The Pragmatic Programmers

grox.io Learning

Programmer Passport

Pony

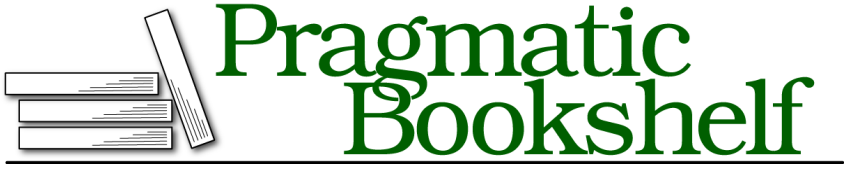
Bruce A. Tate

Edited by Jacquelyn Carter

Pony

Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

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

ISBN-13: pending

Encoded using the finest acid-free high-entropy binary digits.

Book version: B1.0—Monthname, yyyy

Contents

Change History

The book you're reading is in beta. This means that we update it frequently. Here is the list of the major changes that have been made at each beta release of the book, with the most recent change first.

B1.0: Nov 15, 2019

- The first Pony release! Welcome new Programmer Passport customers and thanks to our existing customers. Pony will be a bit more of a challenge for both you and me. We hope you enjoy it!

B2.0: Dec 1, 2019

- The second Pony release! We get to the reference capabilities. This is the hardest concept for Pony developers to grasp. Stay with it! You can do it! We hope you enjoy it!

B3.0: Dec 15, 2019

- The third Pony release! Sean Allen stepped up with some edits. This chapter will give you some of the tools to build programs using iterators, generics and a request-response style flow using promise chaining. If you've stayed with us this far, contratulations! Pony is a demanding language. Enjoy!

B4.0: Jan 1, 2020

- The fourth Pony release is the last one for us! This chapter will walk you through how to use Pony for real work. We'll do a command line based chat utility. Pony has good tools for both networking and command line processing. Hve fun!

CHAPTER 1

Pony Expressions

One of the truisms about studying programming languages is that you rarely encounter new ideas. As we examined Crystal, the syntax was nearly identical to Ruby's; the ideas in the type system are pretty widely available elsewhere and the concurrency model of fibers and channels is less popular but well-charted territory.

As we explore Pony together, you're going to see a huge splash of new ideas, especially the concept of *reference capabilities*, an idea that allows Pony to check concurrency correctness in the compiler. Here's how the Pony web site describes the new language:

> Pony is an open-source, object-oriented, actor-model, capabilities-secure, high-performance programming language.

That's a mouthful, but it's also a great introduction to the time we'll spend together. Let's break it down.

open source
: Pony is an open source language. Currently, there's no corporate backing. That means the community builds pragmatic features it most needs.

object-oriented
: Pony is object-oriented, though not in the way that you might think. It supports polymorphism through composition using interfaces rather than inheritance. (Pony also supports traits, but there's some talk about removing that feature so we'll steer clear of them in this book.)

actor-model
: Pony uses actor-based concurrency, meaning actor entities send and receive messages to and from each other, similar to what you'd find in Erlang or Elixir.

capabilities-secure

Capabilities-secure is probably a new concept to you unless you've encountered Pony or one of the esoteric languages it's based on before now. At a high level, Pony uses the type system in conjunction with capabilities to determine what permissions both objects and references have. We'll discuss this idea in more detail in the following chapters.

This list might already pique your curiosity. The takes on OOP and capabilities make it pretty much fully unique among programming languages you see today. I'll add that Pony is strongly typed with a tremendous emphasis on eliminating runtime errors, deadlocks, and data races. We're intentionally playing fast and loose with terminology, but we'll tighten this language up a little as we go.

In this chapter, we'll introduce you to the Pony basics. We'll tell you the story about how the language came to be, and then show you some programs that don't compile. By understanding what Pony programs don't compile, you'll get a feel for the various guarantees the type system gives you.

