:puts]},
     [line: 11],["hello"]}
Different code
```

Even though we passed IO.puts("hello") as a parameter, it was never executed. Instead, the code fragment we returned using quote was.

Prepared exclusively for Richard Bishop

Before we can write our version of if, we need one more trick—the ability to substitute existing code into a quoted block. There are two ways of doing this: using the unquote function and with bindings.

The unquote function

Let's get two things out of the way. First, you can only use unquote inside a quote block. Second, unquote is a silly name. It should really be something like inject_code_fragment.

Let's see why we need this. Here's a simple macro that tries to output the result of evaluating the code that we pass it:

```
defmacro macro(code) do
  quote do
    IO.inspect(code)
  end
end
```

What it actually does is report an error:

```
** (CompileError).../eg2.ex:11: function code/0 undefined
```

Inside the quote block, Elixir is just parsing regular code, so the name code is inserted literally into the code fragment it returns. But we don't want that. We want Elixir to insert the evaluation of the code we pass in. And that's where we use unquote. It temporarily turns off quoting, and simply injects a code fragment into the sequence of code being returned by quote.

```
defmodule My do
  defmacro macro(code) do
    quote do
      IO.inspect(unquote(code))
    end
  end
end
```

Inside the quote block, Elixir is busy parsing the code and generating its internal representation. But when it hits the unquote, it stops parsing and simply copies the code parameter into the generated code. After unquote, it goes back to regular parsing.

There's another way of thinking about this. Using unquote inside a quote is a way of deferring the execution of the unquoted code. It doesn't run when the quote block is parsed. Instead it runs when the code generated by the quote block is executed.

Prepared exclusively for Richard Bishop

And there's still another way of thinking about unquote. Remember we said that quote is analogous to a string literal? We can make a (slightly tenuous) case that unquote is a little like the interpolation you can do in strings. When you write "sum=#{1+2}", Elixir evaluates 1+2 and interpolates the result into the quoted string. When you write quote do: def unquote(name) do end, Elixir interpolates the contents of name into the code representation it is building as part of the list.

Expanding a list—unquote_splicing

Consider this code:

```
iex> Code.eval_quoted(quote do: [1,2,unquote([3,4])])
{[1,2,[3,4]],[]}
```

The list [3,4] is inserted, as a list, into the overall quoted list, resulting in [1,2,[3,4]].

If we instead wanted to insert just the elements of the list, we could use unquote_splicing.

```
iex> Code.eval_quoted(quote do: [1,2,unquote_splicing([3,4])])
{[1,2,3,4],[]}
```

Remembering that single quoted strings are lists of characters, this means you can write:

```
iex> Code.eval_quoted(quote do: [?a, ?= ,unquote_splicing('1234')])
{'a=1234',[]}
```

Back to our myif macro

We now have everything we need to implement a myif macro.

macros/myif.ex

```
defmodule My do

  defmacro if(condition, clauses) do
    do_clause = Keyword.get(clauses, :do, nil)
    else_clause = Keyword.get(clauses, :else, nil)
    IO.inspect do_clause
    quote do
      case unquote(condition) do
        val when val in [false, nil] -> unquote(else_clause)
        _                            -> unquote(do_clause)
      end
    end
  end

end
```

Prepared exclusively for Richard Bishop

```
defmodule Test do
  require My

  My.if 1==2 do
    IO.puts "1 == 2"
  else
    IO.puts "1 != 2"
  end
end
```

It's worth studying this code.

The myif macro receives a condition and a keyword list. The condition and any entries in the keyword list are passed as code fragments.

The macro extracts the do: and/or else: clauses from that list. It is then ready to generate the code for our if statement, so it opens a quote block. That block contains an Elixir case expression. This case expression has to evaluate the condition that is passed in, so it uses unquote to inject that condition's code as its parameter.

When it comes time to execute this case statement, the condition will be evaluated. At that point, case will match the first clause if the result is nil or false, otherwise it matches the second clause. When a clause matches (and only then), we want to execute the code that was passed in either the do: or else: entries in the keyword list, so we use unquote again to inject that code into the case.

Your turn…

➤ *Exercise: MacrosAndCodeEvaluation-1*

Write a macro called myunless that implements the standard *unless* functionality. You're allowed to use the regular if expresssion in it.

➤ *Exercise: MacrosAndCodeEvaluation-2*

Write a macro called times_n that takes a single numeric argument. It should define a function in the module of the caller that itself takes a single argument, and which multiplies that argument by *n*. The new function should be called times_*n*. So, calling times_n(3) should create a function called times_3, and calling times_3(4) should return 12. Here's an example of it in use:

```
defmodule Test do
  require Times
  Times.times_n(3)
  Times.times_n(4)
```

Prepared exclusively for Richard Bishop

```
  end

  IO.puts Test.times_3(4)   #=> 12
  IO.puts Test.times_4(5)   #=> 20
```

Using Bindings to Inject Values

Remember we said there were two ways of injecting values into quoted blocks. One is unquote. The other is to use a binding. However, the two have different uses and different semantics.

A binding is simply a keyword list of variable names and their values. You can pass a binding to quote, and the given variables are set inside the body of that quote.

This is useful because macros are executed at compile time. This means that they don't have access to values that are calculated at runtime.

Here's an example. The intent is to have a macro that defines a function which returns its own name:

```
defmacro mydef(name) do
  quote do
    def unquote(name)(), do: unquote(name)
  end
end
```

We try this out using something like mydef(:some_name). Sure enough, that defines a function that, when called, returns :some_name.

Buoyed by our success, we try something more ambitious:

macros/macro_no_binding.exs

```
defmodule My do
  defmacro mydef(name) do
    quote do
      def unquote(name)(), do: unquote(name)
    end
  end
end

defmodule Test do
  require My

  [ :fred, :bert ] |> Enum.each(&My.mydef(&1))
end

IO.puts Test.fred
```

Prepared exclusively for Richard Bishop

and are rewarded with:

```
macro_no_binding.exs:12: invalid syntax in def _@1()
```

At the time the macro is called, the each loop hasn't yet executed, so we have no valid name to pass it. This is where bindings come in:

macros/macro_binding.exs

```
defmodule My do
  defmacro mydef(name) do
    quote bind_quoted: [name: name] do
      def unquote(name)(), do: unquote(name)
    end
  end
end

defmodule Test do
  require My
  [ :fred, :bert ] |> Enum.each(&My.mydef(&1))
end

IO.puts Test.fred
```

Two things happen here. First, the binding makes the current value of name available inside the body of the quoted block. And, second, the presence of the bind_quoted: option automatically defers the execution of the unquote calls in the body. This way, the methods are defined at runtime.

As its name implies, bind_quoted takes a quoted code fragment. Simple things such as tuples are the same as normal and quoted code. For all others, you probably want to quote the values, or use Macro.escape to ensure that your code fragment will be interpreted correctly.

Macros are Hygienic

It is tempting to think of macros as some kind of textual substitution—the body of the macro is expanded as text and then compiled at the point of call. But that's not the case. Consider this example:

macros/hygiene.ex

```
defmodule Scope do

  defmacro update_local(val) do
    local = "some value"
    result = quote do
      local = unquote(val)
      IO.puts "End of macro body, local = #{local}"
    end
    IO.puts "In macro definition, local = #{local}"
```

Prepared exclusively for Richard Bishop

```
    result
  end
end

defmodule Test do
  require Scope

  local = 123
  Scope.update_local("cat")
  IO.puts "On return, local = #{local}"
end
```

The result of running this is:

```
In macro definition, local = some value
End of macro body, local = cat
On return, local = 123
```

If the macro body was just substituted in at the point of call, both it and the module Test would share the same scope, and the macro would overwrite the variable local, so you'd see

```
In macro definition, local = some value
End of macro body, local = cat
On return, local = cat
```

But that is not the case. Instead, the macro definition has both its own scope and a scope during execution of the quoted macro body. Both are distinct to the scope within the Test module. The upshot of this is that macros will not clobber each others variables, or the variables of modules and functions that use them.

The import and alias functions are also locally scoped. See the documentation for quote for a full description. This also describes how you can turn off hygiene for variables and how to control the format of the stacktrace if things go wrong while executing a macro.

Other Ways to Run Code Fragments

The function Code.eval_quoted can be used to evaluate code fragments, such as those returned by quote.

```
iex> fragment = quote do: IO.puts("hello")
{{:.,[],[{:__aliases__,[alias: false],[:IO]},:puts]},[],["hello"]}
iex> Code.eval_quoted fragment
hello
{:ok,[]}
```

Prepared exclusively for Richard Bishop

By default, the quoted fragment is hygienic, and so does not have access to variables outside its scope. You can disable this feature, and allow a quoted block to access variables in the containing scope, by setting the :hygiene option. In this case, you pass the binding in as a keyword list.

```
iex> fragment = quote(hygiene: [vars: false], do: IO.puts(a))
{{:.,[],[{:__aliases__,[alias: false],[:IO]},:puts]},[],[{:a,[],nil}]}
iex> Code.eval_quoted fragment, [a: "cat"]
cat
{:ok,[a: "cat"]}
```

You can convert code stored in a string into a code fragment using Code.string_to_quoted and convert it from a code fragment back into a string using Macro.to_string:

```
iex> fragment = Code.string_to_quoted("defmodule A do def b(c) do c+1 end end")
{:ok,{:defmodule,[line: 1],[{:__aliases__,[line: 1],[:A]},
[do: {:def,[line: 1],[{:b,[line: 1],[{:c,[line: 1],nil}]},
[do: {:+,[line: 1],[{:c,[line: 1],nil},1]}]]}]]}}
iex> Macro.to_string(fragment)
"{:ok, defmodule(A) do\n  def(b(c)) do\n    c + 1\n  end\nend}"
```

You can also evaluate a string directly using Code.eval_string.

