Pony tutorial

Contents

Pony Tutorial

Welcome to the Pony tutorial! If you're reading this, chances are you want to learn Pony. That's great, we're going to make that happen.

This tutorial is aimed at people who have some experience programming already. It doesn't really matter if you know a little Python, or a bit of Ruby, or you are a JavaScript hacker, or Java, or Scala, or C/C++, or Haskell, or OCaml: as long as you've written some code before, you should be fine.

What's Pony, anyway?

Pony is an object-oriented, actor-model, capabilities-secure programming language. It's **object-oriented** because it has classes and objects, like Python, Java, C++, and many other languages. It's **actor-model** because it has *actors* (similar to Erlang, Elixir, or Akka). These behave like objects, but they can also execute code *asynchronously*. Actors make Pony awesome.

When we say Pony is **capabilities-secure**, we mean a few things:

- It's type safe. Really type safe. There's a mathematical proof and everything.
- It's memory safe. Ok, this comes with type safe, but it's still interesting. There are no dangling pointers, no buffer overruns, heck, the language doesn't even have the concept of *null*!
- It's exception safe. There are no runtime exceptions. All "exceptional situations" have defined semantics, and they are *always* handled.
- It's data-race free. Pony doesn't have locks or atomic operations or anything like that. Instead, the type system ensures *at compile time* that your concurrent program can never have data races. So you can write highly concurrent code and never get it wrong.
- It's deadlock free. This one is easy, because Pony has no locks at all! So they definitely don't deadlock, because they don't exist.

Pony can't stop you from writing logical bugs, but it can remove entire classes of bugs from being possible. The Pony compiler prevents you from unsafely accessing memory concurrently.

If you've ever done concurrent programming, you know how hard such things can be to track down. With Pony, **poof**, don't worry about it; concentrate on "your logic". We think that makes Pony awesome and we hope you come to agree with us.

We'll talk more about capabilities-security, including both **object capabilities** and **reference capabilities** later.

The Pony Philosophy: Get Stuff Done

In the spirit of Richard Gabriel, the Pony philosophy is neither "the-right-thing" nor "worse-is-better". It is "get-stuff-done".

- Correctness. Incorrectness is simply not allowed. *It's pointless to try to get stuff done if you can't guarantee the result is correct.*
- Performance. Runtime speed is more important than everything except correctness. If performance must be sacrificed for correctness, try to come up with a new way to do things. *The faster the program can get stuff done, the better. This is more important than anything except a correct result.*
- Simplicity. Simplicity can be sacrificed for performance. It is more important for the interface to be simple than the implementation. *The faster the programmer can get stuff done, the better. It's ok to make things a bit harder on the programmer to improve performance, but it's more important to make things easier on the programmer than it is to make things easier on the language/runtime.*
- Consistency. Consistency can be sacrificed for simplicity or performance. *Don't let excessive consistency get in the way of getting stuff done.*
- Completeness. It's nice to cover as many things as possible, but completeness can be sacrificed for anything else. *It's better to get some stuff done now than wait until everything can get done later.*

The "get-stuff-done" approach has the same attitude towards correctness and simplicity as "the-right-thing", but the same attitude towards consistency and completeness as "worse-is-better". It also adds performance as a new principle, treating it as the second most important thing (after correctness).

Guiding Principles

Throughout the design and development of the language the following principles should be adhered to.

- Use the get-stuff-done approach.
- Simple grammar. Language must be trivial to parse for both humans and computers.
- No loadable code. Everything is known to the compiler.
- Fully type safe. There is no "trust me, I know what I'm doing" coercion.
- Fully memory safe. There is no "this random number is really a pointer, honest."
- No crashes. A program that compiles should never crash (although it may hang or do something unintended).
- Sensible error messages. Where possible use simple error messages for specific error cases. It is fine to assume the programmer knows the definitions of words in our lexicon, but avoid compiler or other computer science jargon.

- Inherent build system. No separate applications required to configure or build.
- Aim to reduce common programming bugs through the use of restrictive syntax.
- Provide a single, clean and clear way to do things rather than catering to every programmer's preferred prejudices.
- Make upgrades clean. Do not try to merge new features with the ones they are replacing, if something is broken remove it and replace it in one go. Where possible provide rewrite utilities to upgrade source between language versions.
- Reasonable build time. Keeping down build time is important, but less important than runtime performance and correctness.
- Allowing the programmer to omit some things from the code (default arguments, type inference, etc) is fine, but fully specifying should always be allowed.
- No ambiguity. The programmer should never have to guess what the compiler will do, or vice-versa.
- Document required complexity. Not all language features have to be trivial to understand, but complex features must have full explanations in the docs to be allowed in the language.
- Language features should be minimally intrusive when not used.
- Fully defined semantics. The semantics of all language features must be available in the standard language docs. It is not acceptable to leave behaviour undefined or "implementation dependent".
- Efficient hardware access must be available, but this does not have to pervade the whole language.
- The standard library should be implemented in Pony.
- Interoperability. Must be interoperable with other languages, but this may require a shim layer if non primitive types are used.
- Avoid library pain. Use of 3rd party Pony libraries should be as easy as possible, with no surprises. This includes writing and distributing libraries and using multiple versions of a library in a single program.

More help

Working your way through the tutorial but in need of more help? Not to worry, we have you covered.

