Crystal for Rubyists

Serdar Dogruyol

Contents

Preamble

Nobody should only learn one programming language. Before Ruby, I wrote things in C, Java, PHP, and all sorts of things. I've always said that I love Ruby, but maybe someday, something would come along that'd be better for what I was building.

I'm not sure if Crystal is that language yet, but it is damn interesting. One of the worst things about Crystal is that there's next to no documentation, though. So, I decided that as I figured things out, I'd write them out so that you don't have to suffer the way I did. Maybe 'suffer' is too strong a word; Crystal is worth the hassle, though.

In this book, we'll talk about why you should care about Crystal, how to get up and running, the basics of writing software in Crystal, and maybe even something like building a Ruby gem with Crystal.

NOTE: While this book is called "Crystal for Rubyists," it should be accessible to anyone who knows about OOP and programming in a dynamically typed language. I will make analogies that will make the most sense to Rubyists, but you can still get a lot out of this book if you are a programmer of a different sort. If your favorite language is static, then you're already ahead of the game. You can just disregard a lot of the analogies.

Why Crystal?

You already write software in Ruby. It pays your bills. You enjoy it. Why should you care about Crystal?

Let's think about Ruby for a minute: what's its biggest weakness? For me, it's these things:

- Concurrency
- Speed
- Documentation

What's awesome about Ruby?

- Blocks
- Vaguely functional
- Syntax is pretty easy
- Focus on developer happiness
- Get up and running quickly
- Dynamically typed

So we could learn a lot from a language that's easy as Ruby, handles concurrency well, and is fast. We don't want to sacrifice anonymous functions, pretty syntax, or not making `AbstractFactoryFactoryImpls` just to get work done.

I think that language is *Crystal*.

Now: Crystal is not perfect. It is getting better. But the point is to *learn*. and using a language that's very familiar, yet very different, can teach us a lot.

Here's "Hello World" in Crystal:

```
puts "Hello, world!"
```

Here's a concurrent "Hello World" in Crystal:

```
channel = Channel(String).new
10.times do
  spawn {
    channel.send "Hello?"
  }
  puts channel.receive
end
```

Here's a rough port to Ruby:

```
10.times.map do
  Thread.new do
    puts "Hello?"
  end
end.each(&:join)
```

That's it. Note the stuff that's *similar* to Ruby:

- Pretty same syntax.
- Variables, while statically typed, have inference, so we don't need to declare types

Here's some stuff that's *different*:

- Being compiled and statically typed the compiler will yell at us if we mess up.

Oh, and:

```
$ time ./hello
./hello  0.00s user 0.00s system 73% cpu 0.008 total

$ time ruby hello.rb
ruby hello.rb  0.03s user 0.01s system 94% cpu 0.038 total
```

Five times faster. Yay irrelevant microbenchmarks!

Anyway, I hope you get my point: There's lots of things about Crystal that make it syntactically vaguely similar enough to Ruby that you can feel at home. And its strengths are some of Ruby's greatest weaknesses. That's why I think you can learn a lot from playing with Crystal, even if you don't do it as your day job.

Installing Crystal

Binary installers

The Crystal project provides official binary installers. You can get both releases and nightlies. Binary installers are the fastest and easiest way to get going with Crystal. Because Crystal is written in Crystal, compiling the Crystal compiler actually entails compiling it three times! This means it's quite slow. But a binary install should be snappy!

Crystal has a lovely installation page, so I recommend you just go check that out and download the proper version.

Note that this book has been tested with Crystal 0.12.0, and so if you use the latest nightly, something may have changed.

From Source

You will probably build the nightly version if you build from source, so be ready for some bugs in the code samples. This book was written for 0.10.0.

The Crystal README has great instructions for building form source. Just got follow their instructions!

Future Proofing

The version this book is written for is 0.12.0. While the language itself is pretty stable, things like the standard library and some major subsystems are being revised. I'll be tweaking it with every new release.

If you run

```
$ crystal
```

and it spits out a bunch of help information, you're good to go with Crystal.