Then, we'll take you on a ride through the basic code organization constructs, from classes and actors to Pony primitives. Finally, we'll gallop through the main Pony control structures. Along the way, we'll begin to look into the type system, though a full exploration will take us beyond this brief chapter. When we're done, you'll have a working installation of Pony and know enough to code some trivial problems.

Let's get started.

The Pony Story

Some languages are defined by their lack of friction when you're starting out. Such languages make a bet: you'll get programs written quickly, but you'll pay a price in runtime errors. Python and Ruby fit this model.

Pony is not like those languages. Pony lays down a different bet. Relative to a more dynamic language, you'll have to work harder to make your program compile. There's a payoff for all of that work, though. Once your program does compile, it won't have runtime errors and it won't have concurrency problems such as data races or dead locks. We'll define those problems in more detail in the chapters to come. Pony's bet is that you'll save time in the long run by overcoming some early friction.

You might ask which approach is best. It depends on the problem you're trying to solve. As you might imagine, Pony was built to solve problems where

type mistakes and concurrency problems could cost hundreds of thousands of dollars, and both Python and Ruby were built to minimize friction and solve problems quickly. Both types of languages have their place.

To find this language's place, we need to go back to the beginning, before ponies roamed the land.

The Pre-Pony Days

In 2010, Sylvan Clebsch was writing C. He was building financial high-throughput systems and needed a way to make sure they would be correct. He started working on a C library, one that would enforce his constraints so that his concurrent programs would run quickly and correctly.

With the C libraries, "quickly" is rarely a problem. The problem with libraries is that they can't enforce the way the underlying code works. From the article An Early History of Pony,[1] Sylvan says "Programmers would send a pointer from one actor to another, convinced that it was safe, and it would turn out it wasn't."

So what's the problem?

Take a simple program, a bit of pseudocode in four parts. Let's say `balance` is a global variable that is used in two functions, `set_balance` and `get_balance`, and it looks like this:

```
module account
  function get_balance()
    return balance
  end

  function set_balance(new_balance)
    balance = new_balance
    return new_balance
  end
end
```

That code is straightforward. One function writes the balance and the other reads it. Additionally, say there is a function to add to that balance through a common interface called `credit` that looks like this:

```
function credit(amount)
  bal = get_balance()
  set_balance(bal + amount)
end
```

1. https://www.ponylang.io/blog/2017/05/an-early-history-of-pony/

Finally, there's a function to subtract from a balance called debit that looks similar:

```
function debit(amount)
  bal = get_balance()
  set_balance(bal - amount)
end
```

Let's say we're running a program that uses these functions in one thread. Perhaps we create an account with an initial balance of $100.00, then subtract $50.00, and then add $50.00, like this:

```
set_balance(100)  ->
                      set_balance(100)
                        ->                    balance: 100

debit(50) ->
                      get_balance()
                        100 <-                100

                      set_balance(50)
                        ->                    balance: 50


credit(50) ->
                      get_balance()
                        50- <-                50

                      set_balance(100)
                        ->                    balance: 100
                                                  :)
```

So we run our program, depositing $100, spending $50, and depositing another $50, and we correctly get back $100. Things are working perfectly.

Now, let's break the program down into two processes. The first process creates the account and credits it $50, and the second process debits the account $50. Then, we'll run those programs at the same time from processes P1 and P2. These instructions can be interleaved as the processor runs them like this:

```
P1:set_balance(100)  ->
                      set_balance(100)
                        ->                    balance: 100

P1:credit(50) ->
                      P1:get_balance()
                        100 <-                100
P2:debit(50) ->
                      P2:get_balance()
                        100 <-               100

                      P2:set_balance(50)
                        ->                    balance: 50
```

```
P1:set_balance(150)
  ->                    balance: 150
                             :(
```

Mr. Bank President, we have a problem. Because the instructions were interleaved, we can no longer guarantee the results. The bank is out $50! The problem is that the references in the four functions, bal and balance, all go back to the same place, and we share them. Mutable data and concurrency is problematic. The intermediate steps of our program are not designed to be run at the same time like this.