If you are looking for an answer "right now", we suggest you give our Zulip community a try. Whatever your question is, it isn't dumb, and we won't get annoyed.

Think you've found a bug? Check your understanding first by writing to the "beginner help" stream in Zulip. Once you know it's a bug, open an issue.

Help us

Found a typo in this tutorial? Perhaps something isn't clear? We welcome pull requests against the tutorial: Pony Tutorial.

Be sure to check out the contribution guidelines before opening your PR.

Getting Started

This chapter will get you up and running with Pony from installing the compiler to running your first program.

As you work your way through this tutorial, you'll likely come across a lot of concepts that are familiar to you from any prior programming languages that you've had experience with. You'll likely want to skim and skip around through these areas, and that's totally fine.

However, if you've never used Pony before, we guarantee that you'll come across concepts that are new to you, which will require close and careful attention if you want to learn and apply them: reference capabilities. These are the core innovation in Pony that make it a unique and compelling offering in the wide world of modern programming languages.

In this tutorial we start off with familiar basics, and try our best to avoid reference capabilities in the code examples until later on when they can be covered in detail. You should feel free to follow along with the code examples in your own text editor - we absolutely encourage it. Just be aware that as you venture off the beaten path of the curated tutorial, you'll likely run into reference capabilities, and you'll need to thoroughly read and understand the basics of those concepts before you'll really feel fluent and able to work with the compiler as it tries to help you prove the safety of your program.

Stick with us and read on, even if you need to read through something a few times. Know that the community is here to help as you climb the learning curve and main new concepts that will change the way you think about concurrency.

What You Need

To get started, you'll need a text editor and the ponyc compiler. Or if you are on a not supported platform or don't want to install the compiler you can use the Pony's Playground.

The Pony compiler

Before you get started, please check out the installation instructions for the Pony compiler.

A text editor

While you can write code using any editor, it's nice to use one with some support for the language. We maintain a list of editors supporting Pony.

The compiler

Pony is a *compiled* language, rather than an *interpreted* one. In fact, it goes even further: Pony is an *ahead-of-time* (AOT) compiled language, rather than a *just-in-time* (JIT) compiled language.

What this means is that once you build your program, you can run it over and over again without needing a compiler or a virtual machine or anything else. It's a complete program, all on its own.

But it also means you need to build your program before you can run it. In an interpreted language or a JIT compiled language, you tend to do things like this to run your program:

```
python helloworld.py
```

Or maybe you put a **shebang** in your program (like `#!/usr/bin/env python`), then `chmod` to set the executable bit, and then do:

```
./helloworld.py
```

When you use Pony, you don't do any of that!

Compiling your program

If you are in the same directory as your program, you can just do:

```
ponyc
```

That tells the Pony compiler that your current working directory contains your source code, and to please compile it. If your source code is in some other directory, you can tell ponyc where it is:

```
ponyc path/to/my/code
```

There are other options as well, but we'll cover those later.

Hello World – Your First Pony Program

Now that you've successfully installed the Pony compiler, let's start programming! Our first program will be a very traditional one. We're going to print "Hello, world!". First, create a directory called `helloworld`:

```
mkdir helloworld
cd helloworld
```

Does the name of the directory matter? Yes, it does. It's the name of your program! By default when your program is compiled, the resulting executable binary will have the same name as the directory your program lives in. You can also set the name using the –bin-name or -b options on the command line.

The code

Then, create a file in that directory called `main.pony`.

Does the name of the file matter? Not to the compiler, no. Pony doesn't care about filenames other than that they end in `.pony`. But it might matter to you! By giving files good names, it can be easier to find the code you're looking for later.

In your file, put the following code:

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

hello-world-main.pony

Compiling the program

Now compile it:

```
$ ponyc
Building .
Building builtin
```

```
Generating
Optimising
Writing ./helloworld.o
Linking ./helloworld
```

(If you're using Docker, you'd write something like `$ docker run -v Some_Absolute_Path/helloworld:/src` `ponylang/ponyc`, depending of course on what the absolute path to your `helloworld` directory is.)

Look at that! It built the current directory, `.`, plus the stuff that is built into Pony, `builtin`, it generated some code, optimised it, created an object file (don't worry if you don't know what that is), and linked it into an executable with whatever libraries were needed. If you're a C/C++ programmer, that will all make sense to you, otherwise, it probably won't, but that's ok, you can ignore it.

Wait, it linked too? Yes. You won't need a build system (like `make`) for Pony. It handles that for you (including handling the order of dependencies when you link to C libraries, but we'll get to that later).

Running the program

Now we can run the program:

```
$ ./helloworld
Hello, world!
```

Congratulations, you've written your first Pony program! Next, we'll explain what some of that code does.

Hello World – How It Works

Let's look at our `helloworld` code again:

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

hello-world-main.pony

Let's go through that line by line.

Line 1

```
actor Main
```

hello-world-main.pony:1:1

This is a **type declaration**. The keyword `actor` means we are going to define an actor, which is a bit like a class in Python, Java, C#, C++, etc. Pony has classes too, which we'll see later.

The difference between an actor and a class is that an actor can have **asynchronous** methods, called **behaviours**. We'll talk more about that later.

A Pony program has to have a `Main` actor. It's kind of like the `main` function in C or C++, or the `main` method in Java, or the `Main` method in C#. It's where the action starts.

