Pony for C# developers

We all love C# as a high-level language, with good performance and watched over by the .NET virtual machine in which it executes. It is highly applicable in a whole range of domains, from desktop to web and mobile applications, and Visual Studio offers us almost the nirvana of Integrated Development Environments.
So why would you ever leave this safe heaven, and start doing projects with a new and still quite unknown language called Pony?

Why Pony?
Pony stands for raw speed, because it compiles to native code, instead of being compiled at runtime by a JIT-compiler. Furthermore, its main focal point is concurrent and distributed applications, which is where software development should go, mirroring the hardware multicore evolution. But writing bug-free concurrent software with standard software tooling is extremely difficult: think about data races mutilating your data.

Pony is an open-source language whose development started around 2012 at Imperial College in London. It fully recognizes that the future world of software development will have to focus on multicore and distributed applications, and that these applications will have to guarantee safety and correct execution, meaning: no data-races.

These are Pony's two main focuses:

1) A highly-performant language with concurrency and parallel execution built-in as a first class goal. To achieve this, it implements the well-known actor model also used by Erlang and Akka, but with more guarantees and delivering better performance. Actors are the basic building blocks of a Pony app, and if needed, millions of actors (each consuming very little resources) can work in parallel together to achieve the goal of the application.

2) Safety on every front, so that no Pony program will ever crash. It does this:
- by applying strong type guarantees
- implementing a concurrent, per-actor garbage collection
- not allowing any null or non-initialized values
- no uncaught exceptions
- by supplementing the type system with additional qualifiers (so called *reference capabilities*) to make it safe to work on data with multiple actors, so as to avoid data-races.

These outstanding performance and guarantees are made possible by an ahead of time compiler, generating native code. This eliminates the need for a heavy, resource consuming virtual machine on your target platform.
As we will see, a lot of care has been taken to make sure that the language has a clear, very readable and easy-to-use and learn syntax, avoiding unnecessary cruft of braces and semicolons.
And as a bonus for us OOP programmers, objects, classes and interfaces still play an important part in Pony.

A first program.
To get a first feel for the syntax, let us compare the inevitable Hello-World program. Here is the C# version:

```
using System;

namespace HelloWorld
{
    class Program
```

```
        {
            static void Main(string[] args)
            {
                Console.WriteLine("Hello C# World!");
                Console.Read();
            }
        }
    }
```

And here is the Pony version:

```
    actor Main
      new create(env: Env) =>
        env.out.print("Hello, Pony world!")
```

We see a superficial resemblance: in both languages the action starts in something called `Main`. In C#, this is a static method from a class `Program`. In Pony however, `Main` is an actor which starts executing automatically in its `create` constructor. For safety reasons, Pony has no global variables, so the `Main` actor has to have a reference `Env` to its startup environment for program arguments and output, which is used here to print a string with `env.out.print`

What is immediately clear from this trivial example is how much easier and shorter the Pony code is, by using less syntactical cruft and avoiding curly braces as begin- and end-markers.

A more elaborate program:
As our next example we make a program that loops through some rectangles and calculates their circumference and area.
Here is the C# version (48 lines of code):

```
using System;

namespace Rectangle_Calculations
{
    class Program
    {
        static void Main(string[] args)
        {
            String output = "";
            for (int i = 1; i < 4; i++)
            {
                Rectangle rt = new Rectangle(i, i + 2);
                output = "Width and height: " + rt.ToString() +
                "\nCircumference: " + rt.Circumference() +
                "\nArea: " + rt.Area() + "\n";
                Console.WriteLine(output);
            }

        }
    }

    class Rectangle
    {
        private double width = 0;
        private double height = 0;
```

```
        public Rectangle(double width, double height) // constructor
        {
            this.width = width;
            this.height = height;
        }

        public override String ToString()
        {
            return width + " " + height;
        }

        public double Circumference()
        {
            return 2 * (width + height);
        }

        public double Area()
        {
            return width * height;
        }
    }
}
```

And here is the Pony version (30 lines of code):

```
use "collections"

actor Main
  new create(env: Env) =>                      // constructor
    var str: String = ""
    for i in Range[F32](1, 4) do
      let rt = Rectangle(i, i + 2)
      // make a String str with the info
      str = "Width and height: " + rt.get_width_and_height() +
            "\nCircumference: " + rt.circumference().string() +
            "\nArea: " + rt.area().string() + "\n"
      env.out.print(str)
    end

class Rectangle
  var _width: F32 = 0
  var _height: F32 = 0

  new create(width: F32, height: F32) =>      // constructor
    _width = width
    _height = height

  fun get_width_and_height(): String =>
    _width.string() + " " + _height.string()

  fun circumference(): F32 =>
    2 * (_width + _height)
```

```
  fun area(): F32 =>
     _width * _height
```