```
iex> Code.eval_string("[a, a*b, c]", [a: 2, b: 3, c: 4])
{[2,6,4],[a: 2, b: 3, c: 4]}
```

Macros and Operators

(This is definitely dangerous ground....)

The unary and binary operators in Elixir can be overridden using macros. When you do so, you'll need to remove any existing definition first.

For example, the operator + (which adds two numbers) is defined in the Kernel module. To remove the Kernel definition and substitute your own, you'd need to do something like the following (which redefines addition to concatenate the string representation of the left and right arguments).

macros/operators.ex

```
defmodule Operators do

  defmacro a + b do
    quote do
      to_string(unquote(a)) <> to_string(unquote(b))
    end
  end

end
```

Prepared exclusively for Richard Bishop

```
defmodule Test do

  IO.puts(123 + 456)   #=> "579"

  import Kernel, except: [+: 2]
  import Operators

  IO.puts(123 + 456)   #=> "123456"

end

IO.puts(123 + 456)   #=> "579"
```

The thing to note here is that the definition of the macro is lexically scoped—the + operator is overridden from the point when we import the Operators module through the end of the module that imports it. We could also have done the import inside a single method, and the scoping would be just that method.

The Macro module has two functions that list the unary and binary operators:

```
iex> require Macro
nil
iex> Macro.binary_ops
[:===, :!==, :==, :!=, :<=, :>=, :&&, :||, :<>, :++, :--, ://, :\\, :::, :<-,
 :.., :|>, :=~, :<, :>, :->, :+, :-, :*, :/, :=, :|, :., :and, :or, :xor, :when,
 :in, :inlist, :inbits, :<<<, :>>>, :|||, :&&&, :^^^, :~~~]
iex> Macro.unary_ops
[:!, :@, :^, :not, :+, :-, :~~~, :&]
```

Digging Deeper

The functions that manipulate the internal representation of code are contained in the Code and Macro modules.

Check the source of the Kernel module for a list of the majority of the operator macros, along with macros for things such as def, defmodule, alias and so on. If you look at the source code, you'll see what the calling sequence is for these. However, many of the bodies will be absent, as the macros are defined within the source of Elixir.

Digging Ridiculously Deep

Here's the internal representation of a simple expression:

```
iex(1)> quote do: 1 + 2
{:+, [context: Elixir, import: Kernel], [1, 2]}
```

Prepared exclusively for Richard Bishop

It's just a three element tuple. In this particular case, the first element is the function (or macro), the second is housekeeping metadata, and the third is the arguments.

We know that we can evaluate this code fragment using eval_quoted (and we can save typing by leaving off the metadata:

```
iex> Code.eval_quoted {:+, [], [1,2]}
{3,[]}
```

And now we can start to see the promise (and dangers) of a homoiconic language. Because code is just tuples, and because we can manipulate those tuples, we take the definitions of existing functions and rewrite them. We can create new code on the fly. And we can do it in a safe way, because we can control the scope of both the changes and of the access to variables.

Next, we'll look at *protocols*, a way of adding functionality to built-in code, and a way of integrating your code into build-in capabilities.

Your turn…

➤ *Exercise: MacrosAndCodeEvaluation-3*

The Elixir test framework, ExUnit, uses some clever code quoting tricks. For example, if you assert

```
assert 5 < 4
```

You'll get the error "expected 5 to be less than 4."

The Elixir source code is on Github (at https://github.com/elixir-lang/elixir). The implementation of this is in the file elixir/lib/ex_unit/lib/ex_unit/assertions.ex. Spend some time reading this file, and work out how it implements this trick.

(Hard) Once you've done that, see if you can use the same technique to implement a function that takes an arbitrary arithmetic expression and returns a natural language version.

```
explain do: 2 + 3*4
#=> multiply 3 and 4, then add 2
```

Prepared exclusively for Richard Bishop

CHAPTER 21

Protocols—Polymorphic Functions

We have used the inspect function many times in this book. It returns a printable representation of any value as a binary (which is what we hard-core folks call strings).

But stop and think for a minute. Just how can Elixir, which doesn't have objects, manage to know just what to call to do the conversion to a binary. You can pass inspect anything, and it somehow makes sense of it.

It *could* be done using guard clauses:

```
def inspect(value) when is_atom(value), do: ...
def inspect(value) when is_binary(value), do: ...
  :    :
```

But there's a better way.

Elixir has the concept of protocols. A protocol is a little like a behaviour, in that it defines the functions that must be provided to achieve something. But a behaviour is internal to a module—the module implements the behaviour. Protocols are different—you can place the implementation of a protocol completely outside the module. This means that you can extend the functionality of modules without having to add code to them—in fact you can extend them even if you don't have their source code.

Defining a Protocol

Protocol definitions are very similar to basic module definitions. They can contain module and function level documentation (@moduledoc and @doc), and they will contain one or more function definitions. However, these functions will not have bodies—they are there simply to declare the interface that the protocol requires.

For example, here is the definition of the Inspect protocol:

Prepared exclusively for Richard Bishop

```
defprotocol Inspect do
  def inspect(thing, opts)
end
```

So, just like a module, the protocol defines one or more functions. But there is no function body, so how does Elixir know what code to run when this function is called?

Implementing a Protocol

The defimpl macro lets you give Elixir the implementation of a protocol for one or more types. The code that follows is the implementation of the Inspect protocol for process IDs and references.

```
defimpl Inspect, for: PID do
  def inspect(pid, _opts) do
    "#PID" <> iolist_to_binary(pid_to_list(pid))
  end
end

defimpl Inspect, for: Reference do
  def inspect(ref, _opts) do
    '#Ref' ++ rest = :erlang.ref_to_list(ref)
    "#Reference" <> iolist_to_binary(rest)
  end
end
```

Finally, the Kernel module implements inspect, which basically calls Inspect.inspect with its parameter. This means that when you call inspect(self), it becomes a call to Inspect.inspect(self). And because self is a PID, this in turn resolves to something like "#PID<0.25.0>".

Behind the scenes, defimpl puts the implementation for each protocol and type combination into a separate module. The protocol for Inspect for the PID type is in the module Inspect.PID. And because you can recompile modules, you can change the implementation of functions implemented via protocols.

```
iex> inspect self
"#PID<0.25.0>"
iex> defimpl Inspect, for: PID do
...>   def inspect(pid, _) do
...>     "#Process: " <> iolist_to_binary(:erlang.pid_to_list(pid)) <> "!!"
...>   end
...> end
iex:3: redefining module Inspect.PID
{:module,
 Inspect.PID,
 <<70,79....
iex> inspect self
```

Prepared exclusively for Richard Bishop

```
"#Process: <0.25.0>!!"
```

The Available Types

You can define implementations for one or more of the following types.

Any	Atom	BitString	Float	Function	Integer
List	PID	Port	Record	Reference	Tuple

The type BitString is used in place of Binary.

The type Any is a catchall, allowing you to match an implementation with any type. Just as with function definitions, you'll want to put the implementations for specific types before an implementation for Any.

You can list multiple types on a single defimpl. For example, the following protocol that can be called to determine if a type is a collection.

protocols/is_collection.exs

```
defprotocol Collection do
  @fallback_to_any true
  def is_collection?(value)
end

defimpl Collection, for: [List, Tuple, BitString] do
  def is_collection?(_), do: true
end

defimpl Collection, for: Any do
  def is_collection?(_), do: false
end

Enum.each [ 1, 1.0, [1,2], {1,2}, HashDict.new, "cat" ], fn value ->
  IO.puts "#{inspect value}:  #{Collection.is_collection?(value)}"
end
```

We write a defimpl stanzas for the three collection types: List, Tuple, BitString. But what about the other types? To handle those, we use the special type Any in a second defimpl. If we use Any, though, we also have to add an annotation to the protocol definition. Thats what the @fallback_to_any line does.

This produces:

```
1:  false
1.0:  false
[1,2]:  true
{1,2}:  true
#HashDict<[]>:  true
"cat":  true
```

Prepared exclusively for Richard Bishop

Your turn…

➤ *Exercise: Protocols-1*

A basic Caesar cypher consists of shifting the letters in a message by a fixed offset. For an offset of 1, for example, a will become b, b will become c, and z will become a. If the offset is 13, we have the ROT13 algorithm.

Lists and binaries can both be *string-like*. Write a Caesar protocol that applies to both. It would include two functions: encrypt(string, shift) and rot13(string).

➤ *Exercise: Protocols-2*

Use a list of words in your language to look for words where rot13(word) is also a word in the list. For various types of English word list, have a look at http://wordlist.sourceforge.net/. The SCOWL collection looks promising, as it already has words divided by size.

Protocols and Structs

Elixir doesn't have classes, but (perhaps surprisingly) it does have user-defined types. It pulls off this magic using structs and a few conventions.

Let's play with a simple struct. Here's the definition:

protocols/basic.exs
```
defmodule Blob do
  defstruct content: nil
end
```

And here we use it in iex:

```
iex> c "basic.exs"
[Blob]
iex> b = %Blob{content: 123}
%Blob{content: 123}
iex> inspect b
"%Blob{content: 123}"
```

It looks for all the world as if we've created some new type, the blob. But that's only because Elixir is hiding something from us. By default, inspect recognizes structs. If we turn this off using the structs: false option, inspect reveals the true nature of our blob value:

```
iex> inspect b, structs: false
"%{__struct__: Blob, content: 123}"
```

A struct value is actually just a map with the key __struct__ referencing the struct's module (Blob in this case) and the remaining elements containing the

Prepared exclusively for Richard Bishop

keys and values for this instance. The inspect implementation for maps checks for this—if you ask it to inspect a map containing a key __struct__ which references a module, it displays it as a struct.

It turns out that many built-in types in Elixir are represented as structs internally. It's instructive to try creating values and inspecting them with records: false.

Built-in Protocols: Access

Let's define a type Bitmap which lets us access the individual bits in the binary representation of a number. To do this, we'll create a struct that contains a single field, value.

protocols/bitmap.exs

```
defmodule Bitmap do
  defstruct value: 0
end
```

The built-in Access protocol defines the [] operator, for accessing members of a collection. We can use this to access the bits in our value, so accessing a bitmap value with value[0] would return the least-significant bit. The implementation that follows uses the Bitwise module that comes with Elixir—this gives us the &&& and <<< bitwise and and shift operators.[1]

protocols/bitmap_access.exs

```
defmodule Bitmap do
  defstruct value: 0

  defimpl Access
    use Bitwise
    def access(%Bitmap{value: value}, bit) do
      if (value &&& (1 <<< bit)) == 0, do: 0, else: 1
    end
  end
end

fifty = %Bitmap{value: 50}

[5,4,3,2,1,0] |> Enum.each(fn bit -> IO.puts fifty[bit] end)
```

This produces:

```
1
1
0
0
```

1. The Access protocol may be removed from future versions of Elixir.

Prepared exclusively for Richard Bishop

```
1
0
```

When we write fifty[bit] on the last line of our code, we're actually invoking the Access protocol. The handler for this sees that its value type is a map, and that the map has a _struct_ key. It looks up the corresponding value, and finds the Bitmap module. It then looks for a module called Bitmap.Access and invokes its access function, passing in the original value and the parameter between the square brackets.[2]

Built-in Protocols: Enumerable

The Enumerable protocol is the basis of all the functions in the Enum module—any type implementing it can be used as a collection argument to Enum functions.

The protocol is defined in terms of three functions:

```
defprotocol Enumerable do
  def count(collection)
  def member?(collection, value)
  def reduce(collection, acc, fun)
end
```

count returns the number of elements in the collection, member? is truthy if the collection contains value, and reduce applies the given function to successive values in the collection and the accumulator, and the value it reduces becomes the next accumulator. Perhaps surprisingly, all the Enum functions can be defined in terms of these three.

However, life isn't quite that simple. Maybe you're using Enum.find to find a value in a large collection. Once you've found it, you want to halt the iteration—continuing on is pointless. Similarly, you may way to suspend an iteration and resume it sometime later. These two features become particular important when we talk about streams, which let you enumerate a collection lazily.

So, let's look at implementing the count part of the enumerable protocol. We return the number of bits required to represent the value.

protocols/bitmap_enumerable.exs

```
defmodule Bitmap do
  defstruct value: 0

  defimpl Enumerable  do
    import :math, only: [log: 1]
```

2. Don't tell anyone, but this is actually quite like method dispatch in an object-oriented language.

Prepared exclusively for Richard Bishop

```
    def count(%Bitmap{value: value}) do
      { :ok, trunc(log(abs(value))/log(2)) + 1 }
    end
  end
end

fifty = %Bitmap{value: 50}

IO.puts Enum.count fifty    # => 6
```

Our count method returns a tuple containing :ok and the actual count. If our collection was not countable (perhaps it represents data coming over a network connection) we would return {:error, __MODULE__}.

I've decided the member? function should return true if the number you pass it is greater than or equal to zero and less that the number of bits in our value. Again the implementation returns a tuple:

protocols/bitmap_enumerable.exs

```
def member?(value, bit_number) do
  { :ok, 0 <= bit_number && bit_number < Enum.count(value) }
end

IO.puts Enum.member? fifty, 4    # => true
IO.puts Enum.member? fifty, 6    # => false
```

However, the meaning of the :ok part is slightly different. You'll normally return {:ok, boolean} for all collections where you know the size, and {:error, __MODULE__} otherwise. In this way, it is like count. However, the reason you do it is different. If you return :ok it means that you have a fast way of determining membership. If you return :error, you're saying that you don't. In this case, the enumerable code will simply perform a linear search.

Finally, we get to reduce. First, remember the general form of the reduce function:

```
reduce(enumerable, accumulator, function)
```

Reduce takes each item in turn from enumerable, passing it and the current value of the accumulator to the function. The value returned by the function becomes the next value of the accumulator.

The reduce function we implement for the Enumerable protocol is the same. But it has some additional conventions associated with it. These conventions are used to manage the early halting and suspension when iterating over streams.

The first convention is that the accumulator value is passed as the second element of a tuple. The first element is a verb telling our reduce function what to do:

Prepared exclusively for Richard Bishop

:cont	continue processing
:halt	terminate processing
:suspend	temporarily suspend processing

The second convention is that the value returned by reduce is another tuple. Again, the second element is the updated accumulator value. The first element passed back the state of the enumerator:

:done	this is the final value—we've reached the end of the enumerable
:halted	we terminated the enumeration because we were passed :halt
:suspended	reponse to a suspend

The suspended case is special. Rather than return a new accumulator, we instead return a function which represents the current state of the enumeration. The library can call this function to kick the enumeration off again.

Once we implement this, our Bitmap can participate in all the features of the Enum module:

protocols/bitmap_enumerable.exs

```
def reduce(bitmap, {:cont, acc}, fun) do
  bit_count =  Enum.count(bitmap)
  _reduce({bitmap, bit_count}, { :cont, acc }, fun)
end

defp _reduce({_bitmap, -1}, { :cont, acc }, _fun), do: { :done, acc }

defp _reduce({bitmap, bit_number}, { :cont, acc }, fun) do
  _reduce({bitmap, bit_number-1}, fun.(bitmap[bit_number], acc), fun)
end

defp _reduce({_bitmap, _bit_number}, { :halt, acc }, _fun), do: { :halted, acc }

defp _reduce({bitmap, bit_number}, { :suspend, acc }, fun),
do: { :suspended, acc, &_reduce({bitmap, bit_number}, &1, fun), fun }

IO.inspect Enum.reverse fifty       # => [0, 1, 0, 0, 1, 1, 0]
IO.inspect Enum.join fifty, ":"     # => "0:1:1:0:0:1:0"
```

If you think this is complicated—well, you're correct. It is. But part of the reason is that these conventions allow all enumerable values to be used both eagerly and lazily. And when you're dealing with big (or even infinite) collections, this is a big deal.

Prepared exclusively for Richard Bishop

Built-in Protocols: String.Chars

The String.Chars protocol is used to convert a value to a string (binary). It consists of a single method, to_string. This is the protocol used for string interpolation:

protocols/bitmap_string.exs

```
defmodule Bitmap do
  defstruct value: 0

  defimpl String.Chars do
    def to_string(value), do: Enum.join(value, "")
  end
end

fifty = %Bitmap{value: 50}

IO.puts "Fifty in bits is #{fifty}"  # => Fifty in bits is 0110010
```

Built-in Protocols: Inspect

This is the protocol that is used to inspect a value. The rule is simple—if you can return a representation that is a valid Elixir literal, do so. Otherwise, prefix the representation with #Typename.

We *could* just delegate the inspect function to the Elixir default. For our value 50, this would be %Bitmap{value: 50}. But let's override it. We need to implement the inspect function. It takes a value and some options.

protocols/bitmap_inspect.exs

```
defmodule Bitmap do
  defstruct value: 0

  defimpl Inspect do
    def inspect(bitmap=%Bitmap{value: value}, _opts = Inspect.Opts[]) do
      "%Bitmap{#{value}=#{bitmap}}"
    end
  end
end

fifty = %Bitmap{value: 50}

IO.inspect fifty                    # => %Bitmap{50=0110010}
IO.inspect fifty, structs: false    # => %{__struct__: Bitmap, value: 50}
```

Run this, and you'll see

```
#Bitmap[50=0110010]
{Bitmap,50}
```

Prepared exclusively for Richard Bishop

There's a wrinkle here. If you pass structs: true to IO.inspect (or Kernel.inspect), it never calls our inspect function. Instead, it just formats it as a tuple.

The formatting of our bitmap leaves a little to be desired for large numbers:

```
iex> Bitmap.new(value: 12345678901234567890)
%Bitmap{12345678901234567890=01010101101010100101010011000110011101
10110001111100001010110100 10}
```

The output was all on one line, and was wrapped by the console. To fix this, we use a feature called *algebra documents*. An algebra document[3] is simply a tree structure that represents some data you'd like to pretty print. Your job is to create the structure based on the data you want to inspect, and Elixir will then find a nice way to display it.

In our case, I'd like the bitmap values to display on a single line if they fit, and I'd like them to break intelligently onto multiple lines if not.

We do this by having our inspect function return an algebra document, rather than a string. In that document, we indicate places where breaks are allowed (but not required) and we show how the nesting works.

protocols/bitmap_algebra.exs