Line 2

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

hello-world-main.pony:2:2

This is a **constructor**. The keyword `new` means it's a function that creates a new instance of the type. In this case, it creates a new **Main**.

Unlike other languages, constructors in Pony have names. That means there can be more than one way to construct an instance of a type. In this case, the name of the constructor is `create`.

The parameters of a function come next. In this case, our constructor has a single parameter called `env` that is of the type `Env`.

In Pony, the type of something always comes after its name and is separated by a colon. In C, C++, Java or C#, you might say `Env env`, but we do it the other way around (like Go, Pascal, Rust, TypeScript, and a bunch of other languages).

It turns out, our `Main` actor **has** to have a constructor called `create` that takes a single parameter of type `Env`. That's how all programs start! So the beginning of your program is essentially the body of that constructor.

Wait, what's the body? It's the code that comes after the `=>`.

Line 3

```
    env.out.print("Hello, world!")
```

hello-world-main.pony:3:3

This is your program! What the heck is it doing?

In Pony, a dot is either a field access or a method call, much like other languages. If the name after the dot has parentheses after it, it's a method call. Otherwise, it's a field access.

So here, we start with a reference to `env`. We then look up the field `out` on our object `env`. As it happens, that field represents `stdout`, i.e. usually it means printing to your console. Then, we call the `print` method on `env.out`. The stuff inside the parentheses are the arguments to the function. In this case, we are passing a **string literal**, i.e. the stuff in double quotes.

In Pony, string literals can be in double quotes, `"`, in which case they follow C/C++ style escaping (using stuff like `\n`), or they can be triple-quoted, `"""` like in Python, in which case they are considered raw data.

What's an `Env`, anyway? It's the "environment" your program was invoked with. That means it has command line arguments, environment variables, `stdin`, `stdout`, and `stderr`. Pony has no global variables, so these things are explicitly passed to your program.

That's it!

Really, that's it. The program begins by creating a `Main` actor, and in the constructor, we print "Hello, world!" to `stdout`. Next, we'll start diving into the Pony type system.

Types

Pony's type system is what makes it special. There's plenty to love about it otherwise but, in the end, it's the type system that contains much of what makes Pony novel. In this chapter, we are going to explore the basics of the type system. If you worked with a statically typed language before, there shouldn't be anything surprising to you. By the time you've finished this chapter, you should have a handle on basics of the Pony type system.

The Pony Type System at a Glance

Pony is a *statically typed* language, like Java, C#, C++, and many others. This means the compiler knows the type of everything in your program. This is different from *dynamically typed* languages, such as Python, Lua, JavaScript, and Ruby.

Static vs Dynamic: What's the difference?

In both kinds of language, your data has a type. So what's the difference?

With a *dynamically typed* language, a variable can point to objects of different types at different times. This is flexible, because if you have a variable `x`, you can assign an integer to it, then assign a string to it, and your compiler or interpreter doesn't complain.

But what if I try to do a string operation on `x` after assigning an integer to it? Generally speaking, your program will raise an error. You might be able to handle the error in some way, depending on the language, but if you don't, your program will crash.

When you use a *statically typed* language, a variable has a type. That is, it can only point to objects of a certain type (although in Pony, a type can actually be a collection of types, as we'll see later). If you have an `x` that expects to point to an integer, you can't assign a string to it. Your compiler complains, and it complains **before** you ever try to run your program.

Types are guarantees

When the compiler knows what types things are, it can make sure some things in your program work without you having to run it or test it. These things are the *guarantees* that a language's type system provides.

The more powerful a type system is, the more things it can prove about your program without having to run it.

Do dynamic types make guarantees too? Yes, but they do it at runtime. For example, if you call a method that doesn't exist, you will usually get some kind of exception. But you'll only find out when you try to run your program.

What guarantees does Pony's type system give me?

The Pony type system offers a lot of guarantees, even more than other statically typed languages.

- If your program compiles, it won't crash.
- There will never be an unhandled exception.
- There's no such thing as `null`, so your program will never try to dereference `null`.
- There will never be a data race.
- Your program will never deadlock.
- Your code will always be capabilities-secure.
- All message passing is causal. (Not **casual**!)

Some of those will make sense right now. Some of them may not mean much to you yet (like capabilities-security and causal messaging), but we'll get to those concepts later on.

If I use Pony's FFI to call code written in another language, does Pony make the same guarantees for the code I call? Sadly, no. Pony's type system can only guarantee code written in Pony. Code written in other languages get only the guarantees provided by that language.

Pony requires FFI parameters to be explicitly declared as Pony types in advance. The type system provides a guarantee at compile-time that no code-paths exist that violate the definitions you provided on the Pony side of your FFI call.

This guarantee cannot be inclusive of foreign code as the compiler has no way to verify that it won't violate Pony's type system guarantees at runtime.

Classes

Just like other object-oriented languages, Pony has **classes**. A class is declared with the keyword `class`, and it has to have a name that starts with a capital letter, like this:

```
class Wombat
```

classes-wombat.pony:9:9

Do all types start with a capital letter? Yes! And nothing else starts with a capital letter. So when you see a name in Pony code, you will instantly know whether it's a type or not.

What goes in a class?