Writing Your First Crystal Program

Okay! Let's get down to it: in order to call yourself an "X Programmer," you must write "Hello, world" in X. So let's do it. Open up a text file: I'll use `vim` because I'm that kind of guy, but use whatever you want. Crystal programs end in `.cr`:

```
$ vim hello.cr
```

Put this in it:

```
puts "Hello World!"
```

And run it with `crystal` command:

```
$ crystal hello.cr
Hello World!
```

It should run and print the output without error. If you get one, double check that you have the double quotation marks. Errors look like this:

```
$ crystal hello.cr
Syntax error in ./hello.cr:1: unterminated char literal, use double quotes for strings

puts 'Hello World!'
```

By the way `crystal` command is great for quickly running your code but it's slow. Each time it compiles and runs your program Let's see how much does it takes to run that program.

```
$ time crystal hello.cr
crystal hello.cr  0.30s user 0.20s system 154% cpu 0.326 total
```

0.326 seconds for a `Hello World`? Now that's slow.

Instead of compiling and running again you can compile it to native code with the `build` command.

```
$ crystal build hello.cr
```

And to run your program, do the Usual UNIX Thing:

```
$ time ./hello
./hello  0.00s user 0.00s system 87% cpu 0.006 total
```

You should see "Hello, world." print to the screen 50x faster :) Congrats!

Creating a new project

Up until now we've used the `crystal` command to only run our code.

Actually `crystal` command is pretty useful and does lot more than that. (check `crystal --help` for more)

For example we can use it to create a new Crystal project.

```
$ crystal init app sample
  create  sample/.gitignore
  create  sample/LICENSE
  create  sample/README.md
  create  sample/.travis.yml
  create  sample/shard.yml
  create  sample/src/sample.cr
  create  sample/src/sample/version.cr
  create  sample/spec/spec_helper.cr
  create  sample/spec/sample_spec.cr
Initialized empty Git repository in /Users/serdar/crystal_for_rubyists/code/04/sample/.git/
```

Awesome. `crystal` helped us create a new project. Let's see what it did for us.

- Created a new folder named sample
- Created a LICENSE
- Created `.travis.yml` to easily integrate Travis for continous integration.
- Created `shard.yml` for dependency management.
- Initialized an empty Git repository
- Created a README for our project
- Created `src` and `spec` folders to put our code and tests(ssh..we'll talk about it soon) in it.

Let's run it.

```
$ cd sample
$ crystal src/sample.cr
```

Nothing! Yay :)

Now that we create our first project. Let's use some external libraries.

Using Shards for dependency management

To manage dependencies of a project we use `shards`. `shards` is like `bundler` and `shard.yml` is like `Gemfile`.

Let's open up `shard.yml`.

```
name: sample
version: 0.1.0

authors:
  - sdogruyol <dogruyolserdar@gmail.com>

license: MIT
```

This is a default `shard.yml` and it contains the minimal necessary information about our project. Those are

- `name` specifies the name of the project
- `version` specifies the version of the project. Crystal itself uses semver for version management so it's a good convention for you to follow.
- `authors` section specifies the authors of the project. By default this is taken from your global `git` configuration.
- `license` specifies the type of your project license. By default this is `MIT`.

Okay. That's great but what can we do with this `shard.yml`? Well we can use this file to add external libraries(we call it dependency) and manage them without even worrying about any folders / paths e.g.. Sweet isn't it?

Now that we know the true power of `shards` let's add Kemal to our `shard.yml` and build a simple web application :)

Open up `shard.yml`. First we need to add `Kemal` as a dependency to our project. We do this by including

```
dependencies:
  kemal:
    github: sdogruyol/kemal
    version: 0.14.1
```

That's great! Now we added `Kemal` to our project. First, we need to install it.

```
$ shards install
Updating https://github.com/sdogruyol/kemal.git
Updating https://github.com/luislavena/radix.git
Updating https://github.com/jeromegn/kilt.git
Installing kemal (0.14.1)
Installing radix (0.3.0)
Installing kilt (0.3.3)
```

Okay now we are ready to use `Kemal` in our project. Open up `src/sample.cr`

```
require "./sample/*"
require "kemal"

module Sample

  get "/" do
    "Hello World!"
  end

end

Kemal.run
```

Look how we used `require` to access `Kemal` in our program.

Let's run.

```
$ crystal src/sample.cr
[development] Kemal is ready to lead at http://0.0.0.0:3000
```

Go to `localhost:3000` and see it in action!

Now you know how to add dependencies and use others' `shard`s :)

Testing

Rubyists love testing, so before we go any farther, let's talk about testing. In Crystal, there is a testing framework built in, and it's named `spec`. It's pretty similar to `RSpec`.

Let's continue with the project we created in Chapter 04.

