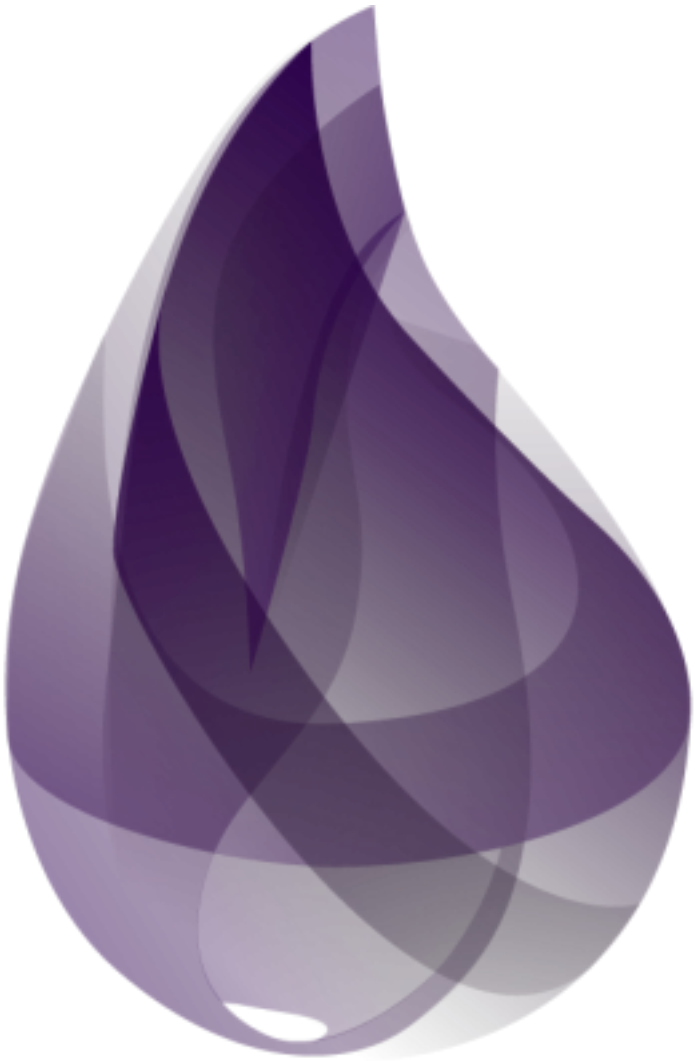

Elixir

Getting Started

64 contributors

Contents

Preface

Elixir Documentation from https://github.com/elixir-lang/elixir-lang.github.com

Chapter 1

Introduction

Welcome! In this tutorial we are going to show you how to get started with Elixir. We will start with how to install Elixir, how to use its interactive shell, basic data types and operators. In later chapters, we are going to discuss more advanced subjects such as macros, protocols and other features provided by Elixir.

To see Elixir in action, check out these introductory screencasts by Dave Thomas. The first one, Nine Minutes of Elixir, provides a brief tour of the language. The second one is a 30-minute introduction to Elixir that'll help you get started writing your first functions and creating your first processes in Elixir. Be sure to follow the next section of this guide to install Elixir on your machine and then follow along with the videos.

PeepCode also has a two hour video with José Valim called Meet Elixir.

Keep in mind that Elixir is still in development, so if at any point you receive an error message and you are not sure how to proceed, please let us know in the issues tracker. Having explanative and consistent error messages is one of the many features we aim for in Elixir.

1.1 Installation

The only prerequisite for Elixir is Erlang, version R16B or later, which can be easily installed with Precompiled packages. In case you want to install it

directly from source, it can be found on the Erlang website or by following the excellent tutorial available in the Riak documentation.

For Windows developers, we recommend the precompiled packages. Those on a UNIX platform can probably get Erlang installed via one of the many package management tools.

After Erlang is installed, you should be able to open up the command line (or command prompt) and check the Erlang version by typing **`erl`**. You will see some information as follows:

```
Erlang R16B (erts-5.10.1) [source] [64-bit] [smp:2:2] [rq:2]
```

Notice that depending on how you installed Erlang, it will not add Erlang binaries to your environment. Be sure to have Erlang binaries in your PATH, otherwise Elixir won't work!

After Erlang is up and running, it is time to install Elixir. You can do that via Distributions, Precompiled package or Compiling from Source.

1.1.1 Distributions

This tutorial requires Elixir v0.12.4 or later and it may be available in some distributions:

- Homebrew for Mac OS X
  - Update your homebrew to latest with **`brew update`**
  - Install Elixir: **`brew install elixir`**
- Fedora 17+ and Fedora Rawhide
  - **`sudo yum -y install elixir`**
- Arch Linux (on AUR)
  - **`yaourt -S elixir`**

- openSUSE (and SLES 11 SP3+)
  - Add Erlang devel repo with **`zypper ar -f obs://devel:languages:erlang`**
  - Install Elixir: **`zypper in elixir`**
- Gentoo
  - **`emerge -ask dev-lang/elixir`**
- Chocolatey for Windows
  - **`cinst elixir`**

If you don't use any of the distributions above, don't worry, we also provide a precompiled package!

1.1.2 Precompiled package

Elixir provides a precompiled package for every release. After downloading and unzip-ing the package, you are ready to run the **`elixir`** and **`iex`** commands from the **`bin`** directory. It is recommended that you also add Elixir's **`bin`** path to your PATH environment variable to ease development.

1.1.3 Compiling from source (Unix and MinGW)

You can download and compile Elixir in few steps. You can get the latest stable release here, unpack it and then run **`make`** inside the unpacked directory. After that, you are ready to run the **`elixir`** and **`iex`** commands from the **`bin`** directory. It is recommended that you add Elixir's **`bin`** path to your PATH environment variable to ease development:

```
$ export PATH="$PATH:/path/to/elixir/bin"
```

In case you are feeling a bit more adventurous, you can also compile from master:

```
$ git clone https://github.com/elixir-lang/elixir.git
$ cd elixir
$ make clean test
```

If the tests pass, you are ready to go. Otherwise, feel free to open an issue in the issues tracker on Github.

1.2 Interactive mode

When you install Elixir, you will have three new executables: **iex**, **elixir** and **elixirc**. If you compiled Elixir from source or are using a packaged version, you can find these inside the **bin** directory.

For now, let's start by running **iex** which stands for Interactive Elixir. In interactive mode, we can type any Elixir expression and get its result straight away. Let's warm up with some basic arithmetic expressions:

```
iex> 1 + 1
2
iex> 10 - 5
5
iex> 10 / 2
5.0
```

Notice **10 / 2** returned a float **5.0** instead of an integer. This is expected. In Elixir the operator **/** always returns a float. If you want to do integer division or get the division remainder, you can invoke the **div** and **rem** functions:

```
iex> div(10, 2)
5
iex> div 10, 2
5
iex> rem 10, 3
1
```

In the example above, we called two functions called `div` and `rem`. Notice that parentheses are not required in order to invoke a function. We are going to discuss more about this later. Let's move forward and see what other data types we have in Elixir.

1.3 Basic types

Some basic types are:

```
iex> 1          # integer
iex> 0x1F       # integer
iex> 1.0        # float
iex> :atom      # atom / symbol
iex> {1,2,3}    # tuple
iex> [1,2,3]    # list
iex> <<1,2,3>>  # bitstring
```

Elixir by default imports many functions to work on these types:

```
iex> size { 1, 2, 3 }
3

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

Elixir also supports UTF-8 encoded strings:

```
iex> "hellö"
"hellö"
```

Strings support interpolation, too:

```
iex> name = "world"
iex> "hello #{name}"
"hello world"
```

At the end of the day, strings are nothing more than a bunch of bytes packed together. We can verify this using the **`is_binary`** function:

```
iex> is_binary("hello")
true
iex> byte_size("hello")
5
```

And a binary is nothing more than a bunch of bits packed together:

```
iex> is_bitstring("hello")
true
iex> bit_size("hello")
40
```

Elixir allows you to create "raw" binaries too. Let's create a binary with three null bytes:

```
iex> is_binary <<0, 0, 0>>
true
```

Note that a single-quoted expression in Elixir is a char list and is not the same as a double-quoted one:

```
iex> is_binary('hello')
false
iex> is_list('hello')
true
```

We will go into more details about char lists in the next chapter. Elixir also provides **`true`** and **`false`** as booleans:

```
iex> true
true
iex> is_boolean false
true
```

Booleans are represented internally as atoms:

```
iex> is_atom(true)
true
```

Elixir also provides anonymous functions (note the dot between the variable and arguments when calling an anonymous function):

```
# function
iex> x = fn(a, b) -> a + b end
#Fun<erl_eval.12.111823515>
iex> x.(1, 2)
3
```

Elixir also provides **Port**, **Reference**s and **PID**s as data types (usually used in process communication), but they are out of the scope of a getting started tutorial. For now, let's take a look at the basic operators in Elixir before we move on to the next chapter.

1.4 Operators

As we saw earlier, Elixir provides **+**, **-**, *****, **/** as arithmetic operators.

Elixir also provides **++** and **--** to manipulate lists:

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

String concatenation is done via **<>**:

```
iex> "foo" <> "bar"
"foobar"
```

Elixir also provides three boolean operators: **or**, **and** and **not**. These operators are strict in the sense that they expect a boolean (**true** or **false**) as their first argument:

```
iex> true and true
true
iex> false or is_atom(:example)
true
```

Providing a non-boolean will raise an exception:

```
iex> 1 and true
** (ArgumentError) argument error
```

or and **and** are short-circuit operators. They only execute the right side if the left side is not enough to determine the result:

```
iex> false and error("This error will never be raised")
false

iex> true or error("This error will never be raised")
true
```

> Note: If you are an Erlang developer, **and** and **or** in Elixir actually map to the **andalso** and **orelse** operators in Erlang.

Besides these boolean operators, Elixir also provides **||**, **&&** and **!** which accept arguments of any type. For these operators, all values except **false** and **nil** will evaluate to true:

```
# or
iex> 1 || true
1
iex> false || 11
11

# and
```

```
iex> nil && 13
nil
iex> true && 17
17

# !
iex> !true
false
iex> !1
false
iex> !nil
true
```

As a rule of thumb, use **and**, **or** and **not** when you are expecting booleans. If any of the arguments are non-boolean, use **&&**, **||** and **!**.

Elixir also provides **==**, **!=**, **===**, **!==**, **<=**, **>=**, **<** and **>** as comparison operators:

```
iex> 1 == 1
true
iex> 1 != 2
true
iex> 1 < 2
true
```

The difference between **==** and **===** is that the latter is more strict when comparing integers and floats:

```
iex> 1 == 1.0
true
iex> 1 === 1.0
false
```

In Elixir, we can compare two different data types:

```
iex> 1 < :atom
true
```

The reason we can compare different data types is for pragmatism. Sorting algorithms don't need to worry about different data types in order to sort. The overall sorting order is defined below:

```
number < atom < reference < functions < port < pid < tuple <
```

You don't actually need to memorize this ordering, but it is important just to know an order exists.

Well, that is it for the introduction. In the next chapter, we are going to discuss some basic functions, data types conversions and a bit of control-flow.

Chapter 2

Diving in

In this chapter we'll go a bit deeper into the basic data-types, learn some control flow mechanisms and talk about anonymous functions.

2.1 Lists and tuples

Elixir provides both lists and tuples:

```
iex> is_list [1,2,3]
true
iex> is_tuple {1,2,3}
true
```

While both are used to store items, they differ on how those items are stored in memory. Lists are implemented as linked lists (where each item in the list points to the next item) while tuples are stored contiguously in memory.

This means that accessing a tuple element is very fast (constant time) and can be achieved using the **elem** function:

```
iex> elem { :a, :b, :c }, 0
:a
```

On the other hand, updating a tuple is expensive as it needs to duplicate the tuple contents in memory. Updating a tuple can be done with the **set_elem** function:

```
iex> set_elem { :a, :b, :c }, 0, :d
{:d,:b,:c}
```

> Note: If you are an Erlang developer, you will notice that we used the **elem** and **set_elem** functions instead of Erlang's **element** and **setelement**. The reason for this choice is that Elixir attempts to normalize Erlang API's to always receive the "subject" of the function as the first argument and employ zero-based access.

Since updating a tuple is expensive, when we want to add or remove elements, we use lists. Since lists are linked, it means accessing the first element of the list is very cheap. Accessing the n-th element, however, will require the algorithm to pass through n-1 nodes before reaching the n-th. We can access the **head** of the list as follows:

```
# Match the head and tail of the list
iex> [head | tail] = [1,2,3]
[1,2,3]
iex> head
1
iex> tail
[2,3]

# Put the head and the tail back together
iex> [head | tail]
[1,2,3]
iex> length [head | tail]
3
```

In the example above, we have matched the head of the list to the variable **head** and the tail of the list to the variable **tail**. This is called **pattern matching**. We can also pattern match tuples:

```
iex> { a, b, c } = { :hello, "world", 42 }
{ :hello, "world", 42 }
iex> a
:hello
iex> b
"world"
```

A pattern match will error in case the sides can't match. This is, for example, the case when the tuples have different sizes:

```
iex> { a, b, c } = { :hello, "world" }
** (MatchError) no match of right hand side value: {:hello,"world"}
```

And also when comparing different types:

```
iex> { a, b, c } = [:hello, "world", '!']
** (MatchError) no match of right hand side value: [:hello,"world",'!']
```

More interestingly, we can match on specific values. The example below asserts that the left side will only match the right side in case that the right side is a tuple that starts with the atom **`:ok`** as a key:

```
iex> { :ok, result } = { :ok, 13 }
{:ok,13}
iex> result
13

iex> { :ok, result } = { :error, :oops }
** (MatchError) no match of right hand side value: {:error,:oops}
```

Pattern matching allows developers to easily destruct data types such as tuples and lists. As we will see in following chapters, it is one of the foundations of recursion in Elixir. However, in case you can't wait to iterate through and manipulate your lists, the **`Enum`** module provides several helpers to manipulate lists (and other enumerables in general) while the **`List`** module provides several helpers specific to lists:

```
iex> Enum.at [1,2,3], 0
1
iex> List.flatten [1,[2],3]
[1,2,3]
```

2.2 Keyword lists

Elixir also provides a special syntax to create a list of keywords. They can be created as follows:

```
iex> [a: 1, b: 2]
[a: 1, b: 2]
```

Keyword lists are nothing more than a list of two element tuples where the first element of the tuple is an atom:

```
iex> [head | tail] = [a: 1, b: 2]
[a: 1, b: 2]
iex> head
{ :a, 1 }
```

The **Keyword** module contains several functions that allow you to manipulate a keyword list ignoring duplicated entries or not. For example:

```
iex> keywords = [foo: 1, bar: 2, foo: 3]
iex> Keyword.get keywords, :foo
1
iex> Keyword.get_values keywords, :foo
[1,3]
```

Since keyword lists are very frequently passed as arguments, they do not require brackets when given as the last argument in a function call. For instance, the examples below are valid and equivalent:

```
iex> if(2 + 2 == 4, [do: "OK"])
"OK"
iex> if(2 + 2 == 4, do: "OK")
"OK"
iex> if 2 + 2 == 4, do: "OK"
"OK"
```

2.3 Strings (binaries) and char lists (lists)

In Elixir, a double-quoted value is not the same as a single-quoted one:

```
iex> "hello" == 'hello'
false
iex> is_binary "hello"
true
iex> is_list 'hello'
true
```

We say a double-quoted value is a **string**, which is represented by a binary. A single-quoted value is a **char list**, which is represented by a list.

In fact, both double-quoted and single-quoted representations are just a shorter representation of binaries and lists respectively. Given that `?a` in Elixir returns the ASCII integer for the letter `a`, we could also write:

```
iex> is_integer ?a
true
iex> <<?a, ?b, ?c>>
"abc"
iex> [?a, ?b, ?c]
'abc'
```

In such cases, Elixir detects that all characters in the binary and in the list are printable and returns the quoted representation. However, adding a non-printable character forces them to be printed differently:

```
iex> <<?a, ?b, ?c, 1>>
<<97,98,99,1>>

iex> [?a, ?b, ?c, 1]
[97,98,99,1]
```

In Elixir, strings (double-quoted) are preferred unless you want to explicitly iterate over a char list (which is sometimes common when interfacing with Erlang code from Elixir). As lists, binaries also support pattern matching:

```
iex> <<a, b, c>> = "foo"
"foo"
iex> a
102
```

It is also possible to match on the "tail" of a binary, extracting the first bytes and matching the rest into a binary:

```
iex> <<f :: integer, rest :: binary>> = "foo"
"foo"
iex> f
102
iex> rest
"oo"
```

In the example above, we tagged each segment in the binary. The first segment has integer type (the default), which captures the ascii code for the letter "f". The remaining bytes "oo" are assigned into the binary **`rest`**.

The binary/bitstring syntax in Elixir is very powerful, allowing you to match on bytes, bits, utf8 codepoints and much more. You can find more about it in Elixir docs. Meanwhile, let's talk more about Unicode.

2.4 Unicode support

A string in Elixir is a binary which is encoded in UTF-8. For example, the string "é" is a UTF-8 binary containing two bytes:

```
iex> string = "é"
"é"
iex> size(string)
2
```

In order to easily manipulate strings, Elixir provides a **`String`** module:

```
# returns the number of bytes
iex> size "héllò"
7

# returns the number of characters as perceived by humans
iex> String.length "héllò"
5
```

> Note: to retrieve the number of elements in a data structure, you will use a function named **`length`** or **`size`**. Their usage is not arbitrary. The first is used when the number of elements needs to be calculated. For example, calling **`length(list)`** will iterate the whole list to find the number of elements in that list. **`size`** is the opposite, it means the value is pre-calculated and stored somewhere and therefore retrieving its value is a cheap operation. That said, we use **`size`** to get the size of a binary, which is cheap, but retrieving the number of unicode characters uses **`String.length`**, since the whole string needs to be iterated.

Each character in the string "héllò" above is an Unicode codepoint. One may use **`String.codepoints`** to split a string into smaller strings representing its codepoints:

```
iex> String.codepoints "héllò"
["h", "é", "l", "l", "ó"]
```

The Unicode standard assigns an integer value to each character. Elixir allows a developer to retrieve or insert a character based on its integer codepoint value as well:

```
# Getting the integer codepoint
iex> ?h
104
iex> ?é
233

# Inserting a codepoint based on its hexadecimal value
iex> "h\xE9ll\xF2"
"héllò"
```

UTF-8 also plays nicely with pattern matching. In the example below, we are extracting the first UTF-8 codepoint of a String and assigning the rest of the string to the variable **`rest`**:

```
iex> << eacute :: utf8, rest :: binary >> = "épa"
"épa"
iex> eacute
233
iex> << eacute :: utf8 >>
"é"
iex> rest
"pa"
```

In general, you will find working with binaries and strings in Elixir a breeze. Whenever you want to work with raw binaries, one can use Erlang's binary module, and use Elixir's String module when you want to work on strings, which are simply UTF-8 encoded binaries.

2.5 Blocks

One of the first control flow constructs we usually learn is the conditional **`if`**. In Elixir, we could write **`if`** as follow:

```
iex> if true, do: 1 + 2
3
```

The **`if`** expression can also be written using the block syntax:

```
iex> if true do
...>   a = 1 + 2
...>   a + 10
...> end
13
```

You can think of **do/end** blocks as a convenience for passing a group of expressions to **do:**. It is exactly the same as:

```
iex> if true, do: (
...>   a = 1 + 2
...>   a + 10
...> )
13
```

We can pass an **else** clause in the block syntax:

```
if false do
  :this
else
  :that
end
```

It is important to notice that **do/end** always binds to the farthest function call. For example, the following expression:

```
is_number if true do
  1 + 2
end
```

Would be parsed as:

```
is_number(if true) do
  1 + 2
end
```

Which is not what we want since **do** is binding to the farthest function call, in this case **is_number**. Adding explicit parenthesis is enough to resolve the ambiguity:

```
is_number(if true do
  1 + 2
end)
```

2.6 Control flow structures

In this section we'll describe Elixir's main control flow structures.

2.6.1 Revisiting pattern matching

At the beginning of this chapter we have seen some pattern matching examples:

```
iex> [h | t] = [1,2,3]
[1, 2, 3]
iex> h
1
iex> t
[2, 3]
```

In Elixir, `=` is not an assignment operator as in programming languages like Java, Ruby, Python, etc. `=` is actually a match operator which will check if the expressions on both left and right side match.

Many control-flow structures in Elixir rely extensively on pattern matching and the ability for different clauses to match. In some cases, you may want to match against the value of a variable, which can be achieved with the `^` operator:

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

In Elixir, it is a common practice to assign a value to underscore _ if we don't intend to use it. For example, if only the head of the list matters to us, we can assign the tail to underscore:

```
iex> [h | _] = [1,2,3]
[1, 2, 3]
iex> h
1
```

The variable _ in Elixir is special in that it can never be read from. Trying to read from it gives an unbound variable error:

```
iex> _
** (ErlangError) erlang error {:unbound_var, :_}
```

Although pattern matching allows us to build powerful constructs, its usage is limited. For instance, you cannot make function calls on the left side of a match. The following example is invalid:

```
iex> length([1,[2],3]) = 3
** (ErlangError) erlang error :illegal_pattern
```

2.6.2 Case

A `case` allows us to compare a value against many patterns until we find a matching one:

```
case { 1, 2, 3 } do
  { 4, 5, 6 } ->
    "This won't match"
  { 1, x, 3 } ->
    "This will match and assign x to 2"
  _ ->
    "This will match any value"
end
```

As in the **=** operator, any assigned variable will be overridden in the match clause. If you want to pattern match against a variable, you need to use the **^** operator:

```
x = 1
case 10 do
  ^x -> "Won't match"
  _  -> "Will match"
end
```

Each match clause also supports special conditions specified via guards:

```
case { 1, 2, 3 } do
  { 4, 5, 6 } ->
    "This won't match"
  { 1, x, 3 } when x > 0 ->
    "This will match and assign x"
  _ ->
    "No match"
end
```

In the example above, the second clause will only match when x is positive. The Erlang VM only allows a limited set of expressions as guards:

- comparison operators (**==**, **!=**, **===**, **!==**, **>**, **<**, **<=**, **>=**);
- boolean operators (**and**, **or**) and negation operators (**not**, **!**);
- arithmetic operators (**+**, **-**, *****, **/**);
- **<>** and **++** as long as the left side is a literal;
- the **in** operator;
- all the following type check functions (the number preceded by slash represents number of arguments):
  - is_atom/1
  - is_binary/1

- is_bitstring/1
- is_boolean/1
- is_float/1
- is_function/1
- is_function/2
- is_integer/1
- is_list/1
- is_number/1
- is_pid/1
- is_port/1
- is_record/1
- is_record/2
- is_reference/1
- is_tuple/1
- is_exception/1

plus these functions:

```
* 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()
* size(Tuple|Bitstring)
* tl(List)
* trunc(Number)
* tuple_size(Tuple)
```

Many independent guard clauses can also be given at the same time. For example, consider a function that checks if the first element of a tuple or a list is zero. It could be written as:

```
def first_is_zero?(tuple_or_list) when
  elem(tuple_or_list, 0) == 0 or hd(tuple_or_list) == 0 do
  true
end
```

However, the example above will always fail. If the argument is a list, calling **elem** on a list will raise an error. If the element is a tuple, calling **hd** on a tuple will also raise an error. To fix this, we can rewrite it to become two different clauses:

```
def first_is_zero?(tuple_or_list)
    when elem(tuple_or_list, 0) == 0
    when hd(tuple_or_list) == 0 do
  true
end
```

In such cases, if there is an error in one of the guards, it won't affect the next one.

2.6.3 Functions

In Elixir, anonymous functions can accept many clauses and guards, similar to the **case** mechanism we have just seen:

```
f = fn
  x, y when x > 0 -> x + y
  x, y -> x * y
end

f.(1, 3)  #=> 4
f.(-1, 3) #=> -3
```

As Elixir is an immutable language, the binding of the function is also immutable. This means that setting a variable inside the function does not affect its outer scope:

```
x = 1
(fn -> x = 2 end).()
x #=> 1
```

2.6.4 2.6.4 Receive

This next control-flow mechanism is essential to Elixir's actors. In Elixir, the code is run in separate processes that exchange messages between them. Those processes are not Operating System processes (they are actually quite light-weight) but are called so since they do not share state with each other.

In order to exchange messages, each process has a mailbox where the received messages are stored. The **`receive`** mechanism allows us to go through this mailbox searching for a message that matches the given pattern. Here is an example that uses the **`send/2`** function to send a message to the current process and then collects this message from its mailbox:

```
# Get the current process id
iex> current_pid = self

# Spawn another process that will send a message to current_pid
iex> spawn fn ->
  send current_pid, { :hello, self }
end
<0.36.0>
```

```
# Collect the message
iex> receive do
...>   { :hello, pid } ->
...>     IO.puts "Hello from #{inspect(pid)}"
...> end
Hello from <0.36.0>
```

You may not see exactly `<0.36.0>` back, but something similar. If there are no messages in the mailbox, the current process will hang until a matching message arrives unless an after clause is given:

```
iex> receive do
...>   :waiting ->
...>     IO.puts "This may never come"
...> after
...>   1000 -> # 1 second
...>     IO.puts "Too late"
...> end
Too late
```

Notice we spawned a new process using the `spawn` function passing another function as argument. We will talk more about those processes and even how to exchange messages in between different nodes in a later chapter.

2.6.5 Try

`try` in Elixir is used to catch values that are thrown. Let's start with an example:

```
iex> try do
...>   throw 13
...> catch
...>   number -> number
...> end
13
```

`try/catch` is a control-flow mechanism, useful in rare situations where your code has complex exit strategies and it is easier to `throw` a value back up in the stack. `try` also supports guards in `catch` and an `after` clause that is invoked regardless of whether or not the value was caught:

```
iex> try do
...>   throw 13
...> catch
...>   nan when not is_number(nan) -> nan
...> after
...>   IO.puts "Didn't catch"
...> end
Didn't catch
** throw 13
    erl_eval:expr/3
```

Notice that a thrown value which hasn't been caught halts the software. For this reason, Elixir considers such clauses unsafe (since they may or may not fail) and does not allow variables defined inside **`try/catch/after`** to be accessed from the outer scope:

```
iex> try do
...>   new_var = 1
...> catch
...>   value -> value
...> end
1
iex> new_var
** (UndefinedFunctionError) undefined function: IEx.Helpers.new_var/0
```

The common strategy is to explicitly return all arguments from **`try`**:

```
{ x, y } = try do
  x = calculate_some_value()
  y = some_other_value()
  { x, y }
catch
  _ -> { nil, nil }
end

x #=> returns the value of x or nil for failures
```

2.6.6 If and Unless

Besides the four main control-flow structures above, Elixir provides some extra control-flow structures to help on our daily work. For example, **`if`** and **`unless`**:

```
iex> if true do
iex>   "This works!"
iex> end
"This works!"

iex> unless true do
iex>   "This will never be seen"
iex> end
nil
```

Remember that **do/end** blocks in Elixir are simply a shortcut to the keyword notation. So one could also write:

```
iex> if true, do: "This works!"
"This works!"
```

Or even more complex examples like:

```
# This is equivalent...
if false, do: 1 + 2, else: 10 + 3

# ... to this
if false do
  1 + 2
else
  10 + 3
end
```

In Elixir, all values except **false** and **nil** evaluate to true. Therefore there is no need to explicitly convert the **if** argument to a boolean. If you want to check if one of many conditions are true, you can use the **cond** macro.

2.6.7 Cond

Whenever you want to check for many conditions at the same time, Elixir allows developers to use **cond** instead of nesting many **if** expressions:

```
cond do
  2 + 2 == 5 ->
    "This will never match"
  2 * 2 == 3 ->
    "Nor this"
  1 + 1 == 2 ->
    "But this will"
end
```

If none of the conditions return true, an error will be raised. For this reason, it is common to see a last condition equal to **`true`**, which will always match:

```
cond do
  2 + 2 == 5 ->
    "This will never match"
  2 * 2 == 3 ->
    "Nor this"
  true ->
    "This will always match (equivalent to else)"
end
```

2.7 Built-in functions

Elixir ships with many built-in functions automatically available in the current scope. In addition to the control flow expressions seen above, Elixir also adds: **`elem`** and **`set_elem`** to read and set values in tuples, **`inspect`** that returns the representation of a given data type as a binary, and many others. All of these functions imported by default are available in **`Kernel`** and Elixir special forms are available in **`Kernel.SpecialForms`**.

All of these functions and control flow expressions are essential for building Elixir programs. In some cases though, one may need to use functions available from Erlang, let's see how.

2.8 Calling Erlang functions

One of Elixir's assets is easy integration with the existing Erlang ecosystem. Erlang ships with a group of libraries called OTP (Open Telecom Platform).

Besides being a standard library, OTP provides several facilities to build OTP applications with supervisors that are robust, distributed and fault-tolerant.

Since an Erlang module is nothing more than an atom, invoking those libraries from Elixir is quite straight-forward. For example, we can call the function `flatten` from the module `lists` or interact with the `math` module as follows:

```
iex> :lists.flatten [1,[2],3]
[1,2,3]
iex> :math.sin :math.pi
1.2246467991473532e-16
```

Erlang's OTP is very well documented and we will learn more about building OTP applications in the Mix chapters:

- OTP docs
- Standard library docs

That's all for now. The next chapter will discuss how to organize our code into modules so it can be easily reused between different applications.

Chapter 3

Modules

In Elixir, you can group several functions into a module. In the previous chapter, for example, we invoked functions from the module **`List`**:

```
iex> List.flatten [1,[2],3]
[1, 2, 3]
```

In order to create our own modules in Elixir, all we have to do is to call the **`defmodule`** function and use **`def`** to define our functions:

```
iex> defmodule Math do
...>   def sum(a, b) do
...>     a + b
...>   end
...> end

iex> Math.sum(1, 2)
3
```

Before diving into modules, let's first have a brief overview about compilation.

3.1 Compilation

Most of the time it is convenient to write modules into files so they can be compiled and reused. Let's assume we have a file named **`math.ex`** with the

following contents:

```
defmodule Math do
  def sum(a, b) do
    a + b
  end
end
```

This file can be compiled using **elixirc** (remember, if you installed Elixir from a package or compiled it, **elixirc** will be inside the bin directory):

```
elixirc math.ex
```

This will generate a file named **Elixir.Math.beam** containing the bytecode for the defined module. Now, if we start **iex** again, our module definition will be available (considering **iex** is being started in the same directory the bytecode file is):

```
iex> Math.sum(1, 2)
3
```

Elixir projects are usually organized into three directories:

- ebin - contains the compiled bytecode
- lib - contains elixir code (usually **.ex** files)
- test - contains tests (usually **.exs** files)

Whenever interacting with an existing library, you may need to explicitly tell Elixir to look for bytecode in the **ebin** directory:

```
iex -pa ebin
```

Where **-pa** stands for **path append**. The same option can also be passed to **elixir** and **elixirc** executables. You can execute **elixir** and **elixirc** without arguments to get a list of options.

3.2 Scripted mode

In addition to the Elixir file `.ex`, Elixir also supports `.exs` files for scripting. Elixir treats both files exactly the same way, the only difference is in intention. `.ex` files are meant to be compiled while `.exs` files are used for scripting, without the need for compilation. For instance, one can create a file called `math.exs`:

```
defmodule Math do
  def sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)
```

And execute it as:

```
elixir math.exs
```

The file will be compiled in memory and executed, printing "3" as the result. No bytecode file will be created.

3.3 Functions and private functions

Inside a module, we can define functions with `def` and private functions with `defp`. A function defined with `def` is available to be invoked from other modules while a private function can only be invoked locally.

```
defmodule Math do
  def sum(a, b) do
    do_sum(a, b)
  end

  defp do_sum(a, b) do
    a + b
  end
end
```

```
Math.sum(1, 2)     #=> 3
Math.do_sum(1, 2) #=> ** (UndefinedFunctionError)
```

Function declarations also support guards and multiple clauses. If a function has several clauses, Elixir will try each clause until it finds one that matches. Here is the implementation of a function that checks if the given number is zero or not:

```
defmodule Math do
  def zero?(0) do
    true
  end

  def zero?(x) when is_number(x) do
    false
  end
end

Math.zero?(0)  #=> true
Math.zero?(1)  #=> false

Math.zero?([1,2,3])
#=> ** (FunctionClauseError)
```

Giving an argument that does not match any of the clauses raises an error.
Named functions also support default arguments:

```
defmodule Concat do
  def join(a, b, sep \\ " ") do
    a <> sep <> b
  end
end

IO.puts Concat.join("Hello", "world")      #=> Hello world
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
```

Any expression is allowed to serve as a default value, but it won't be evaluated during the function definition; it will simply be stored for later use. Every time the function is invoked and any of its default values have to be used, the expression for that default value will be evaluated:

```
defmodule DefaultTest do
  def dowork(x \\ IO.puts "hello") do
    x
  end
end
```

```
iex> DefaultTest.dowork 123
123
iex> DefaultTest.dowork
hello
:ok
```

If a function with default values has multiple clauses, it is recommended to create a separate clause without an actual body, just for declaring defaults:

```
defmodule Concat do
  def join(a, b \\ nil, sep \\ " ")

  def join(a, b, _sep) when nil?(b) do
    a
  end

  def join(a, b, sep) do
    a <> sep <> b
  end
end

IO.puts Concat.join("Hello", "world")      #=> Hello world
IO.puts Concat.join("Hello", "world", "_") #=> Hello_world
IO.puts Concat.join("Hello")               #=> Hello
```

When using default values, one must be careful to avoid overlapping function definitions. Consider the following example:

```
defmodule Concat do
  def join(a, b) do
    IO.puts "***First join"
    a <> b
  end

  def join(a, b, sep \\ " ") do
    IO.puts "***Second join"
```

```
    a <> sep <> b
  end
end
```

If we save the code above in a file named "concat.ex" and compile it, Elixir will emit the following warning:

```
concat.ex:7: this clause cannot match because a previous clau
```

The compiler is telling us that invoking the `join` function with two arguments will always choose the first definition of `join` whereas the second one will only be invoked when three arguments are passed:

```
$ iex concat.ex
```

```
iex> Concat.join "Hello", "world"
***First join
"Helloworld"
```

```
iex> Concat.join "Hello", "world", "_"
***Second join
"Hello_world"
```

3.4 Recursion

Due to immutability, loops in Elixir (and in functional programming languages) are written differently from conventional imperative languages. For example, in an imperative language, one would write:

```
for(i = 0; i < array.length; i++) {
  array[i] = array[i] * 2
}
```

In the example above, we are mutating the array which is not possible in Elixir. Therefore, functional languages rely on recursion: a function is called recursively until a condition is reached. Consider the example below that manually sums all the items in the list:

```
defmodule Math do
  def sum_list([h|t], acc) do
    sum_list(t, h + acc)
  end

  def sum_list([], acc) do
    acc
  end
end

Math.sum_list([1,2,3], 0) #=> 6
```

In the example above, we invoke `sum_list` giving a list `[1,2,3]` and the initial value `0` as arguments. When a function has many clauses, we will try each clause until we find one that matches according to the pattern matching rules. In this case, the list `[1,2,3]` matches against `[h|t]` which assigns `h = 1` and `t = [2,3]` while `acc` is set to 0.

Then, we add the head of the list to the accumulator `h + acc` and call `sum_list` again, recursively, passing the tail of the list as argument. The tail will once again match `[h|t]` until the list is empty, as seen below:

```
sum_list [1,2,3], 0
sum_list [2,3], 1
sum_list [3], 3
sum_list [], 6
```

When the list is empty, it will match the final clause which returns the final result of `6`. In imperative languages, such implementation would usually fail for large lists because the stack (in which our execution path is kept) would grow until it reaches a limit. Elixir, however, does tail call optimization in which the stack does not grow when a function exits by calling another function.

Recursion and tail call optimization are an important part of Elixir and are commonly used to create loops, especially in cases where a process needs to wait and respond to messages (using the **receive** macro we saw in the previous chapter). However, recursion as above is rarely used to manipulate lists, since the **Enum** module already abstracts such use cases. For instance, the example above could be simply written as:

```
Enum.reduce([1,2,3], 0, fn(x, acc) -> x + acc end)
```

3.5 Directives

In order to facilitate software reuse, Elixir supports three directives. As we are going to see below, they are called directives because they have **lexical scope**.

3.5.1 alias

alias allows you to setup aliases for any given module name. Imagine our **Math** module has a special list for doing math specific operations:

```
defmodule Math do
  alias Math.List, as: List
end
```

From now on, any reference to **List** will automatically expand to **Math.List**. In case one wants to access the original **List**, it can be done by accessing the module via **Elixir**:

```
List.values             #=> uses Math.List.values
Elixir.List.values      #=> uses List.values
Elixir.Math.List.values #=> uses Math.List.values
```

> Note: All modules defined in Elixir are defined inside a main Elixir namespace. However, for convenience, you can omit the Elixir main namespace.

Calling **alias** without an **as** option sets the alias automatically to the last part of the module name, for example:

```
alias Math.List
```

Is the same as:

```
alias Math.List, as: List
```

Notice that **alias** is **lexically scoped**, which allows you to set aliases inside specific functions:

```
defmodule Math do
  def add(a, b) do
    alias Math.List
    # ...
  end

  def minus(a, b) do
    # ...
  end
end
```

In the example above, since we are invoking **alias** inside the function **add**, the alias will just be valid inside the function **add**. **minus** won't be affected at all.

3.5.2 require

In general, a module does not need to be required before usage, except if we want to use the macros available in that module. For instance, suppose we created our own **my_if** implementation in a module named **MyMacros**. If we want to invoke it, we need to first explicitly require **MyMacros**:

```
defmodule Math do
  require MyMacros
  MyMacros.my_if do_something, it_works
end
```

An attempt to call a macro that was not loaded will raise an error. Note that like the **alias** directive, **require** is also lexically scoped. We will talk more about macros in chapter 5.

3.5.3 import

We use **import** whenever we want to easily access functions or macros from other modules without using the qualified name. For instance, if we want to use the **duplicate** function from **List** several times in a module and we don't want to always type **List.duplicate**, we can simply import it:

```
defmodule Math do
  import List, only: [duplicate: 2]

  def some_function do
    # call duplicate
  end
end
```

In this case, we are importing only the function **duplicate** (with arity 2) from **List**. Although **only:** is optional, its usage is recommended. **except** could also be given as an option.

import also supports selectors, to filter what you want to import. We have four selectors:

- **:default** - imports all functions and macros, except the ones starting by underscore;
- **:all** - imports all functions and macros;
- **:functions** - imports all functions;

- `:macros` - imports all macros;

For example, to import all macros, one could write:

```
import MyMacros, only: :macros
```

Or to import all functions, you could write:

```
import MyFunctions, only: :functions
```

Note that `import` is also **lexically scoped**, this means we can import specific macros inside specific functions:

```
defmodule Math do
  def some_function do
    import List, only: [duplicate: 2]
    # call duplicate
  end
end
```

In the example above, the imported `List.duplicate` is only visible within that specific function. `duplicate` won't be available in any other function in that module (or any other module for that matter).

Note that importing a module automatically requires it. Furthermore, `import` also accepts the `as:` option which is automatically passed to `alias` in order to create an alias.

3.6 Module attributes

Elixir brings the concept of module attributes from Erlang. For example:

```
defmodule MyServer do
  @vsn 2
end
```

In the example above, we are explicitly setting the version attribute for that module. `@vsn` is used by the code reloading mechanism in the Erlang VM to check if a module has been updated or not. If no version is specified, the version is set to the MD5 checksum of the module functions.

Elixir has a handful of reserved attributes. Here are just a few of them, the most commonly used ones. Take a look at the docs for Module for a complete list of supported attributes.

- `@moduledoc` - provides documentation for the current module;
- `@doc` - provides documentation for the function or macro that follows the attribute;
- `@behaviour` - (notice the British spelling) used for specifying an OTP or user-defined behaviour;
- `@before_compile` - provides a hook that will be invoked before the module is compiled. This makes it possible to inject functions inside the module exactly before compilation;

The following attributes are part of typespecs and are also supported by Elixir:

- `@spec` - provides a specification for a function;
- `@callback` - provides a specification for the behavior callback;
- `@type` - defines a type to be used in `@spec`;
- `@typep` - defines a private type to be used in `@spec`;
- `@opaque` - defines an opaque type to be used in `@spec`;

In addition to the built-in attributes outlined above, custom attributes may also be added:

```
defmodule MyServer do
  @my_data 13
  IO.inspect @my_data #=> 13
end
```

Unlike Erlang, user defined attributes are not stored in the module by default since it is common in Elixir to use such attributes to store temporary data. A developer can configure an attribute to behave closer to Erlang by calling **`Module.register_attribute/3`**.

Finally, notice that attributes can also be read inside functions:

```
defmodule MyServer do
  @my_data 11
  def first_data, do: @my_data
  @my_data 13
  def second_data, do: @my_data
end

MyServer.first_data #=> 11
MyServer.second_data #=> 13
```

Notice that reading an attribute inside a function takes a snapshot of its current value. In other words, the value is read at compilation time and not at runtime. Check the documentation for the module **`Module`** documentation for other functions to manipulate module attributes.

3.7 Nesting

Modules in Elixir can be nested too:

```
defmodule Foo do
  defmodule Bar do
  end
end
```

The example above will define two modules **`Foo`** and **`Foo.Bar`**. The second can be accessed as **`Bar`** inside **`Foo`** as long as they are in the same scope.

If later the developer decides to move **Bar** to another file, it will need to be referenced by its full name (**Foo.Bar**) or an alias needs to be set using the **alias** directive discussed above.

3.8 Aliases

In Erlang (and consequently in the Erlang VM), modules and functions are represented by atoms. For instance, this is valid Erlang code:

```
Mod = lists,
Mod:flatten([1,[2],3]).
```

In the example above, we store the atom **lists** in the variable **Mod** and then invoke the function **flatten** in it. In Elixir, the same idiom is allowed:

```
iex> mod = :lists
:lists
iex> mod.flatten([1,[2],3])
[1,2,3]
```

In other words, we are simply calling the function **flatten** on the atom **:lists**. This mechanism is exactly what empowers Elixir aliases. An alias in Elixir is a capitalized identifier (like **List**, **Keyword**, etc) which is converted to an atom representing a module during compilation. For instance, the **List** alias translates by default to the atom **Elixir.List**:

```
iex> is_atom(List)
true
iex> to_string(List)
"Elixir.List"
```

Chapter 4

Records, Protocols & Exceptions

Elixir provides both records and protocols. This chapter will outline the main features of both and provide some examples. More specifically, we will learn how to use **`defrecord`**, **`defprotocol`** and **`defimpl`**. At the end, we will briefly talk about exceptions in Elixir.

4.1 Records

Records are simple structures that hold values. For example, we can define a **`FileInfo`** record that is supposed to store information about files as follows:

```
defrecord FileInfo, atime: nil, accesses: 0
```

The line above will define a module named **`FileInfo`** which contains a function named **`new`** that returns a new record and other functions to read and set the values:

```
iex> file_info = FileInfo.new(atime: { 2010, 4, 17 }, accesses: 42)
iex> file_info.atime
{2010, 4, 17}
```

```
iex> file_info.atime({ 2012, 10, 13 }) #=> Returns a new FileInfo
FileInfo[atime: {2012, 10, 13}, accesses: 42]
iex> file_info.atime
{2010, 4, 17}
```

Notice that when we change the **`atime`** field of the record, we re-store the record in the **`file_info`** variable. This is because as almost everything else in Elixir, records are immutable. So changing the **`atime`** field does not update the record in place. Instead, it returns a new record with the new value set.

A record is simply a tuple where the first element is the record module name. We can get the record raw representation as follows:

```
inspect FileInfo.new, raw: true
#=> "{ FileInfo, nil, 0 }"
```

Besides defining readers and writers for each attribute, Elixir records also define an **`update_#{attribute}`** function to update values. Such functions expect a function as argument that receives the current value and must return the new one. For example, every time the file is accessed, the accesses counter can be incremented with:

```
file_info = FileInfo.new(accesses: 10)
file_info = file_info.update_accesses(fn(x) -> x + 1 end)
file_info.accesses #=> 11
```

4.1.1 Pattern matching

Elixir also allows one to pattern match against records. For example, imagine we want to check if a file was accessed or not based on the **`FileInfo`** record above, we could implement it as follows:

```
defmodule FileAccess do
  def was_accessed?(FileInfo[accesses: 0]), do: false
  def was_accessed?(FileInfo[]),            do: true
end
```

The first clause will only match if a `FileInfo` record is given and its `accesses` field is equal to zero. The second clause will match any `FileInfo` record and nothing more. We can also link the value of `accesses` to a variable as follows:

```
def was_accessed?(FileInfo[accesses: accesses]), do: accesses > 0
```

The pattern matching syntax can also be used to create new records:

```
file_info = FileInfo[accesses: 0]
```

Whenever using the bracket syntax above, Elixir expands the record to a tuple at compilation time. That said, the clause above:

```
def was_accessed?(FileInfo[accesses: 0]), do: false
```

Is effectively the same as:

```
def was_accessed?({ FileInfo, _, 0 }), do: false
```

Using the bracket syntax is a powerful mechanism not only due to pattern matching but also regarding performance, since it provides faster times compared to `FileInfo.new` and `file_info.accesses`. The downside is that we hardcode the record name. For this reason, Elixir allows you to mix and match both styles as you may find fit.

For more information on records, check out the documentation for the `defrecord` macro

4.2 Protocols

Protocols are a mechanism to achieve polymorphism in Elixir. Dispatching on a protocol is available to any data type as long as it implements the protocol. Lets consider a practical example.

In Elixir, only **`false`** and **`nil`** are treated as false. Everything else evaluates to true. Depending on the application, it may be important to specify a **`blank?`** protocol that returns a boolean for other data types that should be considered blank. For instance, an empty list or an empty binary could be considered blanks.

We could define this protocol as follows:

```
defprotocol Blank do
  @doc "Returns true if data is considered blank/empty"
  def blank?(data)
end
```

The protocol expects a function called **`blank?`** that receives one argument to be implemented. We can implement this protocol for some Elixir data types in the following way:

```
# Integers are never blank
defimpl Blank, for: Integer do
  def blank?(_), do: false
end

# Just empty list is blank
defimpl Blank, for: List do
  def blank?([]), do: true
  def blank?(_),  do: false
end

# Just the atoms false and nil are blank
defimpl Blank, for: Atom do
  def blank?(false), do: true
  def blank?(nil),   do: true
  def blank?(_),     do: false
end
```

And we would do so for all native data types. The types available are:

- **`Record`**
- **`Tuple`**
- **`Atom`**

- `List`
- `BitString`
- `Integer`
- `Float`
- `Function`
- `PID`
- `Port`
- `Reference`
- `Any`

Now, with the protocol defined and implementations in hand, we can invoke it:

```
Blank.blank?(0)        #=> false
Blank.blank?([])       #=> true
Blank.blank?([1,2,3]) #=> false
```

Notice however that passing a data type that does not implement the protocol raises an error:

```
iex> Blank.blank?("hello")
** (UndefinedFunctionError) undefined function: Blank.BitString.blank?/1
```

4.2.1 Fallback to any

In some cases, it may be convenient to provide a default implementation for all types. This can be achieved by setting `@fallback_to_any` to `true` in the protocol definition:

```
defprotocol Blank do
  @fallback_to_any true
  def blank?(data)
end
```

Which can now be implemented as:

```
defimpl Blank, for: Any do
  def blank?(_), do: false
end
```

Now all data types that we have not implemented the **Blank** protocol for will be considered non-blank.

4.2.2 Using protocols with records

The power of Elixir's extensibility comes when protocols and records are mixed.

For instance, Elixir provides a **HashDict** implementation that is an efficient data structure to store many keys. Let's take a look at how it works:

```
dict = HashDict.new
dict = HashDict.put(dict, :hello, "world")
HashDict.get(dict, :hello) #=> "world"
```

If we inspect our **HashDict**, we can see it is a simple tuple:

```
inspect(dict, raw: true)
#=> "{HashDict, 1, [[:hello | \"world\"]]}"
```

Since **HashDict** is a data structure that contains values, it would be convenient to implement the **Blank** protocol for it too:

```
defimpl Blank, for: HashDict do
  def blank?(dict), do: HashDict.size(dict) == 0
end
```

And now we can test it:

```
dict = HashDict.new(hello: "world")
Blank.blank?(dict)          #=> false
Blank.blank?(HashDict.new) #=> true
```

Excellent! The best of all is that we implemented the **Blank** protocol for an existing data structure (**HashDict**) without a need to wrap it or recompile it, which allows developers to easily extend previously defined protocols.

4.2.3 Built-in protocols

Elixir ships with some built-in protocols. Here are a couple of them:

- **Access** - specifies how to access an element. This is the protocol that empowers bracket access in Elixir. For example:

```
iex> x = [a: 1, b: 2]
[{:a, 1}, {:b, 2}]
iex> x[:a]
1
iex> x[:b]
2
```

- **Enumerable** - any data structures that can be enumerated must implement this protocol. This protocol is consumed by the **Enum** module which provides functions like **map**, **reduce** and others:

```
iex> Enum.map [1,2,3], fn(x) -> x * 2 end
[2,4,6]
iex> Enum.reduce 1..3, 0, fn(x, acc) -> x + acc end
6
```

- **Inspect** - this protocol is used to transform any data structure into a readable textual representation. This is what tools like IEx use to print results:

```
iex> { 1, 2, 3 }
{1,2,3}
iex> HashDict.new
#HashDict<[]>
```

Keep in mind that, by convention, whenever the inspected value starts with **#**, it is representing a data structure in non-valid Elixir syntax. For those, the true representation can be retrieved by calling **inspect** directly and passing **raw** as an option:

```
iex> inspect HashDict.new, raw: true
"{HashDict,0,[]}"
```

- **String.Chars** - specifies how to convert a data structure with characters to a string. It's exposed via the **to_string** function:

```
iex> to_string :hello
"hello"
```

Notice that string interpolation in Elixir calls the **to_string** function:

```
iex> "age: #{25}"
"age: 25"
```

The example above only works because numbers implement the **String.Chars** protocol. Passing a tuple, for example, will lead to an error:

```
iex> tuple = {1, 2, 3}
{1, 2, 3}
iex> "tuple: #{tuple}"
** (Protocol.UndefinedError) protocol String.Chars not implemented for {1, 2, 3}
```

When there is a need to "print" a more complex data structure, one can simply use the **inspect** function:

```
iex> "tuple: #{inspect tuple}"
"tuple: {1, 2, 3}"
```

Elixir defines other protocols which can be verified in Elixir's documentation. Frameworks and libraries that you use may define a couple of specific protocols as well. Use them wisely to write code that is easy to maintain and extend.

4.3 Exceptions

Exception handling would have its own chapter in many languages, but here they play a much lesser role.

An exception can be rescued inside a **`try`** block with the **`rescue`** keyword:

```
# rescue only runtime error
try do
  raise "some error"
rescue
  RuntimeError -> "rescued"
end

# rescue runtime and argument errors
try do
  raise "some error"
rescue
  [RuntimeError, ArgumentError] -> "rescued"
end

# rescue and assign to x
try do
  raise "some error"
rescue
  x in [RuntimeError] ->
    # all exceptions have a message
    x.message
end
```

Notice that **`rescue`** works with exception names and doesn't allow guards nor pattern matching. This limitation is intentional: developers should not use

exception values to drive their software. In fact, **exceptions in Elixir should only be used under exceptional circumstances**.

For example, software that does log partitioning and rotation over the network may face network issues or an eventual instability when accessing the file system. These scenarios are not exceptional in this particular software and must be handled accordingly. Therefore, the software can read some file using `File.read` and match the result:

```
case File.read(file) do
  { :ok, contents }  ->
    # We were able to access the file
    # Proceed as expected
  { :error, reason } ->
    # Oops, something went wrong
    # We need to handle the error accordingly
end
```

Notice that `File.read` does not raise an exception in case something goes wrong; it returns a tuple containing `{ :ok, contents }` in case of success and `{ :error, reason }` in case of failure. This allows us to use Elixir's pattern matching constructs to control how our application should behave.

On the other hand, a CLI interface that needs to access or manipulate a file given by the user may necessarily expect a file to be there. If the given file does not exist, there is nothing to do but fail. Then you may use `File.read!`, which raises an exception instead:

```
iex> contents = File.read!("/inexistent/file")
** (File.Error) could not read file /inexistent/file: no such file or directory
```

This pattern is common throughout the Elixir standard library, and many libraries provide both forms.

Finally, exceptions are simply records and can be defined with `defexception`, which has a similar API to `defrecord`. But remember, in Elixir you will use those sparingly.

This showcases Elixir's philosophy of not using exceptions for control-flow. If you feel like you need to rescue an exception in order to change how your code works, you should probably return an atom or tuple instead to allow pattern matching.

Next, let's take a look at how Elixir tackles productivity by building some macros using **defmacro** and **defmacrop**!

> Note: In order to ease integration with Erlang APIs, Elixir also supports "catching errors" coming from Erlang with **try/catch**, as it works in Erlang:
>
> ```
> try do
>   :erlang.error(:oops)
> catch
>   :error, :oops ->
>     "Got Erlang error"
> end
> ```
>
> The first atom can be one of **:error**, **:throw** or **:exit**. Keep in mind that catching errors is as discouraged as rescuing exceptions in Elixir.

Chapter 5

Macros

An Elixir program can be represented by its own data structures. This chapter will describe what those structures look like and how to manipulate them to create your own macros.

5.1 Building blocks of an Elixir program

The building block of Elixir is a tuple with three elements. The function call `sum(1,2,3)` is represented in Elixir as:

```
{ :sum, [], [1, 2, 3] }
```

You can get the representation of any expression by using the `quote` macro:

```
iex> quote do: sum(1, 2, 3)
{ :sum, [], [1, 2, 3] }
```

Operators are also represented as such tuples:

```
iex> quote do: 1 + 2
{:+, [context: Elixir, import: Kernel], [1, 2]}
```

Even a tuple is represented as a call to `{}`:

```
iex> quote do: { 1, 2, 3 }
{ :{}, [], [1, 2, 3] }
```

Variables are also represented using tuples, except the last element is an atom, instead of a list:

```
iex> quote do: x
{ :x, [], Elixir }
```

When quoting more complex expressions, we can see the representation is composed of such tuples, which are nested on each other resembling a tree where each tuple is a node:

```
iex> quote do: sum(1, 2 + 3, 4)
{:sum, [], [1, {:+, [context: Elixir, import: Kernel], [2, 3]}, 4]}
```

In general, each node (tuple) above follows the following format:

```
{ tuple | atom, list, list | atom }
```

- The first element of the tuple is an atom or another tuple in the same representation;
- The second element of the tuple is an list of metadata, it may hold information like the node line number;
- The third element of the tuple is either a list of arguments for the function call or an atom. When an atom, it means the tuple represents a variable.

Besides the node defined above, there are also five Elixir literals that when quoted return themselves (and not a tuple). They are:

```
:sum          #=> Atoms
1.0           #=> Numbers
[1,2]         #=> Lists
"binaries"    #=> Strings
{key, value} #=> Tuples with two elements
```

With those basic structures in mind, we are ready to define our own macro.

5.2 Defining our own macro

A macro can be defined using `defmacro`. For instance, in just a few lines of code we can define a macro called `unless` which does the opposite of `if`:

```
defmodule MyMacro do
  defmacro unless(clause, options) do
    quote do: if(!unquote(clause), unquote(options))
  end
end
```

Similarly to `if`, `unless` expects two arguments: a `clause` and `options`:

```
require MyMacro
MyMacro.unless var, do: IO.puts "false"
```

However, since `unless` is a macro, its arguments are not evaluated when it's invoked but are instead passed literally. For example, if one calls:

```
unless 2 + 2 == 5, do: call_function()
```

Our `unless` macro will receive the following:

```
unless({:==, [], [{:+, [], [2, 2]}, 5]}, { :call_function, [], [] })
```

Then our **unless** macro will call **quote** to return a tree representation of the **if** clause. This means we are transforming our **unless** into an **if**!

There is a common mistake when quoting expressions which is that developers usually forget to **unquote** the proper expression. In order to understand what **unquote** does, let's simply remove it:

```
defmacro unless(clause, options) do
  quote do: if(!clause, options)
end
```

When called as **unless 2 + 2 == 5, do: call_function()**, our **unless** would then literally return:

```
if(!clause, options)
```

Which would fail because the clause and options variables are not defined in the current scope. If we add **unquote** back:

```
defmacro unless(clause, options) do
  quote do: if(!unquote(clause), unquote(options))
end
```

unless will then return:

```
if(!(2 + 2 == 5), do: call_function())
```

In other words, **unquote** is a mechanism to inject expressions into the tree being quoted and it is an essential tool for meta-programming. Elixir also provides **unquote_splicing** allowing us to inject many expressions at once.

We can define any macro we want, including ones that override the built-in macros provided by Elixir. For instance, you can redefine **case**, **receive**, **+**, etc. The only exceptions are Elixir special forms that cannot be overridden, the full list of special forms is available in **Kernel.SpecialForms**.

5.3 Macros hygiene

Elixir macros have late resolution. This guarantees that a variable defined inside a quote won't conflict with a variable defined in the context where that macro is expanded. For example:

```
defmodule Hygiene do
  defmacro no_interference do
    quote do: a = 1
  end
end

defmodule HygieneTest do
  def go do
    require Hygiene
    a = 13
    Hygiene.no_interference
    a
  end
end

HygieneTest.go
# => 13
```

In the example above, even if the macro injects `a = 1`, it does not affect the variable `a` defined by the `go` function. In case the macro wants to explicitly affect the context, it can use `var!`:

```
defmodule Hygiene do
  defmacro interference do
    quote do: var!(a) = 1
  end
end

defmodule HygieneTest do
  def go do
    require Hygiene
    a = 13
    Hygiene.interference
    a
  end
end

HygieneTest.go
# => 1
```

Variables hygiene only works because Elixir annotates variables with their context. For example, a variable **x** defined at the line 3 of a module, would be represented as:

```
{ :x, [line: 3], nil }
```

However, a quoted variable is represented as:

```
defmodule Sample do
  def quoted do
    quote do: x
  end
end

Sample.quoted #=> { :x, [line: 3], Sample }
```

Notice that the third element in the quoted variable is the atom **Sample**, instead of **nil**, which marks the variable as coming from the **Sample** module. Therefore, Elixir considers those two variables come from different contexts and handle them accordingly.

Elixir provides similar mechanisms for imports and aliases too. This guarantees macros will behave as specified by its source module rather than conflicting with the target module.

5.4 Private macros

Elixir also supports private macros via **defmacrop**. As private functions, these macros are only available inside the module that defines them, and only at compilation time. A common use case for private macros is to define guards that are frequently used in the same module:

```
defmodule MyMacros do
  defmacrop is_even?(x) do
    quote do
      rem(unquote(x), 2) == 0
    end
```

```
  end

  def add_even(a, b) when is_even?(a) and is_even?(b) do
    a + b
  end
end
```

It is important that the macro is defined before its usage. Failing to define a macro before its invocation will raise an error at runtime, since the macro won't be expanded and will be translated to a function call:

```
defmodule MyMacros do
  def four, do: two + two
  defmacrop two, do: 2
end

MyMacros.four #=> ** (UndefinedFunctionError) undefined function: two/0
```

5.5 Code execution

To finish our discussion about macros, we are going to briefly discuss how code execution works in Elixir. Code execution in Elixir is done in two steps:

1) All the macros in the code are expanded recursively;

2) The expanded code is compiled to Erlang bytecode and executed

This behavior is important to understand because it affects how we think about our code structure. Consider the following code:

```
defmodule Sample do
  case System.get_env("FULL") do
    "true" ->
      def full?(), do: true
    _ ->
      def full?(), do: false
  end
end
```

The code above will define a function **`full?`** which will return true or false depending on the value of the environment variable **`FULL`** at **compilation time**.

In order to execute this code, Elixir will first expand all macros. Considering that **`defmodule`** and **`def`** are macros, the code will expand to something like:

```
:elixir_module.store Sample, fn ->
  case System.get_env("FULL") do
    "true" ->
      :elixir_def.store(Foo, :def, :full?, [], true)
    _ ->
      :elixir_def.store(Foo, :def, :full?, [], false)
end
```

This code will then be executed, define a module **`Foo`** and store the appropriate function based on the value of the environment variable **`FULL`**. We achieve this by using the modules **`:elixir_module`** and **`:elixir_def`**, which are Elixir internal modules written in Erlang.

There are two lessons to take away from this example:

1) a macro is always expanded, regardless if it is inside a **`case`** branch that won't actually match when executed;

2) we cannot invoke a function or macro just after it is defined in a module. For example, consider:

```
defmodule Sample do
  def full?, do: true
  IO.puts full?
end
```

The example above will fail because it translates to:

```
:elixir_module.store Sample, fn ->
  :elixir_def.store(Foo, :def, :full?, [], true)
  IO.puts full?
end
```

At the moment the module is being defined, there isn't **yet** a function named **`full?`** defined in the module, so **`IO.puts full?`** will cause the compilation to fail.

5.6 Don't write macros

Although macros are a powerful construct, the first rule of the macro club is **don't write macros**. Macros are harder to write than ordinary Elixir functions, and it's considered to be bad style to use them when they're not necessary. Elixir already provides elegant mechanisms to write your every day code and macros should be saved as last resort.

With those lessons, we finish our introduction to macros. Next, let's move to the next chapter which will discuss several topics such as documentation, partial application and others.

Chapter 6

Other topics

This chapter contains different small topics that are part of Elixir's day to day work. We will learn about writing documentation, list and binary comprehensions, partial function application and more!

6.1 String sigils

Elixir provides string sigils via the token ~:

```
~s(String with escape codes \x26 interpolation)
~S(String without escape codes and without #{interpolation})
```

Sigils starting with an uppercase letter never escape characters or do interpolation. Notice the separators are not necessarily parenthesis, but any non-alphanumeric character:

```
~s-another string-
```

Internally, ~**s** is translated as a call to **sigil_s**. For instance, the docs for ~**s** are available in the macro **sigil_s/2** defined in the **Kernel** module.

The sigils defined in Elixir by default are:

- ~c and ~C - Returns a char list;
- ~r and ~R - Returns a regular expression;
- ~s and ~S - Returns a string;
- ~w and ~W - Returns a list of "words" split by whitespace;

6.2 Heredocs

Elixir supports heredocs as a way to define long strings. Heredocs are delimited by triple double-quotes for string or triple single-quotes for char lists:

```
"""
String heredoc
"""

'''
Charlist heredoc
'''
```

The heredoc ending must be in a line on its own, otherwise it is part of the heredoc:

```
"""
String heredocs in Elixir use """
"""
```

Notice the sigils discussed in the previous section are also available as heredocs:

```
~s"""
A heredoc without escaping or interpolation
"""
```

6.3 Documentation

Elixir uses module attributes described in chapter 3 to drive its documentation system. For instance, consider the following example:

```
defmodule MyModule do
  @moduledoc "it does x"

  @doc "returns the version"
  def version, do: 1
end
```

In the example above, we are adding a module documentation to **MyModule** via **@moduledoc** and using **@doc** to document each function. When compiled, we are able to inspect the documentation attributes at runtime (remember to start iex in the same directory in which you compiled the module):

```
$ elixirc my_module.ex
$ iex
```

```
iex> MyModule.__info__(:docs)
[{ { :version, 0 }, 5, :def, [], "Returns the version" }]
iex> MyModule.__info__(:moduledoc)
{1,"It does X"}
```

__info__(:docs) returns a list of tuples where each tuple contains a function/arity pair, the line the function was defined on, the kind of the function (either **def** or **defmacro**, private functions defined with **defp** or **defmacrop** cannot be documented), the function arguments and its documentation. The comment will be either a binary or **nil** (not given) or **false** (explicit no doc).

Similarly, **__info__(:moduledoc)** returns a tuple with the line the module was defined on and its documentation.

Elixir promotes the use of markdown with heredocs to write readable documentation:

```
defmodule Math do
  @moduledoc """
  This module provides mathematical functions
  as sin, cos and constants like pi.

  ## Examples

      Math.pi
      #=> 3.1415...