A class is composed of:

1. Fields.
2. Constructors.
3. Functions.

Fields

These are just like fields in C structures or fields in classes in C++, C#, Java, Python, Ruby, or basically any language, really. There are three kinds of fields: `var`, `let`, and `embed` fields. A `var` field can be assigned to over and over again, but a `let` field is assigned to in the constructor and never again. Embed fields will be covered in more detail in the documentation on variables.

```
class Wombat
  let name: String
  var _hunger_level: U64
```

classes-wombat.pony:9:11

Here, a `Wombat` has a `name`, which is a `String`, and a `_hunger_level`, which is a `U64` (an unsigned 64-bit integer).

What does the leading underscore mean? It means something is **private**. A **private** field can only be accessed by code in the same type. A **private** constructor, function, or behaviour can only be accessed by code in the same package. We'll talk more about packages later.

Constructors

Pony constructors have names. Other than that, they are just like constructors in other languages. They can have parameters, and they always return a new instance of the type. Since they have names, you can have more than one constructor for a type.

Constructors are introduced with the **new** keyword.

```
class Wombat
  let name: String
  var _hunger_level: U64

  new create(name': String) =>
    name = name'
    _hunger_level = 0

  new hungry(name': String, hunger': U64) =>
    name = name'
    _hunger_level = hunger'
```

classes-wombat-constructors.pony:8:18

Here, we have two constructors, one that creates a `Wombat` that isn't hungry, and another that creates a `Wombat` that might be hungry or might not. Unlike some other languages that differentiate between constructors with method overloading, Pony won't presume to know which alternate constructor to invoke based on the arity and type of your arguments. To choose a constructor, invoke it like a method with the `.` syntax:

```
    let defaultWombat = Wombat("Fantastibat") // Invokes the create method by default
    let hungryWombat = Wombat.hungry("Nomsbat", 12) // Invokes the hungry method
```

classes-wombat.pony:3:4

What's with the single quote thing, i.e. name'? You can use single quotes in parameter and local variable names. In mathematics, it's called a *prime*, and it's used to say "another one of these, but not the same one". Basically, it's just convenient.

Every constructor has to set every field in an object. If it doesn't, the compiler will give you an error. Since there is no `null` in Pony, we can't do what Java, C# and many other languages do and just assign either `null` or zero to every field before the constructor runs, and since we don't want random crashes, we don't leave fields undefined (unlike C or C++).

Sometimes it's convenient to set a field the same way for all constructors.

```
class Wombat
  let name: String
  var _hunger_level: U64
  var _thirst_level: U64 = 1

  new create(name': String) =>
    name = name'
    _hunger_level = 0

  new hungry(name': String, hunger': U64) =>
    name = name'
    _hunger_level = hunger'
```

classes-wombat.pony:9:20

Here, every `Wombat` begins a little bit thirsty, regardless of which constructor is called.

Zero Argument Constructors

```
class Hawk
  var _hunger_level: U64 = 0
```

```
class Owl
  var _hunger_level: U64

  new create() =>
    _hunger_level = 42
```

classes-zero-argument-constructors.pony:9:16

Here we have two classes, because the `Hawk` class defines no constructors, a default constructor with zero arguments called `create` is generated. The `Owl` defines its own constructor that sets the `_hunger_level`.

When constructing instances of classes that have zero-argument constructors, they can be constructed with just the class name:

```
class Forest
  let _owl: Owl = Owl
  let _hawk: Hawk = Hawk
```

classes-zero-argument-constructors.pony:5:7

This is explained later, in more detail in the sugar section.

Functions

Functions in Pony are like methods in Java, C#, C++, Ruby, Python, or pretty much any other object oriented language. They are introduced with the keyword `fun`. They can have parameters like constructors do, and they can also have a result type (if no result type is given, it defaults to `None`).

```
class Wombat
  let name: String
  var _hunger_level: U64
  var _thirst_level: U64 = 1

  new create(name': String) =>
    name = name'
    _hunger_level = 0

  new hungry(name': String, hunger': U64) =>
    name = name'
    _hunger_level = hunger'

  fun hunger(): U64 => _hunger_level

  fun ref set_hunger(to: U64 = 0): U64 => _hunger_level = to
```

classes-wombat.pony:9:24

The first function, `hunger`, is pretty straight forward. It has a result type of `U64`, and it returns `_hunger_level`, which is a `U64`. The only thing a bit different here is that no `return` keyword is used. This is because the result of a function is the result of the last expression in the function, in this case, the value of `_hunger_level`.

Is there a `return` keyword in Pony? Yes. It's used to return "early" from a function, i.e. to return something right away and not keep running until the last expression.

The second function, `set_hunger`, introduces a *bunch* of new concepts all at once. Let's go through them one by one.

- The `ref` keyword right after `fun`

This is a **reference capability**. In this case, it means the *receiver*, i.e. the object on which the `set_hunger` function is being called, has to be a `ref` type. A `ref` type is a **reference type**, meaning that the object is **mutable**. We need this because we are writing a new value to the `_hunger_level` field.

What's the receiver reference capability of the `hunger` method? The default receiver reference capability if none is specified is `box`, which means "I need to be able to read from this, but I won't write to it".

What would happen if we left the `ref` keyword off the `set_hunger` method? The compiler would give you an error. It would see you were trying to modify a field and complain about it.

- The `= 0` after the parameter `to`

This is a **default argument**. It means that if you don't include that argument at the call site, you will get the default argument. In this case, `to` will be zero if you don't specify it.