So it went with Sylvan's team. With a much more sophisticated program, it was tough to see from a distance which programs were safe to run and which were tainted, like this one. So, they abandoned the idea of writing a library where *the burden was on the programmer* and wrote a language where *the compiler took on the burden of guaranteeing the right results.*

Sylvan Scraps the Library, and Writes a Language

Usually, writing a language to solve a problem is a bad idea, unless many different people have the same problem, or *your need is so acute that building a language becomes the most practical solution.* Such was the case with Pony. By 2014, the small team started building Pony, and in 2015, they open sourced it all. Pony ambitiously focused on the problem of building a high-performance, distributed, concurrent, type-safe programs. Moreover, they wanted compiled programs to have several guarantees.

Some languages allow safe concurrency by making processes wait to use a reference when concurrency might be a problem. This locking solution solves concurrent access but with serious weaknesses:

- Locking depends on programmers to safeguard the right variables in the right way
- Locking introduces bottlenecks by creating critical resources in *data* as well as system services.
- Locking introduces deadlocks where processes are waiting on each other in order to finish, so that neither process can complete its task.

The Pony team solves this problem in another way. The solution zeroes in on the concurrency problem in our previous example: references. Here are the highlights.

- Variables have types based on the shape of the data (such as Bool or String), as in other languages.
- References to variables are also typed.

- Reference types also have information about how each reference is used.
- The compiler gives programs that use variables certain capabilities based on their types.
- References that can not be changed can be used with impunity; references that can be changed must be restricted. The compiler enforces these restrictions.

This description might be a little loose for you right now, but don't worry. The main gist of this solution is that incorrect programs don't compile, and there's no deadlock because there are no locks.

The Pony language has had some ups and downs but it is slowly picking up steam. Let's look at some tangible examples.

Pony in Three Compiler Errors

In this section, we're going to install the Pony compiler and then use it in some simple programs to demonstrate certain Pony guarantees. Fair warning. There's going to be some friction because it takes a while to get the hang of what Pony is doing. As you experience problems making something compile, take a little time to reflect on the payoff. Once your program compiles, it will have certain guarantees that you just don't get with other languages.

Install the Compiler

Pony is a relatively new language, and it doesn't have many of the build tools you might be used to. Basically, you're installing the `ponyc` compiler, and to use it, you'll point it at your source code directory and run it.

Go to the install page, follow the installation instructions[2] for your platform, and you'll be good to go. If you'd like a little peace of mind that your program works, just run the compiler like this:

```
➔ ponyc
Building builtin -> /usr/local/Cellar/ponyc/0.32.0/packages/builtin
Building . -> /Users/batate/...
Error:
no source files in package '.'
```

The error is normal because we haven't coded anything yet. Let's create a directory and put our program in it. We're running this version:

```
➔ ponyc -v
0.32.0 [release]
...
```

2. https://github.com/ponylang/ponyc/blob/master/README.md#installation

Since Pony hasn't yet reached 1.0, you'll want to note this version for the code examples in this book. Speaking of code examples, let's write some now!

Hello, World

It's finally time to write some code. Create a new directory called hello and change to that directory, like this:

```
➔ mkdir hello
➔ cd hello
```

Then, using your favorite editor, open up hello_pony.pony and type this in:

```
actor Main
  new create(env: Env) =>
    env.out.print("Ride that pony!")
```

It's the first Pony program of many we'll create. Let's unpack all of that code. If you don't get all the concepts right away, don't worry. We'll come back to them.

Let's start with the first line, actor Main. So Main is an actor, meaning it can have asynchronous methods called *behaviors*. We'll explore actors in more detail in the sections to come.

Programs must start somewhere. In Pony, the entry point is always the default constructor called dcreate of the Main actor. create must be present, and it must take an environment of type Env. Let's look at that constructor in our program.