  """
end
```

6.4 IEx Helpers

Elixir's interactive console (IEx) ships with many functions to make the developer's life easier. One of these functions is called **h**, which shows documentation directly at the command line:

```
iex> h()
# IEx.Helpers
...
:ok
```

As you can see, invoking **h()** prints the documentation of **IEx.Helpers**. From there, we can navigate to any of the other helpers by giving its name and arity to get more information:

```
iex> h(c/2)
* def c(files, path \\ ".")
...
:ok
```

This functionality can also be used to print the documentation for any Elixir module in the system:

```
iex> h(Enum)
...
iex> h(Enum.each/2)
...
```

The documentation for built-in functions can also be accessed directly or indirectly from the **Kernel** module:

```
iex> h(is_atom/1)
...
iex> h(Kernel.is_atom/1)
...
```

6.5 Function capture

Functions in Elixir are identified by their name and arity (the number of arguments it expects). Since passing functions around is a common practice in Elixir, Elixir provides the capture operator **&** as a convenience. Consider this example:

```
iex> list = ["foo", "bar", "baz"]
["foo","bar","baz"]
iex> Enum.map list, fn(x) -> size(x) end
[3,3,3]
```

We could write the second line using the capture operator as:

```
iex> Enum.map list, &size/1
[3,3,3]
```

The capture operator retrieves the function size with arity 1 as an anonymous function and passes it down to **Enum.map/2**. Remote calls can also be captured:

```
iex> fun = &Kernel.size/1
&Kernel.size/1
iex> fun.("hello")
5
```

A capture also allows the captured functions to be partially applied, for example:

```
iex> fun = &rem(&1, 2)
#Function<6.80484245 in :erl_eval.expr/5>
iex> fun.(4)
0
```

In the example above, we use `&1` as a placeholder, generating a function with one argument. The capture above is equivalent to `fn(x) -> rem(x, 2) end`.

Since operators are treated as regular function calls in Elixir, they are also supported, although they require explicit parentheses:

```
iex> fun = &(&1 + &2)
iex> fun.(1, 2)
3
```

The capture operator is a handy way of capturing functions and passing them around as anonymous functions. To learn more, read the documentation.

6.6 Use

`use` is a macro that provides a common API for extension. For instance, in order to use the `ExUnit` test framework that ships with Elixir, you simply need to use `ExUnit.Case` in your module:

```
defmodule AssertionTest do
  use ExUnit.Case, async: true

  test "always pass" do
    true = true
  end
end
```

This allows **ExUnit.Case** to configure and set up the module for testing, for example, by making the **test** macro used above available.

The implementation of the **use** macro is actually quite trivial. When you invoke **use** with a module, it invokes a hook called **__using__** in this module. For example, the **use** call above is simply a translation to:

```
defmodule AssertionTest do
  require ExUnit.Case
  ExUnit.Case.__using__(async: true)

  test "always pass" do
    true = true
  end
end
```

In general, we recommend APIs to provide a **__using__** hook in case they want to expose functionality to developers.

6.7 Comprehensions

Elixir also provides list and bit comprehensions. List comprehensions allow you to quickly build a list from another list:

```
iex> lc n inlist [1,2,3,4], do: n * 2
[2,4,6,8]
```

Or, using keywords blocks:

```
lc n inlist [1,2,3,4] do
  n * 2
end
```

A comprehension accepts many generators (given by **inlist** or **inbits** operators) as well as filters:

```
# A comprehension with a generator and a filter
iex> lc n inlist [1,2,3,4,5,6], rem(n, 2) == 0, do: n
[2,4,6]

# A comprehension with two generators
iex> lc x inlist [1,2], y inlist [2,3], do: x*y
[2,3,4,6]
```

Elixir provides generators for both lists and bitstrings:

```
# A list generator:
iex> lc n inlist [1,2,3,4], do: n * 2
[2,4,6,8]

# A bit string generator:
iex> lc <<n>> inbits <<1,2,3,4>>, do: n * 2
[2,4,6,8]
```

Bit string generators are quite useful when you need to organize streams:

```
iex> pixels = <<213,45,132,64,76,32,76,0,0,234,32,15>>
iex> lc <<r :: size(8), g :: size(8), b :: size(8)>> inbits pixels, do: {r,g,b}
[{213,45,132},{64,76,32},{76,0,0},{234,32,15}]
```

Remember, as strings are binaries and a binary is a bitstring, we can also use strings in comprehensions. For instance, the example below removes all white space characters from a string via bit comprehensions:

```
iex> bc <<c>> inbits " hello world ", c != ?\s, do: <<c>>
"helloworld"
```

6.8 Pseudo variables

Elixir provides a set of pseudo-variables. These variables can only be read and never assigned to. They are:

- **__MODULE__** - Returns an atom representing the current module or nil;
- **__DIR__** - Returns a string representing the current directory;
- **__ENV__** - Returns a Macro.Env record with information about the compilation environment. Here we can access the current module, function, line, file and others;
- **__CALLER__** - Also returns a Macro.Env record but with information of the calling site. **__CALLER__** is available only inside macros;

Chapter 7

Where To Go Next

7.1 Applications

In order to get your first project started, Elixir ships with a build tool called **Mix**. You can get your new project started by simply running:

```
mix new path/to/new/project
```

You can learn more about Elixir and other applications in the links below:

- Mix - a build tool for Elixir
- ExUnit - a unit test framework

7.2 A Byte of Erlang

As the main page of this site puts it:

> Elixir is a programming language built on top of the Erlang VM.

Sooner than later, an Elixir developer will want to interface with existing Erlang libraries. Here's a list of online resources that cover Erlang's fundamentals and its more advanced features:

- This Erlang Syntax: A Crash Course provides a concise intro to Erlang's syntax. Each code snippet is accompanied by equivalent code in Elixir. This is an opportunity for you to not only get some exposure to the Erlang's syntax but also review some of the things you have learned in the present guide.
- Erlang's official website has a short tutorial with pictures that briefly describe Erlang's primitives for concurrent programming.
- Learn You Some Erlang for Great Good! is an excellent introduction to Erlang, its design principles, standard library, best practices and much more. If you are serious about Elixir, you'll want to get a solid understanding of Erlang principles. Once you have read through the crash course mentioned above, you'll be able to safely skip the first couple of chapters in the book that mostly deal with the syntax. When you reach The Hitchhiker's Guide to Concurrency chapter, that's where the real fun starts.

7.3 Reference Manual

You can also check the source code of Elixir itself, which is mainly written in Elixir (mainly the `lib` directory), or explore Elixir's documentation.

7.4 Join The Community

Remember that in case of any difficulties, you can always visit the **#elixir-lang** channel on **irc.freenode.net** or send a message to the mailing list. You can be sure that there will be someone willing to help. And to keep posted on the latest news and announcements, follow the blog and join elixir-core mailing list.

Chapter 8

Introduction to ExUnit

ExUnit is a unit test framework that ships with Elixir.

Using ExUnit is quite easy, here is a file with the minimum required:

```
ExUnit.start

defmodule MyTest do
  use ExUnit.Case

  test "the truth" do
    assert true
  end
end
```

In general, we just need to invoke **`ExUnit.start`**, define a test case using **`ExUnit.Case`** and our batch of tests. Assuming we saved this file as **`assertion_test.exs`**, we can run it directly:

```
bin/elixir assertion_test.exs
```

In this chapter, we will discuss the most common features available in ExUnit and how to customize it further.

8.1 Starting ExUnit

ExUnit is usually started via **`ExUnit.start`**. This function accepts a couple options, so check its documentation for more details. For now, we will just

detail the most common ones:

- `:formatter` - When you run tests with ExUnit, all the IO is done by the formatter. Developers can define their own formatters and this is the configuration that tells ExUnit to use a custom formatter;
- `:max_cases` - As we are going to see soon, ExUnit allows you to easily run tests concurrently. This is very useful to speed up your tests that have no side affects. This option allows us to configure the maximum number of cases ExUnit runs concurrently.

8.2 Defining a test case

After ExUnit is started, we can define our own test cases. This is done by using `ExUnit.Case` in our module:

```
use ExUnit.Case
```

`ExUnit.Case` provides some features, so let's take a look at them.

8.2.1 The test macro

`ExUnit.Case` runs all functions whose name start with `test` and expects one argument:

```
def test_the_truth(_) do
  assert true
end
```

As a convenience to define such functions, `ExUnit.Case` provides a `test` macro, which allows one to write:

```
test "the truth" do
  assert true
end
```

This construct is considered more readable. The `test` macro accepts either a binary or an atom as name.

8.2.2 Assertions

Another convenience provided by `ExUnit.Case` is to automatically import a set of assertion macros and functions, available in `ExUnit.Assertions`.

In the majority of tests, the only assertion macros you will need to use are `assert` and `refute`:

```
assert 1 + 1 == 2
refute 1 + 3 == 3
```

ExUnit automatically breaks those expressions apart and attempt to provide detailed information in case the assertion fails. For example, the failing assertion:

```
assert 1 + 1 == 3
```

Will fail as:

```
Expected 2 to be equal to (==) 3
```

However, some extra assertions are convenient to make testing easier for some specific cases. A good example is the `assert_raise` macro:

```
assert_raise ArithmeticError, "bad argument in arithmetic expression", fn ->
  1 + "test"
end
```

So don't forget to check `ExUnit.Assertions`' documentation for more examples.

8.2.3 Callbacks

`ExUnit.Case` defines four callbacks: `setup`, `teardown`, `setup_all` and `teardown_all`:

```
defmodule CallbacksTest do
  use ExUnit.Case, async: true

  setup do
    IO.puts "This is a setup callback"
    :ok
  end

  test "the truth" do
    assert true
  end
end
```

In the example above, the `setup` callback will be run before each test. In case a `setup_all` callback is defined, it would run once before all tests in that module.

A callback **must** return `:ok` or `{ :ok, data }`. When the latter is returned, the `data` argument must be a keywords list containing metadata about the test. This metadata can be accessed in any other callback or in the test itself:

```
defmodule CallbacksTest do
  use ExUnit.Case, async: true

  setup do
    IO.puts "This is a setup callback"
    { :ok, from_setup: :hello }
  end

  test "the truth", meta do
    assert meta[:from_setup] == :hello
  end

  teardown meta do
    assert meta[:from_setup] == :hello
    :ok
  end
end
```

Metadata is used when state need to be explicitly passed to tests.

8.2.4 Async

Finally, ExUnit also allows test cases to run concurrently. All you need to do is pass the `:async` option set to true:

```
use ExUnit.Case, async: true
```

This will run this test case concurrently with other test cases which are async too. The tests inside a particular case are still run sequentially.

8.3 Lots To Do

ExUnit is still a work in progress. Feel free to visit our issues tracker to add issues for anything you'd like to see in ExUnit and feel free to contribute.

Chapter 9

Introduction to Mix

Elixir ships with a few applications to make building and deploying projects with Elixir easier and Mix is certainly their backbone.

Mix is a build tool that provides tasks for creating, compiling, testing (and soon releasing) Elixir projects. Mix is inspired by the Leiningen build tool for Clojure and was written by one of its contributors.

In this chapter, you will learn how to create projects using **`mix`** and install dependencies. In the following sections, we will also learn how to create OTP applications and create custom tasks with Mix.

9.1 Bootstrapping

In order to start your first project, simply use the **`mix new`** command passing the path to your project. For now, we will create an project called **`my_project`** in the current directory:

```
$ mix new my_project --bare
```

Mix will create a directory named **`my_project`** with few files in it:

```
.gitignore
README.md
mix.exs
```

```
lib/my_project.ex
test/test_helper.exs
test/my_project_test.exs
```

Let's take a brief look at some of these.

> Note: Mix is an Elixir executable. This means that in order to run **mix**, you need to have elixir's executable in your PATH. If not, you can run it by passing the script as argument to elixir:
>
> ```
> $ bin/elixir bin/mix new ./my_project
> ```
>
> Note that you can also execute any script in your PATH from Elixir via the -S option:
>
> ```
> $ bin/elixir -S mix new ./my_project
> ```
>
> When using -S, elixir finds the script wherever it is in your PATH and executes it.

9.1.1 mix.exs

This is the file with your projects configuration. It looks like this:

```
defmodule MyProject.Mixfile do
  use Mix.Project

  def project do
    [ app: :my_project,
      version: "0.0.1",
      deps: deps ]
  end

  # Configuration for the OTP application
  def application do
    []
  end

  # 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" }
  defp deps do
    []
  end
end
```

Our **mix.exs** defines two functions: **project**, which returns project configuration like the project name and version, and **application**, which is used to generate an Erlang application that is managed by the Erlang Runtime. In this chapter, we will talk about the **project** function. We will go into detail about what goes in the **application** function in the next chapter.

9.1.2 lib/my_project.ex

This file contains a simple module definition to lay out our code:

```
defmodule MyProject do
end
```

9.1.3 test/my_project_test.exs

This file contains a stub test case for our project:

```
defmodule MyProjectTest do
  use ExUnit.Case

  test "the truth" do
    assert true
  end
end
```

It is important to note a couple things:

1) Notice the file is an Elixir script file (**.exs**). This is convenient because we don't need to compile test files before running them;

2) We define a test module named **MyProjectTest**, using **ExUnit.Case** to inject default behavior and define a simple test. You can learn more about the test framework in the ExUnit chapter;

9.1.4 test/test_helper.exs

The last file we are going to check is the **test_helper.exs**, which simply sets up the test framework:

```
ExUnit.start
```

This file will be automatically required by Mix every time before we run our tests. And that is it, our project is created. We are ready to move on!

9.2 Exploring

Now that we created our new project, what can we do with it? In order to check the commands available to us, just run the **help** task:

```
$ mix help
```

It will print all the available tasks. You can get further information by invoking **mix help TASK**.

Play around with the available tasks, like **mix compile** and **mix test**, and execute them in your project to check how they work.

9.3 Compilation

Mix can compile our project for us. The default configurations uses **lib/** for source files and **ebin/** for compiled beam files. You don't even have to provide any compilation-specific setup but if you must, some options are available. For instance, if you want to put your compiled files in another directory besides **ebin**, simply set in **:compile_path** in your **mix.exs** file:

```
def project do
  [compile_path: "ebin"]
end
```

In general, Mix tries to be smart and compiles only when necessary.

Note that after you compile for the first time, Mix generates a `my_project.app` file inside your `ebin` directory. This file defines an Erlang application based on the contents of the `application` function in your Mix project.

The `.app` file holds information about the application, what are its dependencies, which modules it defines and so forth. The application is automatically started by Mix every time you run some commands and we will learn how to configure it in the next chapter.

9.4 Dependencies

Mix is also able to manage dependencies. Dependencies should be listed in the project settings, as follows:

```
def project do
  [ app: :my_project,
    version: "0.0.1",
    deps: deps ]
end

defp deps do
  [ { :some_project, github: "some_project/other", tag: "0.3.0" },
    { :another_project, ">= 1.0.2", git: "https://example.com/another/repo.git" } ]
end
```

Note: Although not required, it is common to split dependencies into their own function.

The dependency is represented by an atom, followed by an optional the version requirement and some options detailed in the next section. Since most of the dependencies are git repositories, it is recommended to depend on a particular tag. However, if you want to depend on master or in a particular branch,

you can define a version requirement to specify which versions of a given dependency you are willing to work against. It supports common operators like **>=**, **<=**, **>**, **==** as follows:

```
# Only version 2.0.0
"== 2.0.0"

# Anything later than 2.0.0
"> 2.0.0"
```

Requirements also support **and** and **or** for complex conditions:

```
# 2.0.0 and later until 2.1.0
">= 2.0.0 and < 2.1.0"
```

Since the example above is such a common requirement, it can be expressed as:

```
"~> 2.0.0"
```

Note that setting the version requirement does not affect the branch or tag that is checked out, so while a definition like the following is possible:

```
{ :some_project, "~> 0.5.0", github: "some_project/other", ta
```

It will lead to a dependency that will never be satisfied, because the tag being checked out does not match the version requirement.

9.4.1 Source Code Management (SCM)

In the example above, we have used **git** to specify our dependencies. Mix is designed in a way it can support multiple SCM tools, shipping with **:git** and **:path** support by default. The most common options are:

- **:git** - the dependency is a git repository that is retrieved and updated by Mix;

- `:path` - the dependency is simply a path in the filesystem;
- `:compile` - how to compile the dependency;
- `:app` - the path of the application expected to be defined by the dependency;
- `:env` - the environment to use from the dependency (more info below), defaults to `:prod`;

Each SCM may support custom options. `:git`, for example, supports the following:

- `:ref` - an optional reference (a commit) to checkout the git repository;
- `:tag` - an optional tag to checkout the git repository;
- `:branch` - an optional branch to checkout the git repository;
- `:submodules` - when true, initializes submodules recursively in the dependency;

9.4.2 Compiling dependencies

In order to compile a dependency, Mix looks into the repository for the best way to proceed. If the dependency contains one of the files below, it will proceed as follows:

1. `mix.exs` - compiles the dependency directly with Mix by invoking the `compile` task;
2. `rebar.config` or `rebar.config.script` - compiles using `rebar compile deps_dir=DEPS`, where `DEPS` is the directory where Mix will install the project dependencies by default;
3. `Makefile` - simply invokes `make`;

If the dependency does not contain any of the above, you can specify a command directly with the `:compile` option:

```
compile: "./configure && make"
```

If `:compile` is set to false, nothing is done.

9.4.3 Repeatability

An important feature in any dependency management tool is repeatability. For this reason when you first get your dependencies, Mix will create a file called `mix.lock` that contains checked out references for each dependency.

When another developer gets a copy of the same project, Mix will checkout exactly the same references, ensuring other developers can "repeat" the same setup.

Locks are automatically updated when `deps.update` is called and can be removed with `deps.unlock`.

9.4.4 Dependencies tasks

Elixir has many tasks to manage the project dependencies:

- `mix deps` - List all dependencies and their status;
- `mix deps.get` - Get all unavailable dependencies;
- `mix deps.compile` - Compile dependencies;
- `mix deps.update` - Update dependencies;
- `mix deps.clean` - Remove dependencies files;
- `mix deps.unlock` - Unlock the given dependencies;

Use `mix help` to get more information.

9.4.5 Dependencies of dependencies

If your dependency is another Mix or rebar project, Mix does the right thing: it will automatically fetch and handle all dependencies of your dependencies. However, if your project has two dependencies that share the same dependency and the SCM information for the shared dependency doesn't match between the parent dependencies, Mix will mark that dependency as diverged and emit a warning. To solve this issue you can declare the shared dependency in your project with the option `override: true` and Mix will use that SCM information to fetch the dependency.

9.5 Umbrella projects

It can be convenient to bundle multiple Mix projects together and run Mix tasks for them at the same time. They can be bundled and used together in what is called an umbrella project. An umbrella project can be created with the following command:

```
$ mix new project --umbrella
```

This will create a `mix.exs` file with the following contents:

```
defmodule Project.Mixfile do
  use Mix.Project

  def project do
    [ apps_path: "apps" ]
  end
end
```

The `apps_path` option specifies the directory where subprojects will reside. Mix tasks that run in the umbrella project will run for every project in the `apps_path` directory. For example `mix compile` or `mix test` will compile or test every project in the directory. It's important to note that an umbrella project is neither a regular Mix project, nor is it an OTP application nor can code source files be added.

If there are interdependencies between subprojects these have to be specified so that Mix can compile the projects in the correct order. If Project A depends on Project B, the dependency has to be specified in Project A's **mix.exs** file; modify the **mix.exs** file to specify the dependency:

```
defmodule A.Mixfile do
  use Mix.Project

  def project do
    [ app: :a,
      deps_path: "../../deps",
      lockfile: "../../mix.lock",
      deps: deps ]
  end

  defp deps do
    [ { :b, in_umbrella: true } ]
  end
end
```

Note the **deps_path** and **lockfile** options in the subproject above. If you have these options in all the subprojects in the umbrella they will share their dependencies. **mix new** inside the apps directory will automatically create a project with these options pre-set.

9.6 Environments

Mix has the concept of environments that allows a developer to customize compilation and other options based on an external setting. By default, Mix understands three environments:

- **dev** - the one in which mix tasks are run by default;
- **test** - used by **mix test**;
- **prod** - the environment in which dependencies are loaded and compiled;

By default, these environments behave the same and all configuration we have seen so far will affect all three environments. Customization per environment can be done using the **env:** option:

```
def project do
  [ env: [
    prod: [compile_path: "prod_ebin"] ] ]
end
```

Mix will default to the **dev** environment (except for tests). The environment can be changed via the **MIX_ENV** environment variable:

```
$ MIX_ENV=prod mix compile
```

In the next chapters, we will learn more about building OTP applications with Mix and how to create your own tasks.

Chapter 10

Building OTP apps with Mix

Where do we keep state in Elixir?

Our software needs to keep state, configuration values, data about the running system, etc. We have learned in previous sections how we can use processes/actors to keep state, receiving and responding to messages in a loop but this approach seems to be brittle. What happens if there is an error in our actor and it crashes? Even more, is it really required to create a new process when all we want to do is to keep simple configuration values?

In this chapter, we will answer those questions by building an OTP application. In practice, we don't need Mix in order to build such applications, however Mix provides some conveniences that we are going to explore throughout this chapter.

10.1 The Stacker server

Our application is going to be a simple stack that allow us push and pop items as we wish. Let's call it stacker:

```
$ mix new stacker --bare
```

Our application is going to have one stack which may be accessed by many processes at the same time. To achieve that, we will create a server that is re-

sponsible to manage the stack. Clients will send messages to the server whenever they want to push or pop something from the stack.

Since creating such servers is a common pattern when building Erlang and Elixir applications, we have a behavior in OTP that encapsulates common server functionalities called **GenServer**. Let's create a file named **lib/stacker/serve** with our first server:

```
defmodule Stacker.Server do
  use GenServer.Behaviour

  def init(stack) do
    { :ok, stack }
  end

  def handle_call(:pop, _from, [h|stack]) do
    { :reply, h, stack }
  end

  def handle_cast({ :push, new }, stack) do
    { :noreply, [new|stack] }
  end
end
```

Our server defines three callbacks: **init/1**, **handle_call/3** and **handle_cast/** We never call those functions directly, they are called by OTP whenever we interact with the server. We will go into details about these soon, let's just ensure it works as expected. To do so, run **iex -S mix** on your command line to start iex with mix and type the following:

```
# Let's start the server using Erlang's :gen_server module.
# It expects 3 arguments: the server module, the initial
# stack and some options (if desired):
iex> { :ok, pid } = :gen_server.start_link(Stacker.Server, [], [])
{:ok,<...>}

# Now let's push something onto the stack
iex> :gen_server.cast(pid, { :push, 13 })
:ok

# Now let's get it out from the stack
# Notice we are using *call* instead of *cast*
iex> :gen_server.call(pid, :pop)
13
```

Excellent, our server works as expected! There are many things happening behind the scenes, so let's discuss them one by one.

First, we started the server using the `:gen_server` module from OTP. Notice we have used `start_link`, which starts the server and links our current process to the server. In this scenario, if the server dies, it will send an exit message to our process, making it crash too. We will see this in action later. The `start_link` function returns the process identifier (`pid`) of the newly spawned server.

Later, we have sent a **cast** message to the `pid`. The message was `{ :push, 13 }`, written in the same format as we specified in the `handle_cast/2` callback in `Stacker.Server`. Whenever we send a `cast` message, the `handle_cast/2` callback will be invoked to handle the message.

Then we finally read what was on the stack by sending a **call** message, which will dispatch to the `handle_call/3` callback. So, what is the difference between **cast** and **call** after all?

`cast` messages are asynchronous: we simply send a message to the server and don't expect a reply back. That's why our `handle_cast/2` callback returns `{ :noreply, [new|stack] }`. The first item of the tuple states nothing should be replied and the second contains our updated stack with the new item.

On the other hand, `call` messages are synchronous. When we send a `call` message, the client expects a response back. In this case, the `handle_call/3` callback returns `{ :reply, h, stack }`, where the second item is the term to be returned and the third is our new stack without its head. Since `call`s are able to send messages back to the client, it also receives the client information as argument (`_from`).

10.1.1 Learning more about callbacks

In the GenServer's case, there are 8 different values a callback such as `handle_call` or `handle_cast` can return:

```
{ :reply, reply, new_state }
{ :reply, reply, new_state, timeout }
{ :reply, reply, new_state, :hibernate }
{ :noreply, new_state }
{ :noreply, new_state, timeout }
{ :noreply, new_state, :hibernate }
{ :stop, reason, new_state }
{ :stop, reason, reply, new_state }
```

There are 6 callbacks required to be implemented in a GenServer. The `GenServer.Behaviour` module defines all of them automatically but allows us to customize the ones we need. The list of callbacks are:

- `init(args)` - invoked when the server is started;
- `handle_call(msg, from, state)` - invoked to handle call messages;
- `handle_cast(msg, state)` - invoked to handle cast messages;
- `handle_info(msg, state)` - handle all other messages which are normally received by processes;
- `terminate(reason, state)` - called when the server is about to terminate, useful for cleaning up;
- `code_change(old_vsn, state, extra)` - called when the application code is being upgraded live (hot code swap);

10.1.2 Crashing a server

Of what use is a server if we cannot crash it?

It is actually quite easy to crash our server. Our `handle_call/3` callback only works if there is something on the stack (remember `[h|t]` won't match an empty list). So let's simply send a message when the stack is empty:

```
# Start another server, but with an initial :hello item
iex> { :ok, pid } = :gen_server.start_link(Stacker.Server, [:hello], [])
{:ok,<...>}

# Let's get our initial item:
iex> :gen_server.call(pid, :pop)
:hello

# And now let's call pop again
iex> :gen_server.call(pid, :pop)

=ERROR REPORT==== 6-Dec-2012::19:15:33 ===
...
** (exit) {{:function_clause,...}}
...
```

You can see there are two error reports. The first one is generated by server, due to the crash. Since the server is linked to our process, it also sent an exit message which was printed by `IEx` as `** (exit) ...`.

Since our servers may eventually crash, it is common to supervise them, and that's what we are going to next. There is a bit more to `GenServer` than what we have seen here. For more information, check `GenServer.Behaviour`'s documentation.

10.2 Supervising our servers

When building applications in Erlang/Elixir, a common philosophy is to "let it crash". Resources are going to become unavailable, timeout in between services are going to happen and other possible failures exist. That's why it is important to recover and react to such failures. With this in mind, we are going to write a supervisor for our server.

Create a file at `lib/stacker/supervisor.ex` with the following:

```
defmodule Stacker.Supervisor do
  use Supervisor.Behaviour

  # A convenience to start the supervisor
  def start_link(stack) do
    :supervisor.start_link(__MODULE__, stack)
```

```
  end

  # The callback invoked when the supervisor starts
  def init(stack) do
    children = [ worker(Stacker.Server, [stack]) ]
    supervise children, strategy: :one_for_one
  end
end
```

In case of supervisors, the only callback that needs to be implemented is **`init(args)`**. This callback needs to return a supervisor specification, in this case returned by the helper function **`supervise/2`**.

Our supervisor is very simple: it has to supervise one worker, in this case, **`Stacker.Server`** and the worker will be started by receiving one argument, which is the default stack. The defined worker is then going to be supervised using the **`:one_for_one`** strategy, which restarts each worker after it dies.

Given that our worker is specified by the **`Stacker.Server`** module passing the **`stack`** as argument, the supervisor will by default invoke the **`Stacker.Server.s`** function to start the worker, so let's implement it:

```
defmodule Stacker.Server do
  use GenServer.Behaviour

  def start_link(stack) do
    :gen_server.start_link({ :local, :stacker }, __MODULE__, stack, [])
  end

  def init(stack) do
    { :ok, stack }
  end

  def handle_call(:pop, _from, [h|stack]) do
    { :reply, h, stack }
  end

  def handle_cast({ :push, new }, stack) do
    { :noreply, [new|stack] }
  end
end
```

The **`start_link`** function is quite similar to how we were starting our server previously, except that now we passed one extra argument: **`{ :local,`**

`:stacker }`. This argument registers the server on our local nodes, allowing it to be invoked by the given name (in this case, `:stacker`), instead of directly using the `pid`.

With our supervisor in hand, let's start the console by running `iex -S mix` once again, which will recompile our files too:

```
# Now we will start the supervisor with a
# default stack containing :hello
iex> Stacker.Supervisor.start_link([:hello])
{:ok,<...>}

# And we will access the server by name since
# we registered it
iex> :gen_server.call(:stacker, :pop)
:hello
```

Notice the supervisor started the server for us and we were able to send messages to it via the name `:stacker`. What happens if we crash our server again?

```
iex> :gen_server.call(:stacker, :pop)

=ERROR REPORT==== 6-Dec-2012::19:15:33 ===
...
** (exit) {{:function_clause,...}}
...

iex> :gen_server.call(:stacker, :pop)
:hello
```

It crashes exactly as before but it is restarted right away by the supervisor with the default stack, allowing us to retrieve `:hello` again. Excellent!

By default the supervisor allows a worker to restart at maximum 5 times in a 5 seconds timespan. If the worker crashes more frequently than that, the supervisor gives up on the worker and no longer restarts it. Let's check it by sending 5 unknown messages one right after the other (be fast!):

```
iex> :gen_server.call(:stacker, :unknown)
... 5 times ...

iex> :gen_server.call(:stacker, :unknown)
** (exit) {:noproc,{:gen_server,:call,[:stacker,:unknown]}}
    gen_server.erl:180: :gen_server.call/2
```

The sixth message no longer generates an error report, since our server was no longer started automatically. Elixir returns `:noproc` (which stands for no process), meaning there isn't a process named `:stacker`. The number of restarts allowed and its time interval can be customized by passing options to the `supervise` function. Different restart strategies, besides the `:one_for_one` used above, can be chosen for the supervisor as well. For more information on the supported options, check the documentation for `Supervisor.Be`

10.3 Who supervises the supervisor?

We have built our supervisor but a pertinent question is: who supervises the supervisor? To answer this question, OTP contains the concept of applications. Applications can be started and stopped as an unit and, when doing so, they are often linked to a supervisor.

In the previous chapter, we have learned how Mix automatically generates an `.app` file every time we compile our project based on the information contained on the `application` function in our `mix.exs` file.

The `.app` file is called **application specification** and it must contain our application dependencies, the modules it defines, registered names and many others. Some of this information is filled in automatically by Mix but other data needs to be added manually.

In this particular case, our application has a supervisor and, furthermore, it registers a server with name `:stacker`. That said, it is useful to add to the **application specification** all registered names in order to avoid conflicts. If it happens that two applications register the same name, we will be able to find about this conflict sooner. So, let's open the `mix.exs` file and edit the `application` function to the following:

```
def application do
  [ registered: [:stacker],
    mod: { Stacker, [:hello] } ]
end
```

In the `:registered` key we specify all names registered by our application. The `:mod` key specifies that, as soon as the application is started, it must invoke the **application module callback**. In this case, the **application module callback** will be the `Stacker` module and it will receive the default stack `[:hello]` as argument. The callback must return the `pid` of the supervisor which is associated to this application.

With this in mind, let's open up the `lib/stacker.ex` file and add the following:

```
defmodule Stacker do
  use Application.Behaviour

  def start(_type, stack) do
    Stacker.Supervisor.start_link(stack)
  end
end
```

The `Application.Behaviour` expects two callbacks, `start(type, args)` and `stop(state)`. We are required to implement `start/2` though we have decided to not bother about `stop(state)` for now.

After adding the application behavior above, all you need to do is to start `iex -S mix` once again. Our files are going to be recompiled and the supervisor (and consequently our server) will be automatically started:

```
iex> :gen_server.call(:stacker, :pop)
:hello
```

Amazing, it works! As you may have noticed, the application `start/2` callback receives a type argument, which we have ignored. The type controls how the VM should behave when the supervisor, and consequently our application, crashes. You can learn more about it by reading the documentation for `Application.Behaviour`.

Finally, notice that `mix new` supports a `-sup` option, which tells Mix to generate a supervisor with an application module callback, automating some of the work we have done here. Try it!

10.4 Starting applications

We did not have to, at any point, start the application we have just defined. That's because Mix starts all applications, and all application dependencies, by default. We can start any application manually by calling functions from the :application module provided by OTP:

```
iex> :application.start(:stacker)
{ :error, { :already_started, :stacker } }
```

In this case, since the application was previously started, it returns so as an error message.

Mix does not only starts your application but all of your application dependencies. Notice there is a difference between your project dependencies (the ones defined under the `deps` key we have discussed in the previous chapter) and the application dependencies.

The project dependencies may contain your test framework or a compile-time only dependency. The application dependency is everything you depend on at runtime. Any application dependency needs to be explicitly added to the `application` function too:

```
def application do
  [ registered: [:stacker],
    applications: [:some_dep],
    mod: { Stacker, [:hello] } ]
end
```

When running tasks on Mix, it will ensure the application and all application dependencies are started.

10.5 Configuring applications

Besides the `:registered`, `:applications` and `:mod` keys we have seen above, applications also support configuration values that can be get and set explicitly.

Still in the command line, try:

```
iex> :application.get_env(:stacker, :foo)
:undefined
iex> :application.set_env(:stacker, :foo, :bar)
:ok
iex> :application.get_env(:stacker, :foo)
{ :ok, :bar }
```

This is a very useful mechanism for providing configuration values in your applications without a need to create the whole supervise chain. Default values for the application configuration can be defined in the `mix.exs` file as follows:

```
def application do
  [ registered: [:stacker],
    mod: { Stacker, [:hello] },
    env: [foo: :bar] ]
end
```

Now, leave the current shell and restart it with `iex -S mix`:

```
iex> :application.get_env(:stacker, :foo)
{ :ok, :bar }
```

For example, IEx and ExUnit are two applications that ship with Elixir that relies on such configuration values, as seen in their `mix.exs` files: IEx and ExUnit. Such applications then provide wrappers for reading and setting such values.

With this note, we finalize this chapter. We have learned how to create servers, supervise them, hook them into our application lifecycle and provide simple configuration options. In the next chapter, we will learn how to create custom tasks in Mix.

Chapter 11

Creating custom Mix tasks

In Mix, a task is simply an Elixir module inside the **`Mix.Tasks`** namespace containing a **`run/1`** function. For example, the **`compile`** task is a module named **`Mix.Tasks.Compile`**.

Let's create a simple task:

```
defmodule Mix.Tasks.Hello do
  use Mix.Task

  @shortdoc "This is short documentation, see"

  @moduledoc """
  A test task.
  """
  def run(_) do
    IO.puts "Hello, World!"
  end
end
```

Save this module to a file named **`hello.ex`** then compile and run it as follows:

```
$ elixirc hello.ex
$ mix hello
Hello, World!
```

The module above defines a task named `hello`. The function `run/1` takes a single argument that will be a list of binary strings which are the arguments that were passed to the task on the command line.

When you invoke `mix hello`, this task will run and print `Hello, World!`. Mix uses its first argument (`hello` in this case) to lookup the task module and execute its `run` function.

You're probably wondering why we have a `@moduledoc` and `@shortdoc`. Both are used by the `help` task for listing tasks and providing documentation. The former is used when `mix help TASK` is invoked, the latter in the general listing with `mix help`.

Besides those two, there is also `@hidden` attribute that, when set to true, marks the task as hidden so it does not show up on `mix help`. Any task without `@shortdoc` also won't show up.

11.1 Common API

When writing tasks, there are some common mix functionality we would like to access. There is a gist:

- `Mix.project` - Returns the project configuration under the function `project`; Notice this function returns an empty configuration if no `mix.exs` file exists in the current directory, allowing many Mix functions to work even if a `mix.exs` project is not defined;

- `Mix.Project.current` - Access the module for the current project, useful in case you want to access special functions in the project. It raises an exception if no project is defined;

- `Mix.shell` - The shell is a simple abstraction for doing IO in Mix. Such abstractions make it easy to test existing mix tasks. In the future, the shell will provide conveniences for colored output and getting user input;

- **`Mix.Task.run(task, args)`** - This is how you invoke a task from another task in Mix; Notice that if the task was already invoked, it works as no-op;

There is more to the Mix API, so feel free to check the documentation, with special attention to **`Mix.Task`** and **`Mix.Project`**.

11.2 Namespaced Tasks

While tasks are simple, they can be used to accomplish complex things. Since they are just Elixir code, anything you can do in normal Elixir you can do in Mix tasks. You can distribute tasks however you want just like normal libraries and thus they can be reused in many projects.

So, what do you do when you have a whole bunch of related tasks? If you name them all like **`foo`**, **`bar`**, **`baz`**, etc, eventually you'll end up with conflicts with other people's tasks. To prevent this, Mix allows you to namespace tasks.

Let's assume you have a bunch of tasks for working with Riak.

```
defmodule Mix.Tasks.Riak do
  defmodule Dostuff do
    ...
  end

  defmodule Dootherstuff do
    ...
  end
end
```

Now you'll have two different tasks under the modules **`Mix.Tasks.Riak.Dostuff`** and **`Mix.Tasks.Riak.Dootherstuff`** respectively. You can invoke these tasks like so: **`mix riak.dostuff`** and **`mix riak.dootherstuff`**. Pretty cool, huh?

You should use this feature when you have a bunch of related tasks that would be unwieldy if named completely independently of each other. If you have a few unrelated tasks, go ahead and name them however you like.

11.3 OptionParser

Although not a Mix feature, Elixir ships with an `OptionParser` which is quite useful when creating mix tasks that accepts options. The `OptionParser` receives a list of arguments and returns a tuple with parsed options and the remaining arguments:

```
OptionParser.parse(["--debug"])
#=> { [debug: true], [] }

OptionParser.parse(["--source", "lib"])
#=> { [source: "lib"], [] }

OptionParser.parse(["--source", "lib", "test/enum_test.exs", "--verbose"])
#=> { [source: "lib", verbose: true], ["test/enum_test.exs"] }
```

Check `OptionParser` documentation for more information.

11.4 Sharing tasks

After you create your own tasks, you may want to share them with other developers or re-use them inside existing projects. In this section, we will see different ways to share tasks in Mix.

11.4.1 As a dependency

Imagine you've created a Mix project called `my_tasks` which provides many tasks. By adding the `my_tasks` project as a dependency to any other project, all the tasks in `my_tasks` will be available in the parent project. It just works!

11.4.2 As an archive

Mix tasks are useful not only inside projects, but also to create new projects, automate complex tasks and to avoid repetitive work. For such cases, you want

a task always available in your workflow, regardless if you are inside a project or not.

For such cases, Mix allows developers to install and uninstall archives locally. To generate an archive for the current project and install it locally, run:

```
$ mix do archive, local.install
```

Archives can be installed from a path or any URL:

```
$ mix local.install http://example.org/path/to/sample/archive
```

After installing an archive, you can run all tasks contained in the archive, list them via **`mix local`** or uninstall the package via **`mix local.uninstall archive.ez`**.

11.4.3 MIX_PATH

The last mechanism for sharing tasks is **`MIX_PATH`**. By setting up your **`MIX_PATH`**, any task available in the **`MIX_PATH`** will be automatically visible to Mix. Here is an example:

```
$ export MIX_PATH="/full/path/to/my/project/ebin"
```

This is useful for complex projects that must be installed at **`/usr`** or **`/opt`** but still hook into Mix facilities.

With all those options in mind, you are ready to go out, create and install your own tasks! Enjoy!