- What does the function return?

It returns the *old* value of `_hunger_level`.

Wait, seriously? The *old* value? Yes. In Pony, assignment is an expression rather than a statement. That means it has a result. This is true of a lot of languages, but they tend to return the *new* value. In other words, given `a = b`, in most languages, the value of that is the value of `b`. But in Pony, the value of that is the *old* value of `a`.

...why? It's called a "destructive read", and it lets you do awesome things with a capabilities-secure type system. We'll talk about that later. For now, we'll just mention that you can also use it to implement a *swap* operation. In most languages, to swap the values of `a` and `b` you need to do something like:

```
var temp = a
a = b
b = temp
```

classes-swap-values.pony:6:8

In Pony, you can just do:

```
a = b = a
```

classes-swap-values-sugar.pony:6:6

Finalisers

Finalisers are special functions. They are named `_final`, take no parameters and have a receiver reference capability of `box`. In other words, the definition of a finaliser must be `fun _final()`.

The finaliser of an object is called before the object is collected by the GC. Functions may still be called on an object after its finalisation, but only from within another finaliser. Messages cannot be sent from within a finaliser.

Finalisers are usually used to clean up resources allocated in C code, like file handles, network sockets, etc.

What about inheritance?

In some object-oriented languages, a type can *inherit* from another type, like how in Java something can **extend** something else. Pony doesn't do that. Instead, Pony prefers *composition* to *inheritance*. In other words, instead of getting code reuse by saying something **is** something else, you get it by saying something **has** something else.

On the other hand, Pony has a powerful **trait** system (similar to Java 8 interfaces that can have default implementations) and a powerful **interface** system (similar to Go interfaces, i.e. structurally typed).

We'll talk about all that stuff in detail later.

Naming rules

All names in Pony, such as type names, method names, and variable names may contain only **ASCII** characters.

In fact all elements of Pony code are required to be ASCII, except string literals, which happily accept any kind of bytes directly from the source file, be it `UTF-8` encoded or `ISO-8859-2` and represent them in their encoded form.

A Pony type, whether it's a class, actor, trait, interface, primitive, or type alias, must start with an uppercase letter. After an underscore for private or special *methods* (behaviors, constructors, and functions), any method or variable, including parameters and fields, must start with a lowercase letter. In all cases underscores in a row or at the end of a name are not allowed, but otherwise, any combination of letters and numbers is legal.

In fact, numbers may use single underscores inside as a separator too!

Only variable names can end in primes (`'`).

Primitives

A **primitive** is similar to a **class**, but there are two critical differences:

1. A **primitive** has no fields.
2. There is only one instance of a user-defined **primitive**.

Having no fields means primitives are never mutable. Having a single instance means that if your code calls a constructor on a **primitive** type, it always gets the same result back (except for built-in "machine word" primitives, covered below).

What can you use a primitive for?

There are three main uses of primitives (four, if you count built-in "machine word" primitives).

1. As a "marker value". For example, Pony often uses the **primitive** `None` to indicate that something has "no value". Of course, it *does* have a value, so that you can check what it is, and the value is the single instance of `None`.
2. As an "enumeration" type. By having a **union** of **primitive** types, you can have a type-safe enumeration. We'll cover **union** types later.
3. As a "collection of functions". Since primitives can have functions, you can group functions together in a primitive type. You can see this in the standard library, where path handling functions are grouped in the **primitive** `Path`, for example.

```
// 2 "marker values"
primitive OpenedDoor
primitive ClosedDoor

// An "enumeration" type
type DoorState is (OpenedDoor | ClosedDoor)

// A collection of functions
primitive BasicMath
  fun add(a: U64, b: U64): U64 =>
    a + b

  fun multiply(a: U64, b: U64): U64 =>
    a * b

actor Main
  new create(env: Env) =>
    let doorState : DoorState = ClosedDoor
    let isDoorOpen : Bool = match doorState
      | OpenedDoor => true
      | ClosedDoor => false
    end
    env.out.print("Is door open? " + isDoorOpen.string())
    env.out.print("2 + 3 = " + BasicMath.add(2,3).string())
```

primitives-doors.pony

Primitives are quite powerful, particularly as enumerations. Unlike enumerations in other languages, each "value" in the enumeration is a complete type, which makes attaching data and functionality to enumeration values easy.

Built-in primitive types

The `primitive` keyword is also used to introduce certain built-in "machine word" types. Other than having a value associated with them, these work like user-defined primitives. These are:

- `Bool`. This is a 1-bit value that is either `true` or `false`.
- `ISize`, `ILong`, `I8`, `I16`, `I32`, `I64`, `I128`. Signed integers of various widths.
- `USize`, `ULong`, `U8`, `U16`, `U32`, `U64`, `U128`. Unsigned integers of various widths.
- `F32`, `F64`. Floating point numbers of various widths.

`ISize/USize` correspond to the bit width of the native type `size_t`, which varies by platform. `ILong/ULong` similarly correspond to the bit width of the native type `long`, which also varies by platform. The bit width of a native `int` is the same across all the platforms that Pony supports, and you can use `I32/U32` for this.

Primitive initialisation and finalisation

Primitives can have two special functions, `_init` and `_final`. `_init` is called before any actor starts. `_final` is called after all actors have terminated. The two functions take no parameter. The `_init` and `_final` functions for different primitives always run sequentially.