```
defmodule Bitmap do
  defstruct value: 0

  defimpl Inspect, for: Bitmap do
    import Inspect.Algebra
    def inspect(bitmap=%Bitmap{value: value}, _opts = Inspect.Opts[]) do
      concat([
        nest(
         concat([
           "%Bitmap{",
           break(""),
           nest(
             concat([
               to_string(value),
               "=",
               break(""),
               to_string(bitmap)
             ]), 2),
         ]), 2),
        break(""),
        "}"])
    end
  end
```

3. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.2200

Prepared exclusively for Richard Bishop

```
end

big_bitmap = %Bitmap{value: 12345678901234567890}

IO.inspect big_bitmap
IO.inspect big_bitmap, structs: false
```

and we get the output:

```
iex> %Bitmap{value: 12345}
%Bitmap{12345=011000000111001}
iex> %Bitmap{value: 123456789123456789}
%Bitmap{
  123456789123456789=
    0110110110100110110100101110101100110100000101111100010101
}
```

For more information, see the documentation for Inspect.Algebra.

Protocols are Polymorphism

When you find yourself wanting to write a function that behaves differently depending on the type of its arguments, you're looking at a polymorphic function. And Elixir protocols give you a tidy and controlled way to implement this. Whether you're integrating your types into the existing Elixir library, or creating a new library with a flexible interface, protocols let you package the behaviour in well-documented and disciplined way.

Your turn…

➤ *Exercise: Protocols-3*

Collections that implement the Enumerable protocol define count, member?, and reduce functions. The Enum module uses these to implement methods such as each, filter, and map.

Implement your own versions of each, filter, and map in terms of reduce.

➤ *Exercise: Protocols-4*

In many cases, inspect will return a valid Elixir literal for the value being inspected. Update the inspect function for structs so that it returns valid Elixir code to contruct a new struct equal to the value being inspected.

Prepared exclusively for Richard Bishop

CHAPTER 22

Linking Modules: Behavio(u)rs and Use

When we wrote our OTP server, we wrote modules that started with code such as

```
defmodule Sequence.Server do
  use GenServer.Behaviour
  ...
```

In this chapter we'll explore what lines such as `use GenServer.Behaviour` actually do, and how you can write modules that extend the capabilities of other modules using them.

Behaviours

An Elixir behaviour is nothing more than a list of functions. A module that declares that it implements a particular behaviour must implement all of these functions. If it doesn't, Elixir will generate a compilation warning.

A behaviour is therefore a little like an *interface* in Java. It is used by a module to declare that it implements a particular interface. For example, an OTP GenServer should implement a standard set of callbacks (`handle_call`, `handle_cast`, and so on). By declaring that our module implements that behaviour, we let the compiler validate that we have done so. This reduces the chance of an unexpected runtime error.

We define a behaviour using the Elixir `Behaviour` module, combined with `defcallback` definitions.

For example, Elixir comes with a URI parsing library. This library delegates a couple of functions to protocol-specific libraries (so there's a library for HTTP, one for FTP, and so on). These protocol-specific libraries must define two functions, `parse` and `default_port`.

Prepared exclusively for Richard Bishop

The interface to these sublibraries is defined in a module called URI.Parser. It looks like this:

```
defmodule URI.Parser do
  @moduledoc """
  Defines the behavior for each URI.Parser.
  Check URI.HTTP for a possible implementation.
  """

  use Behaviour

  @doc """
  Responsible for parsing extra URL information.
  """
  defcallback parse(uri_info :: URI.Info.t) :: URI.Info.t

  @doc """
  Responsible for returning the default port.
  """
  defcallback default_port() :: integer
end
```

This module defines the interface that modules implementing the behaviour must support. There are two parts to this. First, it has the line use Behaviour. This adds the functionality we need to define behaviours.

Next, it uses defcallback to define the functions in the behaviour. But the syntax looks a little different. That's because we're using a minilanguage called Erlang type specifications. For example, the parse function takes a single parameter which should be a URI.Info record, and it returns a value of the same type. The default_port function takes no arguments and returns an integer. For more information on these type specifications, see Appendix 2, *Type Specifications and Type Checking*, on page 295.

As well as the type specification, we can include module and function-level documentation with our behaviour definitions.

Having defined the behaviour, we can declare that some other module implements it using the @behaviour attribute.

```
defmodule URI.HTTP do
  @behaviour URI.Parser
  def default_port(), do: 80
  def parse(info), do: info
end
```

This module will compile cleanly. However, imagine we'd misspelled default_port:

```
defmodule URI.HTTP do
```

Prepared exclusively for Richard Bishop

```
  @behaviour URI.Parser
  def default_prot(), do: 80
  def parse(info), do: info
end
```

When we compiled the module, we'd get the error

```
http.ex:8: undefined behaviour function default_port/0 (for
behaviour URI.Parser)
```

So behaviours give us a way of both documenting and enforcing the public functions that a module should implement.

Use and _using_

In one sense, use is a trivial function. You pass it a module, along with an optional argument, and it invokes the function or macro _using_ in that module, passing it the argument.

And yet this simple interface gives you a powerful extension facility. For example, in our unit tests we write use ExUnit.Case and we get the test macro and assertion support. When we write an OTP server, we write use GenServer.Behaviour and we both get a behaviour which documents the gen_server callback, but also default implementations of those callbacks.

Typically, the _using_ callback will be implemented as a macro, as it will be used to invoke code in the original module.

Putting it Together—Tracing Method Calls

Let's work through a larger example. We want to write a module called Tracer. If we use Tracer in another module, entry and exit tracing will be added to any subsequently defined function. For example, given the following:

use/tracer.ex

```
defmodule Test do
  use Tracer
  def puts_sum_three(a,b,c), do: IO.inspect(a+b+c)
  def add_list(list),        do: Enum.reduce(list, 0, &(&1+&2))
end

Test.puts_sum_three(1,2,3)
Test.add_list([5,6,7,8])
```

we'd get the following output:

```
==> call    puts_sum_three(1, 2, 3)
6
<== returns 6
```

Prepared exclusively for Richard Bishop

```
==> call    add_list([5,6,7,8])
<== returns 26
```

My approach to writing this kind of code is to start by exploring what we have to work with, and then to generalize. The goal is to metaprogram as little as possible.

It looks as if we have to override the def macro, which is defined in Kernel. So let's do that, and see what gets passed to def when we define a method.

use/tracer1.ex

```
defmodule Tracer do

  defmacro def(definition, do: _content) do
    IO.inspect definition
    quote do: {}
  end

end

defmodule Test do

  import Kernel, except: [def: 2]
  import Tracer, only:   [def: 2]

  def puts_sum_three(a,b,c), do: IO.inspect(a+b+c)
  def add_list(list),        do: Enum.reduce(list, 0, &(&1+&2))
end

Test.puts_sum_three(1,2,3)
Test.add_list([5,6,7,8])
```

This outputs:

```
{:puts_sum_three, [line: 15],
 [{:a, [line: 15], nil}, {:b, [line: 15], nil}, {:c, [line: 15], nil}]}
{:add_list, [line: 16], [{:list, [line: 16], nil}]}
tracer1.ex:12: unused import Kernel
** (UndefinedFunctionError) undefined function: Test.puts_sum_three/3
```

So the definition part of each method is a three-element tuple. The first element is the name, the second the line on which it is defined, and the third a list of the parameters, where each parameter is itself a tuple.

We also get an error: puts_sum_three is undefined. That's not surprising—we intercepted the def that defined it, and we didn't create the function.

You may be wondering about the form of the macro definition: defmacro def(definition, do: _content).... The do: in the parameters is not special syntax: it's

Prepared exclusively for Richard Bishop

simply a pattern match on the block passed as the function body, which is a keyword list.

You may also be wondering whether we have in any way affected the built-in Kernel.def macro. The answer is "no." We've created another macro, also called def which is defined in the scope of the Tracer module. In our Test module we tell Elixir not to import the Kernel version of def, but instead to import the version from Tracer. We'll shortly make use of the fact that the original Kernel implementation is unaffected.

Now let's see if we can actually define a real function given this information. That turns out to be surprisingly easy. We already have the two arguments passed to def. All we have to do is pass them on.

use/tracer2.ex

```
defmodule Tracer do

  defmacro def(definition, do: content) do
    quote do
      Kernel.def(unquote(definition)) do
        unquote(content)
      end
    end
  end

end

defmodule Test do

  import Kernel, except: [def: 2]
  import Tracer, only:   [def: 2]

  def puts_sum_three(a,b,c), do: IO.inspect(a+b+c)
  def add_list(list),        do: Enum.reduce(list, 0, &(&1+&2))
end

Test.puts_sum_three(1,2,3)
Test.add_list([5,6,7,8])
```

When we run this, we see "6", the output from puts_sum_three.

Now it's time to add some tracing.

use/tracer3.ex

```
defmodule Tracer do

  def dump_args(args) do
    args |> Enum.map(&inspect/1) |> Enum.join(", ")
  end
```

Prepared exclusively for Richard Bishop