As you remember `crystal` created this project structure for us.

```
$ cd sample && tree
-- LICENSE
-- README.md
-- shard.yml
-- spec
  -- sample_spec.cr
  -- spec_helper.cr
-- src
  -- sample
    -- version.cr
  -- sample.cr
```

Did you see that `spec` folder? Yes, as you guess Crystal created this folder and the first spec for us. In Crystal a file is tested with corresponding `_spec` file. Since we named our project as `sample` it created a file named `sample.cr` and the corresponding spec with `spec/sample_spec.cr`.

By the way, in this context `spec` and `unit test` means the same so we can use them interchangeably.

Without further ado lets open up `spec/sample_spec.cr`

```
require "./spec_helper"

describe Sample do
  # TODO: Write tests

  it "works" do
    false.should eq(true)
  end
end
```

Now this file is pretty interesting. There a three important keywords, `describe`, `it` and `should`.

Those keywords are only used in `specs` with the following purposes.

- `describe` lets you group related specs.
- `it` is used for defining a spec with the given title in between "".
- `should` is used for making assumptions about the spec.

As you can see this file has a group `described` as `Sample` and `it` has one spec with the title of `works` which makes the assumption that false `should` equal true.

You might be asking 'How do we run these tests?'. Well `crystal` command to the rescue.

```
$ crystal spec
F

Failures:

  1) Sample works
     Failure/Error: false.should eq(true)

       expected: true
            got: false

     # ./spec/sample_spec.cr:7

Finished in 0.69 milliseconds
1 examples, 1 failures, 0 errors, 0 pending

Failed examples:

crystal spec ./spec/sample_spec.cr:6 # Sample works
```

Yay! We got a failing(red) test. Reading the output we can easily find which spec failed. Here it's the spec within the group of `Sample` titled `works` a.k.a `Sample works`. Let's make it pass(green).

```
require "./spec_helper"

describe Sample do
  # TODO: Write tests

  it "works" do
    true.should eq(true)
  end
end
```

Rerun the specs.

```
$ crystal spec

.

Finished in 0.63 milliseconds
1 examples, 0 failures, 0 errors, 0 pending
```

Green! That's all you need to know to get started. Next up: FizzBuzz.

FizzBuzz

Of course, the first thing that your job interview for that cushy new Crystal job will task you with is building FizzBuzz. Let's do it!

If you're not familiar, FizzBuzz is a simple programming problem:

> "Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz"."

This will give us a good excuse to go over some basics of Crystal: Looping, tests, printing to standard output, and a host of other simple things.

First, let's create our project.

```
$ crystal init app fizzbuzz
```

Let's write our first failing test. Open up `/spec/fizzbuz_spec.cr`

```
require "./spec_helper"

describe Fizzbuzz do
  it "shouldn't divide 1 by 3" do
    div_by_three(1).should eq(false)
  end
end
```

And run it:

```
$ crystal spec
Error in ./spec/fizzbuzz_spec.cr:7: undefined method 'div_by_three'

div_by_three(1).should eq(false)
```

This makes sense: We haven't defined any method yet. Let's define one:

```
require "./fizzbuzz/*"

def div_by_three(n)
  false
end
```

Akin to Ruby, the value of the last expression gets returned.

TDD means do the simplest thing! Now that we've defined our method, let's compile and run our tests:

```
$  crystal spec
.

Finished in 0.82 milliseconds
1 examples, 0 failures, 0 errors, 0 pending
```

Awesome! We pass! Let's write another test, and see what happens:

```
require "./spec_helper"

describe Fizzbuzz do
  it "shouldn't divide 1 by 3" do
    div_by_three(1).should eq(false)
  end

  it "should divide 3 by 3" do
    div_by_three(3).should eq(true)
  end
end
```

Run it!

```
$ crystal spec

.F

Failures:

  1) Fizzbuzz should divide 3 by 3
     Failure/Error: div_by_three(3).should eq(true)

       expected: true
            got: false

     # ./spec/fizzbuzz_spec.cr:9

Finished in 0.83 milliseconds
2 examples, 1 failures, 0 errors, 0 pending

Failed examples:
```

```
crystal spec ./spec/fizzbuzz_spec.cr:8 # Fizzbuzz should divide 3 by 3
```

We have 1 failure. Let's make this pass.

```
require "./fizzbuzz/*"

def div_by_three(n)
  if n % 3 == 0
    true
  else
    false
  end
end
```

Run it.

```
$ crystal spec

..

Finished in 0.61 milliseconds
2 examples, 0 failures, 0 errors, 0 pending
```

Awesome! This shows off how `else` work, as well. It's probably what you expected. Go ahead and try to refactor this into a one-liner.