A common use case for this is initialising and cleaning up C libraries without risking untimely use by an actor.

Actors

An **actor** is similar to a **class**, but with one critical difference: an actor can have **behaviours**.

Behaviours

A **behaviour** is like a **function**, except that functions are *synchronous* and behaviours are *asynchronous*. In other words, when you call a function, the body of the function is executed immediately, and the result of the call is the result of the body of the function. This is just like method invocation in any other object-oriented language.

But when you call a behaviour, the body is **not** executed immediately. Instead, the body of the behaviour will execute at some indeterminate time in the future.

A behaviour looks like a function, but instead of being introduced with the keyword `fun`, it is introduced with the keyword `be`.

Like a function, a behaviour can have parameters. Unlike a function, it doesn't have a receiver capability (a behaviour can be called on a receiver of any capability) and you can't specify a return type.

So what does a behaviour return? Behaviours always return `None`, like a function without explicit result type, because they can't return something they calculate (since they haven't run yet).

```
actor Aardvark
  let name: String
  var _hunger_level: U64 = 0

  new create(name': String) =>
    name = name'

  be eat(amount: U64) =>
    _hunger_level = _hunger_level - amount.min(_hunger_level)
```

actors-behaviors.pony

Here we have an `Aardvark` that can eat asynchronously. Clever Aardvark.

Message Passing

If you are familiar with actor-based languages like Erlang, you are familiar with the concept of "message passing". It's how actors communicate with one another. Behaviours are the Pony equivalent. When you call a behavior on an actor, you are sending it a message.

If you aren't familiar with message passing, don't worry about it. We've got you covered. All will be explained below.

Concurrent

Since behaviours are asynchronous, it's ok to run the body of a bunch of behaviours at the same time. This is exactly what Pony does. The Pony runtime has its own cooperative scheduler, which by default has a number of threads equal to the number of CPU cores on your machine. Each scheduler thread can be executing an actor behaviour at any given time, so Pony programs are naturally concurrent.

Sequential

Actors themselves, however, are sequential. That is, each actor will only execute one behaviour at a time. This means all the code in an actor can be written without caring about concurrency: no need for locks or semaphores or anything like that.

When you're writing Pony code, it's nice to think of actors not as a unit of parallelism, but as a unit of sequentiality. That is, an actor should do only what *has* to be done sequentially. Anything else can be broken out into another actor, making it automatically parallel.

In the example below, the `Main` actor calls a behaviour `call_me_later` which, as we know, is *asynchronous*, so we won't wait for it to run before continuing. Then, we run the method `env.out.print`, which is *also asynchronous*, and will print the provided text to the terminal. Now that we've finished executing code inside the `Main` actor, the behaviour we've called earlier will eventually run, and it will print the last text.

```
actor Main
  new create(env: Env) =>
    call_me_later(env)
    env.out.print("This is printed first")

  be call_me_later(env: Env) =>
    env.out.print("This is printed last")
```

actors-sequential.pony

Since all this code runs inside the same actor, and the calls to the other behaviour `env.out.print` are sequential as well, we are guaranteed that `"This is printed first"` is always printed **before** `"This is printed last"`.

Why is this safe?

Because of Pony's **capabilities secure type system**. We've mentioned reference capabilities briefly before when talking about function receiver reference capabilities. The short version is that they are annotations on a type that make all this parallelism safe without any runtime overhead.

We will cover reference capabilities in depth later.

Actors are cheap

If you've done concurrent programming before, you'll know that threads can be expensive. Context switches can cause problems, each thread needs a stack (which can be a lot of memory), and you need lots of locks and other mechanisms to write thread-safe code.

But actors are cheap. Really cheap. The extra cost of an actor, as opposed to an object, is about 256 bytes of memory. Bytes, not kilobytes! And there are no locks and no context switches. An actor that isn't executing consumes no resources other than the few extra bytes of memory.

It's pretty normal to write a Pony program that uses hundreds of thousands of actors.

Actor finalisers

Like classes, actors can have finalisers. The finaliser definition is the same (`fun _final()`). All guarantees and restrictions for a class finaliser are also valid for an actor finaliser. In addition, an actor will not receive any further message after its finaliser is called.

Traits and Interfaces

Like other object-oriented languages, Pony has **subtyping**. That is, some types serve as *categories* that other types can be members of.

There are two kinds of **subtyping** in programming languages: **nominal** and **structural**. They're subtly different, and most programming languages only have one or the other. Pony has both!

Nominal subtyping

This kind of subtyping is called **nominal** because it is all about **names**.

If you've done object-oriented programming before, you may have seen a lot of discussion about *single inheritance*, *multiple inheritance*, *mixins*, *traits*, and similar concepts. These are all examples of **nominal subtyping**.

The core idea is that you have a type that declares it has a relationship to some category type. In Java, for example, a **class** (a concrete type) can **implement** an **interface** (a category type). In Java, this means the class is now in the category that the interface represents. The compiler will check that the class actually provides everything it needs to.

In Pony, nominal subtyping is done via the keyword `is`. `is` declares at the point of declaration that an object has a relationship to a category type. For example, to use nominal subtyping to declare that the class `Name` provides `Stringable`, you'd do:

```
class Name is Stringable
```

traits-and-interfaces-nominal-subtyping.pony

Structural subtyping

There's another kind of subtyping, where the name doesn't matter. It's called **structural subtyping**, which means that it's all about how a type is built, and nothing to do with names.