```
  def dump_defn(name, args) do
    "#{name}(#{dump_args(args)})"
  end

  defmacro def(definition={name,_,args}, do: content) do
    quote do
      Kernel.def(unquote(definition)) do
        IO.puts "==> call:   #{Tracer.dump_defn(unquote(name), unquote(args))}"
        result = unquote(content)
        IO.puts "<== result: #{result}"
        result
      end
    end
  end

end

defmodule Test do

  import Kernel, except: [def: 2]
  import Tracer, only:   [def: 2]

  def puts_sum_three(a,b,c), do: IO.inspect(a+b+c)
  def add_list(list),        do: Enum.reduce(list, 0, &(&1+&2))
end

Test.puts_sum_three(1,2,3)
Test.add_list([5,6,7,8])
```

Looking good:

```
==> call:   puts_sum_three(1, 2, 3)
6
<== result: 6
==> call:   add_list([5,6,7,8])
<== result: 26
```

Now let's package our Tracer module, so clients only have to add use Tracer to their own modules. We'll implement the __using__ callback. The tricky part here is differentiating between the two modules: Tracer and the module that uses it.

use/tracer4.ex

```
defmodule Tracer do

  def dump_args(args) do
    args |> Enum.map(&inspect/1) |> Enum.join(", ")
  end

  def dump_defn(name, args) do
    "#{name}(#{dump_args(args)})"
```

Prepared exclusively for Richard Bishop

```
  end

  defmacro def(definition={name,_,args}, do: content) do
    quote do
      Kernel.def(unquote(definition)) do
        IO.puts "==> call:   #{Tracer.dump_defn(unquote(name), unquote(args))}"
        result = unquote(content)
        IO.puts "<== result: #{result}"
        result
      end
    end
  end

  defmacro __using__(_opts) do
    quote do
      import Kernel, except: [def: 2]
      import unquote(__MODULE__), only: [def: 2]
    end
  end
end

defmodule Test do
  use Tracer
  def puts_sum_three(a,b,c), do: IO.inspect(a+b+c)
  def add_list(list),        do: Enum.reduce(list, 0, &(&1+&2))
end

Test.puts_sum_three(1,2,3)
Test.add_list([5,6,7,8])
```

Your turn…

➤ *Exercise: LinkingModules-BehavioursAndUse-1*

In the body of the def macro, there's a quote block that defines the actual method. It contains:

```
IO.puts "==> call:   #{Tracer.dump_definition(unquote(name), unquote(args))}"
result = unquote(content)
IO.puts "<== result: #{result}"
```

Why does the first call to puts have to unquote the values in its interpolation, but the second call does not?

➤ *Exercise: LinkingModules-BehavioursAndUse-2*

The built-in function IO.ANSI.escape will insert ANSI escape sequences in a string. If you put the resulting strings to a terminal, you can add colors and bold or underlined text. Explore the library, and then use it to colorize the output of our tracing.

Prepared exclusively for Richard Bishop

➤ *Exercise: LinkingModules-BehavioursAndUse-3*

(Hard). Try adding a method definition with a guard clause to the Test module. You'll find that the tracing no longer works.

- Find out why
- See if you can fix it

Prepared exclusively for Richard Bishop

CHAPTER 23

More Cool Stuff

Elixir is packed with features that make coding a joy. This chapter contains a smattering of them.

Writing Your Own Sigils

You know by now that you can create strings and regular expression literals using sigils:

```
string = ~s{now is the time}
regex  = ~r{..h..}
```

Have you ever wished you could extend these sigils to add your own specific literal types? You can.

When you write a sigil such as ~s{...}, Elixir converts in into a call to the function sigil_s. It passes the function two values. The first is the string between the delimiters. The second is a list containing any lowercase letters that immediately follow the closing delimiter. (This second parameter is used to pick up any options you pass to a regex literal, such as ~r/cat/if.)

Here's the implementation of a sigil ~l that takes a multiline string and returns a list containing each line as a separate string. We know that ~l... is converted into a call to sigil_l, so we just write a simple function in the LineSigil module.

odds/line_sigil.exs

```
defmodule LineSigil do

  @doc """

  Implement the `~l` sigil, which takes a string containing
  multiple lines and returns a list of those lines.

  ## Example usage
```

Prepared exclusively for Richard Bishop

```
    iex> import LineSigil
    nil
    iex> ~l"""
    ...> one
    ...> two
    ...> three
    ...> """
    ["one","two","three"]

  """

  def sigil_l(lines, _opts) do
    lines |> String.rstrip |> String.split("\n")
  end

end
```

We can play with this in a separate module:

odds/line_sigil.exs

```
defmodule Example do
  import LineSigil

  def lines do
    ~l"""
    line 1
    line 2
    and another line in #{__MODULE__}
    """
  end

end

IO.inspect Example.lines
```

This produces ["line 1","line 2","and another line in Elixir.Example"].

Because we import the sigil_l function inside the example module, the ~l sigil is lexically scoped to this module. Note also that Elixir performs interpolation before passing the string to our method. That's because we used a lower-case l. If our sigil was ~L{}, and the function renamed sigil_L, no interpolation would be performed.

The predefined sigil functions are 'sigil_C', 'sigil_c', 'sigil_R', 'sigil_r', 'sigil_S', 'sigil_s', 'sigil_W', and 'sigil_w. If you want to override one of these, you'll need to explicitly import the Kernel module and use an except clause to exclude it.

In this example, we used the heredoc syntax ("""). This passes our function a multiline string with leading spaces removed. Sigil options are not supported with heredocs, so we'll switch to a regular literal syntax to play with them.

Prepared exclusively for Richard Bishop

Picking up the Options

Let's write a sigil that let's us specify color constants. If we say ~c{red}, we'll get 0xff0000, the RGB representation. We'll also support the option h to return an HSB value, so ~c{red}h will be {0,100,100}.

Here's the code:

odds/color.exs

```
defmodule ColorSigil do

  @color_map [
    rgb: [
          red: 0xff0000, green: 0x00ff00, blue: 0x0000ff, # ...
         ],
    hsb: [
          red: {0,100,100}, green: {120,100,100}, blue: {240,100,100}
         ]
  ]


  def sigil_c(color_name, []),  do: _c(color_name, :rgb)
  def sigil_c(color_name, 'r'), do: _c(color_name, :rgb)
  def sigil_c(color_name, 'h'), do: _c(color_name, :hsb)

  defp _c(color_name, color_space) do
    @color_map[color_space][binary_to_atom(color_name)]
  end

  defmacro __using__(_opts) do
    quote do
      import Kernel, except: [sigil_c: 2]
      import unquote(__MODULE__), only: [sigil_c: 2]
    end
  end
end

defmodule Example do

  use ColorSigil

  def rgb, do: IO.inspect ~c{red}
  def hsb, do: IO.inspect ~c{red}h
end

Example.rgb   #=> 16711680  (== 0xff0000)
Example.hsb   #=> {0,100,100}
```

The three clauses for the sigil_c function let us select the colorspace to use based on the option passed. As the single quoted string 'r' is actually repre-

Prepared exclusively for Richard Bishop

sented by the list [?r], we can use the string literal to pattern match the options parameter.

Because I'm overriding a built-in sigil, I decided to implement a _using_ macro which automatically removes the Kernel version and adds our own (but only in the lexical scope that calls use on our module).

The fact that you can write your own sigils is liberating. But, at the same time, misuse could lead to some pretty impenetrable code.

Your turn…

➤ *Exercise: MoreCoolStuff-1*

Write a sigil ~v that parses multiple lines of comma-separated data, returning a list where each element is a row of data, and each row is a list of values. Don't worry about quoting—just assume that each field is separated by a comma. So

```
csv = ~v"""
1,2,3
cat,dog
"""
```

Would generate [["1","2","3"], ["cat","dog"]]

➤ *Exercise: MoreCoolStuff-2*

The function Float.parse converts leading characters of a string to a float, returning either a tuple containing the value and the rest of the string, or the atom :error.

Update your CSV sigil so that numbers are automatically converted:

```
csv = ~v"""
1,2,3.14
cat,dog
"""
```

Would generate [[1.0,2.0,3.14], ["cat","dog"]]

➤ *Exercise: MoreCoolStuff-3*

(Harder) Sometimes the first line of a CSV file is a list of the column names. Update your code to support this, and return the values in each row as a keyword list using the column names as the keys.

```
csv = ~v"""
Item,Qty,Price
Teddy bear,4,34.95
Milk,1,2.99
Battery,6,8.00
```

Prepared exclusively for Richard Bishop

```
"""
```

Would generate:

```
[
  [Item: "Teddy bear", Qty: 4, Price: 34.95],
  [Item: "Milk", Qty: 1, Price: 2.99],
  [Item: "Battery", Qty: 6, Price: 8.00]
]
```

MultiApp Umbrella Projects

It is unfortunate that Erlang chose to call self-contained bundles of code *apps*. In many ways, they are closer to being shared libraries. And as your projects grow, you may find yourself wanting to split your code into multiple libraries, or apps. Fortunately, mix makes this painless.

To illustrate the process, we'll create a simple Erlang evaluator. Given a set of input lines, it will return the result of evaluating each. This will be one app.

To test it, we'll need to pass in lists of lines. And we've already written a trivial ~l sigil that creates lists of lines for us. So we'll make that sigil code into a separate application.

Elixir calls these multi-app projects umbrella projects.

Create An Umbrella Project

We use mix new to create an umbrella project, passing it the --umbrella option.