Done? How'd you do? Here's mine:

```
def div_by_three(n)
  n % 3 == 0
end
```

Remember, the value of the last expression gets returned.

Okay, now try to TDD out the `div_by_five` and `div_by_fifteen` methods. They should work the same way, but this will let you get practice actually writing it out. Once you see this, you're ready to advance:

```
$ crystal spec -v

Fizzbuzz
  shouldn't divide 1 by 3
  should divide 3 by 3
  shouldn't divide 8 by 5
```

```
  should divide 5 by 5
  shouldn't divide 13 by 15
  should divide 15 by 15

Finished in 0.61 milliseconds
6 examples, 0 failures, 0 errors, 0 pending
```

Okay! Let's talk about the main program now. We've got the tools to build FizzBuzz, let's make it work. First thing we need to do is print out all the numbers from one to 100. It's easy!

```
100.times do |num|
  puts num
end
```

Step one: print **something** 100 times. If you run this via `crystal build src/fizzbuzz.cr && ./fizzbuzz` you should see `num` printed 100 times. Note that our tests didn't actually run. Not only are they not run, they're actually not even in the executable:

Now we can put the two together:

```
100.times do |num|
  answer = ""

  if div_by_fifteen num
    answer = "FizzBuzz"
  elsif div_by_three num
    answer = "Fizz"
  elsif div_by_five num
    answer = "Buzz"
  else
    answer = num
  end

  puts answer
end
```

Because the `if` returns a value, we could also do something like this:

```
(1..100).each do |num|
  answer = if div_by_fifteen num
    "FizzBuzz"
  elsif div_by_three num
    "Fizz"
```

```
  elsif div_by_five num
    "Buzz"
  else
    num
  end

  puts answer
end
```

Notice that we also changed `100.times` to `(1..100).each`, to make `num` go from 1 to 100 instead of from 0 to 99.

Try running it.

Awesome! We've conquered FizzBuzz.

Types and Method Overloading

Crystal is like Ruby, but it's not Ruby!

Unlike Ruby, Crystal is a statically typed and compiled language. Most of the time you don't have to specify the types and the compiler is smart enough to do `type inference`.

So why do we need types? Let's start with something simple.

```
def add(x, y)
  x + y
end

add 3, 5 # 8
```

This is the same in Ruby! We just defined a method that adds two numbers. What if we try to add a number to a string?

```
add 3, "Serdar"
```

First let's do that in Ruby.

```
types.cr:2:in `+': String can't be coerced into Fixnum (TypeError)
    from types.cr:2:in `add'
    from types.cr:5:in `<main>'
```

What??? We just got a `TypeError` but we don't have to care about types in Ruby (or not :)). This is also a `runtime error` meaning that your program just crashed at runtime (definitely not good).

Now let's do the same in Crystal.

```
Error in ./types.cr:5: instantiating 'add(Int32, String)'

add 3, "Serdar"
^~~

in ./types.cr:2: no overload matches 'Int32#+' with types String
Overloads are:
- Int32#+(other : Int8)
- Int32#+(other : Int16)
- Int32#+(other : Int32)
- Int32#+(other : Int64)
- Int32#+(other : UInt8)
```

```
- Int32#+(other : UInt16)
- Int32#+(other : UInt32)
- Int32#+(other : UInt64)
- Int32#+(other : Float32)
- Int32#+(other : Float64)
- Int32#+()

x + y
  ^
```

Okay, that's quite a scary output but actually it's great. Our Crystal code didn't compile and also told us that there's no overload for `Int32#+` and showed us the possible overloads. This is a `compile time error` meaning that our code didn't compile and we catch the error before running the program. Lovely!

Now let's add some types and restrict that method to only accept `Number`s.

```
def add(x : Number, y : Number)
  x + y
end

puts add 3, "Serdar"
```

Run it.

```
Error in ./types.cr:5: no overload matches 'add' with types Int32, String
Overloads are:
- add(x : Number, y : Number)

puts add 3, "Serdar"
     ^~~
```

Awesome! Our program didn't compile again. And this time with shorter and more accurate error output. We just used `type restriction` on `x` and `y`. We restricted them to be `Number` and Crystal is smart enough to stop us from using the method with a `String`.