A concrete type is a member of a structural category if it happens to have all the needed elements, no matter what it happens to be called.

Structural typing is very similar to duck typing from dynamic programming languages, except that type compatibility is determined at compile time rather than at run time. If you've used Go, you'll recognise that Go interfaces are structural types.

You do not declare structural relationships ahead of time, instead it is done by checking if a given concrete type can fulfill the required interface. For example, in the code below, we have the interface `Stringable` from the standard library. Anything can be used as a `Stringable` so long as it provides the method `fun string(): String iso^`. In our example below, `ExecveError` provides the `Stringable` interface and can be used anywhere a `Stringable` is needed. Because `Stringable` is a structural type, `ExecveError` doesn't have to declare a relationship to `Stringable`, it simply has that relationship because it has “the same shape”.

```
interface box Stringable
  """
  Things that can be turned into a String.
  """
  fun string(): String iso^
    """
    Generate a string representation of this object.
  """

primitive ExecveError
  fun string(): String iso^ => "ExecveError".clone()
```

traits-and-interfaces-structural-subtyping.pony

Nominal and structural subtyping in Pony

When discussing subtyping in Pony, it is common to say that `trait` is nominal subtyping and `interface` is structural subtyping, however, that isn't really true.

Both `trait` and `interface` can establish a relationship via nominal subtyping. Only `interface` can be used for structural subtyping.

Nominal subtyping in Pony

The primary means of doing nominal subtyping in Pony is using **traits**. A **trait** looks a bit like a **class**, but it uses the keyword `trait` and it can't have any fields.

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

class Bob is Named
```

traits-and-interfaces-trait.pony:6:9

Here, we have a trait `Named` that has a single function `name` that returns a String. It also provides a default implementation of `name` that returns the string literal "Bob".

We also have a class `Bob` that says it `is Named`. This means `Bob` is in the `Named` category. In Pony, we say *Bob provides Named*, or sometimes simply *Bob is Named*.

Since `Bob` doesn't have its own `name` function, it uses the one from the trait. If the trait's function didn't have a default implementation, the compiler would complain that `Bob` had no implementation of `name`.

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

trait Bald
  fun hair(): Bool => false

class Bob is (Named & Bald)
```

traits-and-interfaces-multiple-traits.pony:6:12

It is possible for a class to have relationships with multiple categories. In the above example, the class `Bob` *provides both Named and Bald.*

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

trait Bald is Named
  fun hair(): Bool => false

class Bob is Bald
```

traits-and-interfaces-nested-traits.pony:6:12

It is also possible to combine categories together. In the example above, all `Bald` classes are automatically `Named`. Consequently, the `Bob` class has access to both hair() and name() default implementation of their respective trait. One can think of the `Bald` category to be more specific than the `Named` one.

```
class Larry
  fun name(): String => "Larry"
```

traits-and-interfaces-nominal-subtyping-in-pony.pony:6:7

Here, we have a class `Larry` that has a `name` function with the same signature. But `Larry` does **not** provide `Named`!

Wait, why not? Because `Larry` doesn't say it `is Named`. Remember, traits are **nominal**: a type that wants to provide a trait has to explicitly declare that it does. And `Larry` doesn't.

You can also do nominal subtyping using the keyword `interface`. **Interfaces** in Pony are primarily used for structural subtyping. Like traits, interfaces can also have default method implementations, but in order to use default method implementations, an interface must be used in a nominal fashion. For example:

```
interface HasName
  fun name(): String => "Bob"

class Bob is HasName

class Larry
  fun name(): String => "Larry"
```

traits-and-interfaces-nominal-and-structural-subtyping.pony:8:14

Both `Bob` and `Larry` are in the category `HasName`. `Bob` because it has declared that it is a `HasName` and `Larry` because it is structurally a `HasName`.

Structural subtyping in Pony

Pony has structural subtyping using **interfaces**. Interfaces look like traits, but they use the keyword `interface`.

```
interface HasName
  fun name(): String
```

traits-and-interfaces-structural-subtyping-in-pony.pony

Here, `HasName` looks a lot like `Named`, except it's an interface instead of a trait. This means both `Bob` and `Larry` provide `HasName`! The programmers that wrote `Bob` and `Larry` don't even have to be aware that `HasName` exists.

Differences between traits and interfaces

It is common for new Pony users to ask, **Should I use traits or interfaces in my own code?** Both! Interfaces are more flexible, so if you're not sure what you want, use an interface. But traits are a powerful tool as well.

Private methods

A key difference between traits and interfaces is that interfaces can't have private methods. So, if you need private methods, you'll need to use a trait and have users opt in via nominal typing. Interfaces can't have private methods because otherwise, users could use them to break encapsulation and access private methods on concrete types from other packages. For example:

```
actor Main
  new create(env: Env) =>
    let x: String ref = "sailor".string()
    let y: Foo = x
    y._set(0, 'f')
    env.out.print("Hello, " + x)

interface Foo
  fun ref _set(i: USize, value: U8): U8
```

traits-and-interfaces-private-methods.pony

In the code above, the interface `Foo` allows access to the private `_set` method of `String` and allows for changing `sailor` to `failor` or it would anyway, if interfaces were allowed to have private methods.

Open world enumerations

Traits allow you to create "open world enumerations" that others can participate in. For example:

```
trait Color

primitive Red is Color
primitive Blue is Color
```

traits-and-interfaces-open-world-enumerations.pony

Here we are using a trait to provide a category of things, `Color`, that any other types can opt into by declaring itself to be a `Color`. This creates an "open world" of enumerations that you can't do using the more traditional Pony approach using type unions.