Both programs produce the same output:

```
/* Output:
Width and height: 1 3
Circumference: 8
Area: 3

Width and height: 2 4
Circumference: 12
Area: 8

Width and height: 3 5
Circumference: 16
Area: 15

*/
```

The C# and Pony code are quite similar, but because Pony doesn't use braces or semicolons, it gives a more succinct and readable impression.
Also note how Pony declares variables: `var _width: F32 = 0`
In contrast to the C# way: `private double width = 0;`
In Pony we can also use `let` instead of `var,` to indicate that the variable is immutable, like `readonly` in C#.

We see that Pony like C# can define classes and objects (`create` is the constructor). Methods are indicated with `fun`, and => starts the method body. A variable that starts with an _ is private in Pony. As in our first example, the program action starts in the `create` of actor `Main.`

An important difference is that in C# all code has to be in classes (here classes Program and class Rectangle), and there has to be an enclosing namespace. In Pony, actors and classes are concepts of equal importance. Actors are the building blocks for concurrency, so one could say that Pony embodies the two dimensions of object-oriented-ness and of concurrent execution.
Pony also has the concept of a package, roughly the equivalent of a C# namespace: all Pony source files within a certain directory belong to a package that has the same name as the directory they live in. Like `System` in C#, Pony has the `builtin` package, which is implicitly imported.
However we see in our Pony program that another package, namely `collections`, is imported with `use`. This is needed because we use a `Range` in the for-loop: `for i in Range[F32](1, 4)` … `end` , in contrast to the more primitive C-style `for` used in C#.

In the same way, Pony has an `if … then … elsif … then … else …, a while … do` or `repeat … until …` loop, and a `try … else … then` construct for error handling. All these constructs are terminated with the `end` keyword. Furthermore there is a very nice pattern matching, illustrated in the following code snippet:

```
actor Main
  let a: U64 = 2
  let b: U64 = 1
  new create(env : Env) =>
    env.out.print( match (a, b)
      | where a > b => "a is bigger than b"
      | where a < b => "b is bigger than a"
      else "they are the same"
      end )
```

This also illustrates that everything in Pony is an expression that returns a value. In this case match returns a string which is printed out.

Pony does not have the traditional inheritance as in C#: class A : B.
However there is subtyping using traits or interfaces (like in Java and Go):

```
trait Named
      fun name(): String => "John Doe"

class Person is Named
```

Objects of `class Person` can now use the `name()` method from `trait Named`, or `class Person` can provide its own.

A example program with interacting actors:
To get used to how actors interact with each other, have a look at the following program where actor Main engages an actor Counter to count to 10, after which Counter signals Main to output the result:

```
use "collections"

actor Counter
  var _count: U32

  new create() =>
    _count = 0

  be inc() =>
    _count = _count + 1

  be show_and_reset(main: Main) =>
    main.display(_count)
    _count = 0

actor Main
  var _env: Env

  new create(env: Env) =>
    _env = env

    var count: U32 = try env.args(1).u32() else 10 end
    var counter = Counter

    for i in Range[U32](0, count) do
      counter.inc()
    end

    counter.show_and_reset(this)

    be display(result: U32) =>
      _env.out.print(result.string())

// output:
// 10
```

In the line `var counter = Counter` an object of the Counter actor is created (`create` is called)
Then `inc()` is called in a for loop on that actor, and after the loop `show_and_reset(this)` is called.
We could as well have said that we send the Counter actor the `inc()` and `show_and_reset(this)` messages.
Indeed, they are not simply methods, they are annotated with `be`. This stands for a behaviour, which is a function that will be executed asynchronously (so not immediately). This is crucial to a concurrency framework: the actors do their jobs asynchronously, but an executing behaviour cannot be interrupted. At the end of the program `show_and_reset` calls the `display` behavior on the Main actor, passing it the counter result.

Reference capabilities

So far so good, but in order to make it strong guarantees for concurrent behavior, Pony has to add a second dimension as it were to its type system, called *reference capabilities*. These make it safe to both **pass** mutable data between actors and to **share** immutable data amongst actors, with no runtime overhead, that is no copying of data, nor any lock or other synchronization mechanism. The capabilities are checked at compile time, so that the generated code is guaranteed to keep data intact.