```
$ mix new --umbrella eval
* creating README.md
* creating mix.exs
* creating apps
```

Compared to a normal mix project, the umbrella is pretty lightweight—just a mix file and an apps directory.

Create the Subprojects

Subprojects are stored in the apps directory. There's nothing special about them—they are simply regular projects created using mix new.[1] Let's create our two projects now:

1. This means that you don't have to worry about whether to start a new project using an umbrella or not. Simply start as a simple project. If you later discover the need for an umbrella project, create it, and move your existing simple project into the apps directory.

Prepared exclusively for Richard Bishop

```
$ cd eval/apps
$ mix new line_sigil
* creating README.md
 ... and so on
$ mix new evaluator
* creating README.md
 ... and so on
* creating test/evaluator_test.exs
```

At this point we can try out our umbrella project. Go back to the overall project directory, and try `mix compile`.

```
$ cd ..
$ mix compile
==> evaluator
Compiled lib/evaluator.ex
Generated evaluator.app
==> line_sigil
Compiled lib/line_sigil.ex
Generated line_sigil.app
```

So now we have an umbrella project containing two regular projects. Because there's nothing special about the subprojects, you can use all the regular mix commands in them. At the top-level, though, you can build all the subprojects as a unit.

The Line Sigil Project

This project is trivial—just copy the `LineSigil` module from the previous section into `apps/line_sigil/lib/line_sigil.ex`. Verify it builds by running `mix compile`, either in the top-level directory or the `line_sigil` directory.

The Evalutor Project

The evaluator takes a list of strings containing Elixir expressions and evaluates them. It returns a list containing the expressions intermixed with the value of each. For example, given

```
a = 3
b = 4
a + b
```

Our code will return

```
code>  a = 3
value> 3
code>  b = 4
value> 4
code>  a + b
value> 7
```

Prepared exclusively for Richard Bishop

We'll use Code.eval_string to execute the Elixir expressions. In order to have the values of variables pass from one expression to the next, we'll also have to explicitly maintain the current binding.

Here's the code

odds/eval/apps/evaluator/lib/evaluator.ex

```
defmodule Evaluator do

  def eval(list_of_expressions) do
    { result, _final_binding } =
        Enum.reduce(list_of_expressions,
                    {_result = [], _binding=[]},
                    &evaluate_with_binding/2)
    Enum.reverse result
  end

  defp evaluate_with_binding(expression, { result, binding }) do
    { next_result, new_binding } = Code.eval_string(expression, binding)
    { [ "value> #{next_result}", "code>  #{expression}" | result ], new_binding }
  end
end

IO.inspect Evaluator.eval(["a = 1", "b=2", "a+b"])
```

Linking the Subprojects

Now we need to test our evaluator. It makes sense to use our ~l sigil to create lists of expressions, so let's write our tests that way.

Here are some of the tests we want to write:

odds/eval/apps/evaluator/test/evaluator_test.exs

```
defmodule EvaluatorTest do
  use ExUnit.Case

  import LineSigil

  test "evaluates a basic expression" do
    input = ~l"""
    1 + 2
    """

    output = ~l"""
    code>  1 + 2
    value> 3
    """

    run_test input, output
  end
```

Prepared exclusively for Richard Bishop

```
  test "variables are propogated" do
    input = ~l"""
    a = 123
    a + 1
    """
    output = ~l"""
    code>  a = 123
    value> 123
    code>  a + 1
    value> 124
    """

    run_test input, output
  end

  defp run_test(lines, output) do
    assert output == Evaluator.eval(lines)
  end
end
```

But if we simply run this, it won't be able to find the LineSigil module. To remedy that, we need to add it as a dependency of our project. But we want that dependency only in the test environment, so our mix.exs gets a little more complicated.

odds/eval/apps/evaluator/mix.exs

```
defmodule Evaluator.Mixfile do
  use Mix.Project

  def project do
    [ app: :evaluator,
      version: "0.0.1",
      deps: deps(Mix.env) ]
  end

  # Configuration for the OTP application
  def application do
    [ ]
  end

  defp deps(:test) do
     [ { :line_sigil, path: "../line_sigil" } ] ++ deps(:default)
  end

  defp deps(_) do
    []
  end
end
```

Now we can run tests from the top-level directory.

Prepared exclusively for Richard Bishop

```
$ mix test
...

Finished in 0.06 seconds (0.06s on load, 0.00s on tests)
3 tests, 0 failures


Finished in 0.00 seconds
0 tests, 0 failures
```

The first stanza of test output is for the evaluator tests, and the second is for line_sigil, which currently is test free.

Prepared exclusively for Richard Bishop

In this chapter, we'll see
- raising exceptions to indicate errors
- handling exceptions
- catching other errors
- custom exceptions

APPENDIX 1

Exceptions: raise and try, catch and throw

Elixir (and Erlang) take the view that errors should normally be fatal to the process in which they occur. The design of a typical Elixir application will involve many processes, so this means that the effects of the error will be localized. The failing process will be detected by a supervisor, and the restart will be handled at that level.

For that reason, you won't find much exception handling code in Elixir programs. Exceptions are raised, but you rarely catch them.

Use exceptions for things that are exceptional—that should never happen.

Exceptions do exist. This appendix is an overview of how to generate them, and how to catch them when they occur.

Raising an Exception

You can raise an exception using the raise function. At its simplest, you pass it a string, and it generates an exception of type RuntimeError.

```
iex> raise "Giving up"
** (RuntimeError) Giving up
    erl_eval.erl:572: :erl_eval.do_apply/6
```

You can also pass the type of the exception, along with other optional fields. All exceptions implement at least the message field.

```
iex> raise RuntimeError
** (RuntimeError) runtime error
    erl_eval.erl:572: :erl_eval.do_apply/6
iex> raise RuntimeError, message: "override message"
** (RuntimeError) override message
```

Prepared exclusively for Richard Bishop

```
erl_eval.erl:572: :erl_eval.do_apply/6
```

You can intercept exceptions using the try function. It takes a block of code to execute, and optional rescue, catch and after clauses.

The rescue and catch clauses look a bit like the body of a case function—they take patterns and code to execute if the pattern matches. The subject of the pattern is the exception that was raised.

Here's an example of exception handling in action. We define a module that has a public function, start. It calls a different helper function depending on the value of its parameter. With zero, it runs smoothly. With 1, 2, or 3, it causes the VM to raise an error, which we catch and report.

exceptions/exception.ex

```
defmodule Boom do

  def start(n) do
    try do
      raise_error(n)
    rescue
      [ FunctionClauseError, RuntimeError ] ->
        IO.puts "no function match or runtime error"
      error in [ArithmeticError]  ->
        IO.puts "Uh, oh! Arithmetic error: #{error.message}"
        raise error, [ message: "too late, we're doomed"], System.stacktrace
      other_errors ->
        IO.puts "Disaster! #{other_errors.message}"
    after
        IO.puts "DONE!"
    end
  end

  defp raise_error(0) do
    IO.puts "No error"
  end

  defp raise_error(1) do
    IO.puts "About to divide by zero"
    1 / 0
  end

  defp raise_error(2) do
    IO.puts "About to call a function that doesn't exist"
    raise_error(99)
  end

  defp raise_error(3) do
    IO.puts "About to try creating a directory with no permission"
    File.mkdir!("/not_allowed")
```

Prepared exclusively for Richard Bishop

```
  end
end
```

We define three different exception patterns. The first matches one of the two exceptions, FunctionClauseError or RuntimeError. The second matches an ArithmeticError, and stores the exception value in the variable error. And the last clause catches *any* exception into the variable other_error.

We also include an after clause. This will always run at the end of the try function, regardless of whether an exception was raised or not.

Finally, have a look at the handling of ArithmeticError. As well as reporting the error, we call raise again, passing in the existing exception but overriding its message. We also pass in the stacktrace (which is actually the stacktrace at the point the original exception was raised). Let's see all this in iex:

```
iex c("exception.ex")
.../exception.ex:19: this expression will fail with a 'badarith' exception
[Boom]
iex> Boom.start 1
About to divide by zero
Uh, oh! Arithmetic error: bad argument in arithmetic expression
DONE!
** (ArithmeticError) too late, we're doomed
    exception.ex:25: Boom.raise_error/1
    exception.ex:5: Boom.start/1

iex> Boom.start 2
About to call a function that doesn't exist
no function match or runtime error
DONE!
:ok
iex> Boom.start 3
About to try creating a directory with no permission
Disaster! could not make directory /not_allowed: permission denied
DONE!
:ok
```

catch, exit, and throw

Elixir code (and the underlying Erlang libraries) can raise a second kind of error. These are generated when a process calls error, exit, or throw. All three take a parameter, which is available to the catch handler.

Here's an example:

exceptions/catch.ex

```
defmodule Catch do
```

Prepared exclusively for Richard Bishop

```
  def start(n) do
    try do
      incite(n)
    catch
      :exit, code   -> "Exited with code #{inspect code}"
      :throw, value -> "throw called with #{inspect value}"
      what, value   -> "Caught #{inspect what} with #{inspect value}"
    end
  end


  defp incite(1) do
    exit(:something_bad_happened)
  end

  defp incite(2) do
    throw {:animal, "wombat"}
  end

  defp incite(3) do
    :erlang.error "Oh no!"
  end
end
```

Calling the start function with 1, 2, or 3 will cause an exit, a throw, or an error to be thrown. Just to illustrate wildcard pattern matching, we handle the last case by matching any type into the variable what.