Method Overloading

We just saw a lot of overloads. Let's talk about `Method Overloading`.

Method overloading is having different methods with the same name and different number of arguments. They all have the same name but actually they are all different methods.

Let's overload our `add` method and make it work with a String.

```crystal
def add(x : Number, y : Number)
  x + y
end

def add(x: Number, y: String)
  x.to_s + y
end

puts add 3, 5

puts add 3, "Serdar"
```

Let's run it.

```
$ crystal types.cr
8
3Serdar
```

Now, that's method overloading in action. It figured out that we are calling the method with a Number and String and called the appropriate method. You can define as many overload methods as you wish.

Concurrency and Channels

Did you remember Chapter 1? We did a concurrent Hello World!

Here's a quick reminder.

```
channel = Channel(String).new
10.times do
  spawn {
    channel.send "Hello?"
  }
  puts channel.receive
end
```

In Crystal we use the keyword `spawn` to make something work in the background without blocking the main execution.

To achieve this `spawn` creates a lightweight thread called `Fiber`. `Fiber`s are very cheap to create and you can easily create tens of thousands of `Fiber`s on a single core.

Okay, that's really cool! We can use `spawn` to make stuff work in the background but how do we get something back from a `Fiber`.

Now that's where `Channel`s come to play.

Channel

As the name stands a `Channel` is a channel between a sender and the receiver. Therefore a `Channel` lets each other communicate with `send` and `receive` methods.

Let's take a line by line look at our previous example.

```
channel = Channel(String).new
```

We create a `Channel` with `Channel(String).new`. Note that we are creating a `Channel` which will `send` and `receive` messages with type of `String`.

```
10.times do
  spawn {
    channel.send "Hello?"
  }
  puts channel.receive
end
```

Leaving the loop aside, we are sending a message to our channel inside `spawn`. You might ask 'Why are we sending message in the background?' Well, `send` is a blocking operation and if we do that in the main program we gonna block the program forever.

Consider this:

```
channel = Channel(String).new
channel.send "Hello?" # This blocks the program execution
puts channel.receive
```

What's the output of this program? Actually this program won't ever finish because it gets blocked by `channel.send "Hello?"`. Now that we know why we use `spawn` to send a message let's continue.

```
spawn {
  channel.send "Hello?"
}
puts channel.receive
```

We just sent a message through our channel in the background with `spawn`. Then we receive it back with `channel.receive`. In this example the message is `Hello?` so this program prints `Hello?` and then finishes.

Macros and Metaprogramming

We love Ruby because of its' dynamic nature and metaprogramming! Unlike Ruby, Crystal is a compiled language. That's why there are some key differences.

- There's no `eval`.
- There's no `send`.

In Crystal we use `Macro`s to achieve this kind of behaviour and metaprogramming. You can think of `Macro`s as 'Code that writes/modifies code'.

P.S: `Macro`s are expanded into code at compile-time.

Check this.

```
macro define_method(name, content)
  def {{name}}
    {{content}}
  end
end

define_method foo, 1
# This generates:
#
#     def foo
#       1
#     end

foo # => 1
```

In the example we created a macro named `define_method` and we just called that macro like a normal method. That macro expanded into

```
  def foo
    1
  end
```

Pretty cool! We got `eval` behaviour at compile-time.

Macros are really powerful but there's one rule that you can't break.

A macro should expand into a valid Crystal program

C Bindings

There are lots of useful C libraries out there. It's important that we make use of them instead of rewriting every single of them.

In Crystal, It's super easy to use existing C libraries with bindings. Even Crystal itself uses C libraries.

For example Crystal uses `libpcre` for it's `Regex` implementation.

Like I said it's super easy to write bindings for C. Crystal itself links to `libpcre` like this

```
@[Link("pcre")]
lib LibPCRE
...
end
```

With just 3 lines of code you we're linked to `libpcre` :) We use `lib` keyword to group functions and types that belong to a library. And it's a good convetion to start with `Lib` for your C library declarations.

Next we bind to C functions with the `fun` keyword.

```
@[Link("pcre")]
lib LibPCRE
  type Pcre = Void*
  fun compile = pcre_compile(pattern : UInt8*, options : Int, errptr : UInt8**, erroffset :
end
```

Here we binded to `libpcre`s compile function with the matching types. Now we can easily access this function in our Crystal code.

```
LibPCRE.compile(..)
```