The new keyword defines a constructor, and line new create(env:Env) => starts ours. In Pony, constructors have names and you can have as many of them as you want. The constructor named create is the default constructor. We pass an environment, called env and with type Env, describes the environment our program will run inside.

Look at the environment documentation.[3] You can see that these are the things env has, by virtue of its type Env:

- args : Array[String], the command-line arguments that the user typed when the user ran the program
- vars : Array[String], the environment variables from the environment used to start the program
- input : InStream, the standard mechanism for receiving input
- out : OutStream, the standard mechanism for producing normal output, in our case to the console

3. https://stdlib.ponylang.io/builtin-Env/

- stderr : OutStream, the standard mechanism for producing error output

Finally, let's look at the constructor's code:

```
`env.out.print("Ride that pony!")`
```

The . operator returns a field if there are no parentheses or calls a method if there are parentheses. So .env is a field (the environment for your program), out is the standard output stream, and print is a method on the stream. We pass that method a string, and Pony prints it out!

Whew! That's a lot of code for a first program. Now, we can compile it. We'll run ponyc from the directory with our source code, like this:

```
[hello_pony] ➔ ponyc
Building builtin -> /usr/local/Cellar/ponyc/0.32.0/packages/builtin
Building . -> /Users/batate/hello_pony
Generating
 Reachability
 Selector painting
 Data prototypes
 Data types
 Function prototypes
 Functions
 Descriptors
Optimising
Writing ./hello_pony.o
Linking ./hello_pony
```

You can see that the Pony compiler both compiled and linked the program. If you've worked with C, you know that linking is the process that connects your program with the libraries it needs. The output is the file hello_pony. To run it, just type it, like this:

```
[hello_pony] ➔ ./hello_pony
Ride that pony!
```

Hello, world, little Pony! Our program runs just fine. Let's make a few intentional mistakes to understand how Pony's guarantees might work.

Pony is Null Safe

In Pony, nothing returns a null. Some functions do return a special class called None, but not where you would typically get a null value. Take a look at this program:

```
actor Main
  new create(env: Env) =>
    let array: Array[String] = ["doesn't"; "matter"]
    let first: String = array(0)?
```

```
    env.out.print("This won't compile!")
```

We're checking what happens when you access an element of an array, since that's an operation that can fail if our user provides an index that's out of bounds. The third line, `let array: Array[String] = [...]`, defines an array. The fourth one, `let first: String = array(0)?`, accesses an element of that array.

The problem is that the indexing behavior in `array(0)?` might return an index out of bounds, so you need to deal with the error with a `try/else` block, like this:

```
actor Main
  new create(env: Env) =>
    let array: Array[String] = []
    try
      let first: String = array(0)?
    else
      "Oh snap!"
    end
    env.out.print("This will compile just fine!")
```

We'll get into the details later. For those wanting to peek ahead, `array(0)?` is actually a partial function, and we need to deal with all possible outcomes. We deal with the potential out of bounds error by returning a string with the `try/else` control structure. `try` does not work with exceptions. The `else` actually unwinds the stack one step and applies the `else`. If the `else` clause has a statement that might break, you'll need to wrap that statement in a `try/else` as well. The result is that your program must handle all possible exceptions.

So Pony's creators seek to avoid uncaught runtime errors at all costs. How far will Pony go? Check this out.

Go Ahead, Divide By Zero

Our little pony is alone in the room, and dressed like a pony. Here's an iconic example of what to do when the rules of efficiency, or aesthetics, or language design meet the rules of math. What happens when you divide by zero? If you're strictly following the rules of math, you'd want to throw an error. What does Pony do?

```
actor Main
  new create(env: Env) =>
    let x : U32 = 1
    let y : U32 = 0
    let z = x / y
    env.out.print("This won't crash!")
```

It doesn't crash. The result is actually defined to be zero. This behavior may bother some because in math, dividing by zero is almost universally undefined.