```
iex> c("catch.ex")
[Catch]
iex> Catch.start 1
"Exited with code :something_bad_happened"
iex> Catch.start 2
"throw called with {:animal,\"wombat\"}"
iex> Catch.start 3
"Caught :error with \"Oh no!\""
```

Defining Your Own Exceptions

Exceptions in Elixir are basically records. You can define your own exceptions using defexception. This takes the exception name, a keyword list of fields and their default values, and an optional block in which you can define additional functions for the exception.

Say we're writing a library to talk to a Kinect controller. It might want to raise an exception on various kinds of communications error. Some of these are permanent, but others are likely to be transient, and can be retried. We'll define our exception with its (required) message field and an additional can_retry

Prepared exclusively for Richard Bishop

field. We'll also add a function to it that formats these two fields into a nice message.

exceptions/defexception.ex
```
defexception KinectProtocolError,
              message: "Kinect protocol error",
              can_retry: false do
  def full_message(me) do
    "Kinect failed: #{me.message}, retriable: #{me.can_retry}"
  end
end
```

Users of our library could write code like this:

exceptions/defexception.ex
```
try do
  talk_to_kinect
rescue
  error in [KinectProtocolError] ->
    IO.puts error.full_message
    if error.can_retry, do: schedule_retry
end
```

If an exception gets raised, the code handles it and possibly retries:

```
Kinect failed: usb unplugged, retriable: true
Retrying in 10 seconds
```

Now Ignore This Appendix

The Elixir source code for the mix utility contains no exception handlers. The Elixir compiler itself contains a total of five (but it is doing some pretty funky things).

If you find yourself defining new exceptions, ask if you should be isolating the code in a separate process instead. After all, if it can go wrong, wouldn't you want to isolate it?

Prepared exclusively for Richard Bishop

APPENDIX 2

Type Specifications and Type Checking

When we looked at 'defcallback' on page 271, we saw that we defined them in terms of the types of their parameters and return value. For example, we might write

```
defcallback parse(uri_info :: URI.Info.t) :: URI.Info.t
defcallback default_port() :: integer
```

The terms URI.Info.t and integer are examples of type specifications. In this appendix we'll see how to specify types in Elixir.[1] But, before we do, there's another question I should answer: why bother?

When Specifications are Used

Elixir type specifications come from Erlang. It is very common to see Erlang code where every exported (public) function is preceded by a -spec line. This is metadata that gives type information. The following code comes from the Elixir parser (which is [currently] written in Erlang). It says that the return_error function takes two parameters, an integer and any type, and never returns.

```
-spec return_error(integer(), any()) -> no_return().
return_error(Line, Message) ->
      throw({error, {Line, ?MODULE, Message}}).
```

One of the reasons the Erlang folks do this is to document their code. You can read it inline while reading the source, and you can also read it in the pages created by their documentation tool.

1. And, as José Valim pointed out in an email, the cool thing is that they are implemented (by Yurii Rashkovski) directly in the Elixir language itself—there is no special parsing involved. This is a fantastic illustration of the power of Elixir metaprogramming.

Prepared exclusively for Richard Bishop

The other reason is that they have tools such as *dialyzer* which perform static analysis of Erlang code and report on some kinds of type mismatches.[2]

These same benefits can aso apply to Elixir code. We have the `@spec` module attribute for documenting the type specification of a function and the `s` helper in iex for displaying specifications and the `t` helper for showing user-defined types. You can also run Erlang tools such as dialyzer on compiled Elixir beam files.

However, type specifications are not currently in wide use in the Elixir world. Whether you use them is probably a matter of personal taste.

Specifying a Type

A type is simply a subset of all possible values in a language. For example, the type `integer` means all the possible integer values, but excludes lists, binaries, pids, and so on.

The basic types in Elixir are `any`, `atom`, `char_list` (a single-quoted string), `float`, `fun`, `integer`, `none`, `pid`, `port`, `reference`, and `tuple`.

The type `any` (and it's alias, `_`) is the set of all values, and `none` is the empty set.

A literal atom or integer is the set containing just that value.

The value nil can be represented as `[]` or `nil`.

Collection types

A list is represented as [*type*], where type is any of the basic or combined types. This notation does not signify a list of one element—it simply says that elements of the list will be of the given type. If you want to specify a noneempty list, use [*type, ...*]. As a convenience, the type `list` is an alias for `[any]`.

Binaries are represented using the syntax:

`<< >>`
: an empty binary (size 0)

`<< _ : size >>`
: a sequence of *size* bits. This is called a *bitstring*.

2. http://www.erlang.org/doc/man/dialyzer.html

Prepared exclusively for Richard Bishop

*<< _ : size * unit_size >>*
: a sequence of *size* units, where each unit is *unit_size* bits long.

In the last two cases, *size* can be specified as _, in which case the binary has an arbitrary number of bits/units.

The predefined type bitstring is equivalent to <<_::_>>, an arbitrary sized sequence of bits. Similarly, binary is defined as <<_::_*8>>, an arbitrary sequence of 8-bit bytes.

Tuples are represented as { *type, type,...* }. The type tuple is equivalent, so both {atom,integer} and tuple(atom,integer} represent a tuple whose first element is an atom and whose second element is an integer.

Combining Types

The range operator .. can be used with literal integers to create a type representing that range. The three built-in types non_neg_integer, pos_integer, and neg_integer represent integers that are greater than or equal to, greater than, or less than zero.

The union operator, |, indicates that the acceptable values are the unions of its arguments.

Parentheses may be used to group terms in a type specification.

Types and Records

If R is a record or a protocol, then R.t represents values that are instances of that record. The type String.t represents binaries containing UTF8 characters.

You can specify the types of the fields in a record using the record_type function. This goes inside the record definition.

```
defrecord Sale, sku: "", quantity: 1, price: 0.0 do
  record_type sku: String.t, quantity: pos_integer, price: float
end
```

Anonymous Functions

Anonymous functions are specified using (*head* -> *return_type*).

The *head* specifies the arity and possibly the types of the function parameters. It can be ..., meaning an arbitrary number of arbitrarily typed arguments, or a list of types. In the latter case, the number of types in the list is the arity of the function.

```
(... -> integer)              # arbitrary parameters, returns an integer
(list(integer) -> integer)    # takes a list of integers, returns an integer
```

Prepared exclusively for Richard Bishop

```
(() -> String.t)              # takes no parameters, and returns an Elixir string
(integer, atom -> list(atom)) # takes an integer and an atom, and returns
                              # a list of atoms
```

You can put parentheses around the head if you find it clearer:

```
( atom, float -> list )
( (atom, float) -> list )
(list(integer) -> integer)
((list(integer)) -> integer)
```

Handling Truthy Values

The type as_boolean(T) says that the actual value matched will be of type T, but the function that uses the value will treat it as a *truthy* value (anything other than nil or false is considered true). Thus the specification for the Elixir function Enum.count is

```
@spec count(t, (element -> as_boolean(term))) :: non_neg_integer
```

Some Examples

integer | float

Any number (as we'll see, Elixir has an alias for this)

[{atom, any}]
list(atom, any)

A list of key/value pairs. The two forms are the same.

non_neg_integer | {:error, String.t}

Either an integer greater than or equal to zero, or a tuple containing the atom :error and a string.

(integer, atom -> { :pair, atom, integer })

An anonymous function that takes an integer and an atom, and returns a tuple containing the atom :pair, an atom, and an integer.

*<< _ :: _ * 4 >>*

A sequence of 4-bit nibbles

Defining New Types

The attribute @type can be used to define new types.

```
@type type_name :: type_specification
```

Elixir uses this to predefine some built-in types and aliases.

```
@type term      :: any
@type binary    :: <<_::_*8>>
```

Prepared exclusively for Richard Bishop

```
@type bitstring :: <<_::_*1>>
@type boolean   :: false | true
@type byte      :: 0..255
@type char      :: 0..0x10ffff
@type list      :: [ any ]
@type list(t)   :: [ t ]
@type number    :: integer | float
@type module    :: atom
@type mfa       :: {module, atom, byte}
@type node      :: atom
@type timeout   :: :infinity | non_neg_integer
@type no_return :: none
```

As the list entries show, you can parameterize the types in a new definition. Simply use one or more identifiers as parameters on the left hand size, and use these identifiers where you'd otherwise use type names on the left. Then, when you use the newly defined type, pass in actual types for each of these parameters.

```
@type variant(type_name, type) = { :variant, type_name, type)

@spec create_string_tuple(:string, String.t) :: variant(:string, String.t)
```

As well as @type, Elixir has the @typep and @opaque module attributes. They have the same syntax as @type, and do basically the same thing. The difference is in the visibility of the result.

@typep defines a type that is local to the module that contains it—the type is private. @opaque defines a type whose name may be known outside the module, but whose definition is not.

Specs for Functions and Callbacks

The @spec specifies the parameter count, types, and return value type of a function. It can appear anywhere in a module that defines the function, but by convention sits immediately before the function definition, following any function documentation.

We've already seen the syntax:

@spec *function_name*(*param1_type*, ...) :: *return_type*

Let's see some examples. These come from the built-in Dict module.

