Extracted from:

Programming Elixir

Functional |> Concurrent |> Pragmatic |> Fun

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

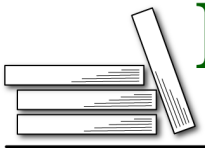
Programming Elixir

Functional
|> Concurrent
|> Pragmatic
|> Fun

Dave Thomas

Foreword by
José Valim,
Creator of Elixir

edited by Lynn Beighley



Pragmatic Bookshelf

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

Printed in the United States of America.
ISBN-13: 978-1-937785-58-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—May 22, 2013

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

CHAPTER 4

Elixir Basics

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

Elixir's built-in types are:

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

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

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

4.1 Value Types

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

Integers

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

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

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

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

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

Floating point numbers

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

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

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

Atoms

Atoms are constants that represent the name of something. They are written using a leading colon (:). The colon can be followed by an identifier[1] or an Elixir operator. You can also create atoms containing arbitrary characters by enclosing the characters following the colon in double quotes. (Ruby calls atoms *symbols*.) These are all atoms:

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

One thing that makes atoms invaluable is that their name is their value. Two atoms with the same name will always compare as being equal, even if they

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

were created by different applications sitting on two computers separated by an ocean.

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

Regular expressions

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

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

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

You manipulate regular expressions with the `Regex` module.

```
iex> Regex.run %r/[aieou]/, "caterpillar"
["a"]
iex> Regex.index %r/[aieou]/g, "caterpillar"
1
iex> Regex.scan %r/[aieou]/, "caterpillar"
["a","e","i","a"]
iex> Regex.split %r/[aieou]/, "caterpillar"
["c","t","rp","ll","r"]
iex> Regex.replace %r/[aieou]/, "caterpillar", "*"
"c*t*rp*ll*r"
```

4.2 System Types

These types reflect resources in the underlying Erlang VM.

2. http://www.pcre.org/

Pids and Ports

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

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

References

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

4.3 Collection Types

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

Elixir containers hold values of any type (including other containers).

Tuples

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

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

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

A typical Elixir tuple has two to four elements—any more and you'll probably want to look at records on page ?.

You can use tuples in pattern matching:

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

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

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

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

A common idiom is to write matches that assume success:

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

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

Lists

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

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

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

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

Elixir has some operators that work specifically on lists:

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

```
false
```

Keyword Lists

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

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

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

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

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

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

can we written more cleanly as

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

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

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

Binaries

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

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

The basic syntax packs successive integers into bytes:

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

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

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

```
11010101
iex> size bin
1
```

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

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

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

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

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

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

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

Source files names are written in lower case with underscores. They will have the extension .ex for programs that you intend to compile into binary form, and .exs for scripts that you want to run without compiling.

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

Truth

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

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

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

Ranges

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

Operators

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

comparison operators

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

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

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

boolean operators

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

```
a  or  b    # true if a is true, otherwise b
a  and b    # false if a is false or nil, otherwise b
not a       # false if a is true, true otherwise
```

relaxed boolean operators

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

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

arithmetic operators

`+ - * / div rem`

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

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

join operators

```
binary1 <> binary2   # concatenates two binaries (such as strings)
list1   ++ list2     # concatenates two lists
list1   -- list2     # set difference between two lists
```

the in operator

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

4.5 End of the Basics

We've now seen the low-level ingredients of an Elixir program. In the next chapter, we'll see how to organize these into modules and named functions.