As a trivial example look at the two classes C# has for strings:

`class String`, for immutable strings
`class StringBuilder`, for mutable strings

In Pony, this difference is indicated by reference capabilities, in this case respectively `val` and `ref`:

`String val`, denotes an immutable string (`val` marks an object as globally immutable, that is: amongst all actors).

`String ref`, denotes a locally mutable string (`ref` marks an object as locally mutable, that is: the current actor can read an write it, but no other actor can).

Only 6 capabilities are necessary to make this system work, apart from `val` and `ref`, we also have:

`iso`: which means isolated to the current actor: only the current actor can work with the object
`box`: can be used to safely read the data by the current actor
`trn`: the object is only writable for the current actor
`tag`: the object can not be read or written to, only identified

Luckily the compiler understands and applies a great number of default capabilities, so that in practice they have to be used only in about 20% of cases. To complete the picture some other concepts are needed, such as ephemeral types, consume and recover, and viewpoint adaptation, but these are only fully needed when you get into the depths of programming in Pony.

We will now discuss a simple example that demonstrates the use of `ref` and `iso`, how the compiler prevents an unsafe situation, and how to remedy this in Pony:

```
class Foo
  var n: U32 = 5

  fun ref set(m: U32) =>
    n = m

  fun print(env: Env) =>
    env.out.print(n.string())

actor Doer
  be do1(env:Env, n: Foo iso) =>
    n.print(env)
```

```
actor Main
  new create(env: Env) =>
    let a = Doer.create()
    let b = Foo.create()
    a.do1(env, b)
```

Here is what happens: the Main actor creates a Doer actor a, and an object b of class Foo. It then asks Doer a to execute behaviour do1, passing it the b reference to the Foo object. do1 accepts this object as `n: Foo iso`, which means it wants to be able to act on the object in a completely isolated way. But this is not possible because the Main actor still has a reference b to the same object! The compiler rejects this code with the error *"argument not a subtype of parameter"* at `a.do1(env, b):` argument b is indeed not a subtype of parameter `n: Foo iso`, meaning the argument does not comply with the parameter.

How can we remedy this to get the output 5? The 1st way is to create the Foo object while passing it to do1, so that actor Main does not retain a reference to it:

```
actor Main
  new create(env: Env) =>
    let a = Doer.create()
    a.do1(env, Foo.create())
```

The 2nd way is to use a special Pony keyword `consume`, which as the name suggests destroys the local alias, effectively only leaving the reference n in actor Doer to the Foo object.

```
actor Main
  new create(env: Env) =>
    let a = Doer.create()
    let b = Foo.create()
    a.do1(env, consume b)
```

We also see that function `set` is annotated with `ref`: this is necessary because `set` changes the state of the `Foo` object by giving its field n a new value.

For which projects would a C# developer consider using Pony for development?

C# is largely targeted and bound to Windows environments, with some possibilities of running your program on a Linux platform. It still needs the .NET framework to be present on the target machine, which restricts the portability in the development of C# programs by a large amount, and makes it more Microsoft Windows dependent. Its virtual machine runtime makes it a no-go for real-time systems.

But Pony is written in C an so can run on any platform where C can run. Moreover it can interface very easily with C, as well as being called by C. Pony is optimized towards running millions of actors in a distributed real-time system.

Pony will excel especially for applications where concurrency and distribution play a crucial role, such as: high-performance financial systems, data analysis applications, video games, physical simulations, cryptography, etc.

Getting started

Visit the Pony website (http://www.ponylang.org/) and especially the tutorial (http://tutorial.ponylang.org/).
At the moment to run Pony on Windows you have to be brave and build the compiler from source, which is explained in detail at http://tutorial.ponylang.org/getting-started/installation.html; prebuilt binaries for Windows will be available soon.
If you don't want to go through the trouble of installing Pony, you can try it out in this web sandbox environment: http://sandbox.ponylang.org/

As to good programming tools, Sublime Text and Atom both have good plugins for Pony. A Visual Studio plugin also exists for installation in VS 2013 (https://github.com/ponylang/VS-pony).

If you are running into questions, ask on Stack Overflow (http://stackoverflow.com/questions/tagged/ponylang), in the mailing list IRC channel (https://groups.io/g/pony+user), or join `#ponylang` on freenode. Have fun!