```
Line 1 @type key   :: any
   2 @type value :: any
   3 @type keys  :: [ key ]
   4 @type t     :: tuple | list    # `t` is the type of the collection
   5
```

Prepared exclusively for Richard Bishop

```
6 @spec values(t) :: [value]
7 @spec size(t) :: non_neg_integer
8 @spec has_key?(t, key) :: boolean
9 @spec update(t, key, value, (value -> value)) :: t
```

Line 6

values takes a collection (tuple or list) and returns a list of values (any).

Line 7

size takes a collection and returns an integer (>= 0)

Line 8

has_key? takes a collection and a key, and returns true or false.

Line 9

update takes a collection, a key, a value, and a function that maps a value to a value. It returns a (new) collection.

For functions with multiple heads (or those that have default values), you can specify multiple @spec attributes. Here's an example from the Enum module.

```
@spec at(t, index) :: element | nil
@spec at(t, index, default) :: element | default

def at(collection, n, default \\ nil) when n >= 0 do
  ...
end
```

The Enum module also has many examples of the use of as_boolean:

```
@spec filter(t, (element -> as_boolean(term))) :: list
def filter(collection, fun) when is_list(collection) do
  ...
end
```

This says that filter takes something enumerable and a function. That function maps an element to a term (which is an alias for any), and the filter function treats that value as being truthy. filter returns a list.

For more information on Elixir support for typespecs, have a look at the documentation for the Kernel.Typespec module.[3]

Using Dialyzer

Dialyzer analyses code that runs on the Erlang VM, looking for potential errors. To use it with Elixir, we have to compile our source into .beam files, and we have to make sure that the debug_info compiler option is set (which it

3. http://elixir-lang.org/docs/stable/elixir/Kernel.Typespec.html

Prepared exclusively for Richard Bishop

is when running mix in the default, development mode). Let's see how to do that by creating a trivial project with two source files. We'll also remove the supervisor that mix creates, because we don't want to drag OTP into this exercise.

```
$ mix new simple
...
$ cd simple
$ rm lib/simple/supervisor.ex
```

Inside the project, let's create a simple function. Being lazy, I haven't implemented the body yet.

```
defmodule Simple do
  @type atom_list :: list(atom)
  @spec count_atoms(atom_list)  :: non_neg_integer
  def count_atoms(list) do
    # ...
  end
end
```

Let's run dialyzer on our code. Because it works from .beam files, we have to remember to compile before we run dialyzer.

```
$ mix compile
.../simple/lib/simple.ex:4: variable list is unused
Compiled lib/simple.ex
Generated simple.app
$ dialyzer _build/dev/lib/simple/ebin
  Checking whether the PLT /Users/dave/.dialyzer_plt is up-to-date...
dialyzer: Could not find the PLT: /Users/dave/.dialyzer_plt
Use the options:
   --build_plt   to build a new PLT; or
   --add_to_plt  to add to an existing PLT

For example, use a command like the following:
   dialyzer --build_plt --apps erts kernel stdlib mnesia
Note that building a PLT such as the above may take 20 mins or so

If you later need information about other applications, say crypto,
you can extend the PLT by the command:
  dialyzer --add_to_plt --apps crypto
For applications that are not in Erlang/OTP use an absolute file name.
```

Oops. This looks serious, but it's not. Dialyzer needs the specifications for all the runtime libraries you're using. It stores them in a cache, which it calls a *persistent lookup table*, or *plt*. For now, we'll just initialize this with the basic runtime, erts and the basic Elixir runtime. You can always add more apps to it later.

Prepared exclusively for Richard Bishop

To do this, we first have to *find* your Elixir libraries. Fire up iex, and run:

```
iex> :code.lib_dir(:elixir)
/users/dave/Play/elixir/lib/elixir
```

The path on my system is a little unusual, as I build locally. But, take whatever path it shows you, and add /ebin to it—that's what we'll give to dialyser. (This will take several minutes.)

```
$ dialyzer --build_plt --apps erts /Users/dave/Play/elixir/lib/elixir/ebin
  Creating PLT /Users/dave/.dialyzer_plt ...
Unknown functions:
  auth:get_cookie/0
  auth:set_cookie/2
     :     :
```

You can safely ignore the warnings about unknown functions and types.

Now let's rerun the analysis of our project.

```
$ dialyzer _build/dev/lib/simple/ebin
  Checking whether the PLT /Users/dave/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
simple.ex:1: Invalid type specification for function
'Elixir.Simple':count_atoms/1. The success typing is (_) -> 'nil'
 done in 0m0.29s
done (warnings were emitted)
```

It's complaining that the typespec for count_atoms doesn't agree with the implementation. The *success typing* (think of this as the *actual type*) returns nil, but the spec says it is an integer. Dialyzer has caught our stubbed-out body. Let's fix that:

```
defmodule Simple do
  @type atom_list :: list(atom)
  @spec count_atoms(atom_list)  :: non_neg_integer
  def count_atoms(list) do
    length list
  end
end
```

Compile and dialyze:

```
$ mix compile
Compiled lib/simple.ex
Generated simple.app
$ dialyzer _build/dev/lib/simple/ebin
  Checking whether the PLT /Users/dave/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis... done in 0m0.29s
done (passed successfully)
```

Let's add a second module that calls our count_atoms function:

Prepared exclusively for Richard Bishop

typespecs/simple/lib/simple/client.ex
```
defmodule Client do
  @spec other_function() :: non_neg_integer
  def other_function do
    Simple.count_atoms [:a, :b, :c]
  end
end
```

Compile and dialyze:

```
$ mix compile
Compiled lib/client.ex
Compiled lib/simple.ex
Generated simple.app
$ dialyzer _build/dev/lib/simple/ebin
  Checking whether the PLT /Users/dave/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
client.ex:4: Function other_function/0 has no local return
client.ex:5: The call 'Elixir.Simple':count_atoms([1 | 2 | 3,...])
breaks the contract (atom_list()) -> non_neg_integer()
 done in 0m0.29s
```

That's pretty cool. Dialyzer noticed that we called `count_atoms` with a list of integers, and it is specced to receive a list of atoms. It also decided that this would raise an error, so the function would never return (that's the *no local return* warning). Let's fix that.

```
defmodule Client do
  @spec other_function() :: non_neg_integer
  def other_function do
    Simple.count_atoms [:a, :b, :c]
  end
end
```

```
$ mix compile
Compiled lib/client.ex
Compiled lib/simple.ex
Generated simple.app
$ dialyzer _build/dev/lib/simple/ebin
  Checking whether the PLT /Users/dave/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis... done in 0m0.27s
done (passed successfully)
```

And so it goes

Dialyzer and Type Inference

In this appendix, we've shown dialyzer working with type specs that we added to our functions. But it also does a credible job with unannotated code. This is because dialyzer knows the types of the built-in functions (remember when

Prepared exclusively for Richard Bishop

we ran it with --build_plt?) and can infer (some of) the types of your functions from this. Here's a simple example:

```
defmodule NoSpecs do
  def length_plus_n(list, n) do
    length(list) + n
  end
  def call_it do
    length_plus_n(2, 1)
  end
end
```

Compile this, and run dialyzer on the .beam file:

```
$ dialyzer _build/dev/lib/simple/ebin/Elixir.NoSpecs.beam
  Checking whether the PLT /Users/dave/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
no_specs.ex:7: Function call_it/0 has no local return
no_specs.ex:8: The call 'Elixir.NoSpecs':length_plus_n(2,1) will never
return since it differs in the 1st argument from the success typing
arguments: ([any()],number())
 done in 0m0.28s
done (warnings were emitted)
```

Here it noticed that the length_plus_n function called length on its first parameter, and length requires a list as an argument. This means that length_plus_n also needs a list argument, and so it complains.

What happens if we change the call to Simple.count_atoms [[:a, :b], :c]?

```
$ dialyzer _build/dev/lib/simple/ebin/Elixir.NoSpecs.beam
  Checking whether the PLT /Users/dave/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
no_specs.ex:7: Function call_it/0 has no local return
no_specs.ex:8: The call 'Elixir.NoSpecs':length_plus_n(['a', 'b'],'c')
will never return since it differs in the 2nd argument from the
success typing arguments: ([any()],number())
 done in 0m0.29s
done (warnings were emitted)
```

This is even cooler. It knows that + (which is implemented as a function) takes two numeric arguments. When we pass an atom as the second parameter, dialyzer recognizes that this makes no sense, and complains. But look at the error. It isn't complaining about the addition. Instead, it has assigned a default typespec to our function, based on its analysis of what we call inside that function.

Prepared exclusively for Richard Bishop

This is so-called *success typing*.[4] Dialyzer attempts to infer the most permissive types that are compatible with the code—it assumes the code is correct until it finds a contradiction. This makes it a powerful tool, as it can make assumptions as it runs.

Does that mean we don't need `@spec` attributes? That's your call. Try it with and without. Often, adding a `@spec` will further constrain the type signature of a function. We saw this with our `count_of_atoms` function, where the spec made it explicit that we expected a list of atoms as an argument.

Ultimately, dialyzer is a tool, not a test of your coding chops. Use it as such, but don't waste inordinate amounts of time adding specs if you're just doing it to get a gold star.

4. http://www.it.uu.se/research/group/hipe/papers/succ_types.pdf

Prepared exclusively for Richard Bishop

Bibliography

[Arm13] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, Second, 2013.

Prepared exclusively for Richard Bishop

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
http://pragprog.com/book/elixir
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *http://pragprog.com/book/elixir*

Contact Us

Online Orders:	*http://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://pragprog.com/write-for-us*
Or Call:	+1 800-699-7764

Prepared exclusively for Richard Bishop