```
primitive Red
primitive Blue

type Color is (Red | Blue)
```

traits-and-interfaces-type-union.pony

In our trait based example, we can add new colors at any time. With the type union based approach, we can only add them by modifying definition of `Color` in the package that provides it.

Interfaces can't be used for open world enumerations. If we defined `Color` as an interface:

```
interface Color
```

traits-and-interfaces-marker-methods.pony:1:1

Then literally everything in Pony would be a `Color` because everything matches the `Color` interface. You can however, do something similar using "marker methods" with an interface:

```
interface Color
  fun is_color(): None

primitive Red
  fun is_color(): None => None
```

traits-and-interfaces-marker-methods.pony

Here we are using structural typing to create a collection of things that are in the category `Color` by providing a method that "marks" being a member of the category: `is_color`.

Open world typing

We've covered a couple ways that traits can be better than interfaces, let's close with the reason for why we say, unless you have a reason to, you should use `interface` instead of `trait`. Interfaces are incredibly flexible. Because traits only provide nominal typing, a concrete type can only be in a category if it was declared as such by the programmer who wrote the concrete type. Interfaces allow you to create your own categorizations on the fly, as you need them, to group existing concrete types together however you need to.

Here's a contrived example:

```
interface Compactable
  fun ref compact()
  fun size(): USize

class Compactor
  """
  Compacts data structures when their size crosses a threshold
  """
  let _threshold: USize

  new create(threshold: USize) =>
    _threshold = threshold

  fun ref try_compacting(thing: Compactable) =>
    if thing.size() > _threshold then
      thing.compact()
    end
```

traits-and-interfaces-open-world-typing.pony:20:36

The flexibility of `interface` has allowed us to define a type `Compactable` that we can use to allow our `Compactor` to accept a variety of data types including `Array`, `Map`, and `String` from the standard library.

Structs

A `struct` is similar to a `class`. There's a couple very important differences. You'll use classes throughout your Pony code. You'll rarely use structs. We'll discuss structs in more depth in the C-FFI chapter of the tutorial. In the meantime, here's a short introduction to the basics of structs.

Structs are “classes for FFI”

A `struct` is a class like mechanism used to pass data back and forth with C code via Pony’s Foreign Function Interface.

Like classes, Pony structs can contain both fields and methods. Unlike classes, Pony structs have the same binary layout as C structs and can be transparently used in C functions. Structs do not have a type descriptor, which means they cannot be used in algebraic types or implement traits/interfaces.

What goes in a struct?

The same as a class! A struct is composed of some combination of:

1. Fields
2. Constructors
3. Functions

Fields

Pony struct fields are defined in the same way as they are for Pony classes, using `embed`, `let`, and `var`. An embed field is embedded in its parent object, like a C struct inside C struct. A var/let field is a pointer to an object allocated separately.

For example:

```
struct Inner
  var x: I32 = 0

struct Outer
  embed inner_embed: Inner = Inner
  var inner_var: Inner = Inner
```

structs-fields.pony

Constructors

Struct constructors, like class constructors, have names. Everything you previously learned about Pony class constructors applies to struct constructors.

```
struct Pointer[A]
  """
  A Pointer[A] is a raw memory pointer. It has no descriptor and thus can't be
  included in a union or intersection, or be a subtype of any interface. Most
  functions on a Pointer[A] are private to maintain memory safety.
  """
  new create() =>
    """
    A null pointer.
    """
    compile_intrinsic

  new _alloc(len: USize) =>
    """
    Space for len instances of A.
    """
    compile_intrinsic
```

structs-constructors.pony

Here we have two constructors. One that creates a new null Pointer, and another creates a Pointer with space for many instances of the type the Pointer is pointing at. Don't worry if you don't follow everything you are seeing in the above example. The important part is, it should basically look like the class constructor example we saw earlier.

Functions

Like Pony classes, Pony structs can also have functions. Everything you know about functions on Pony classes applies to structs as well.

We'll see structs again

Structs play an important role in Pony's interactions with code written using C. We'll see them again in C-FFI section of the tutorial. We probably won't see too much about structs until then.

Type Aliases

A **type alias** is just a way to give a different name to a type. This may sound a bit silly: after all, types already have names! However, Pony can express some complicated types, and it can be convenient to have a short way to talk about them.

We'll give a couple examples of using type aliases, just to get the feel of them.

Enumerations

One way to use type aliases is to express an enumeration. For example, imagine we want to say something must either be Red, Blue or Green. We could write something like this:

```
primitive Red
primitive Blue
primitive Green

type Colour is (Red | Blue | Green)
```

type-aliases-enumerations.pony

There are two new concepts in there. The first is the type alias, introduced with the keyword `type`. It just means that the name that comes after `type` will be translated by the compiler to the type that comes after `is`.

The second new concept is the type that comes after `is`. It's not a single type! Instead, it's a **union** type. You can read the `|` symbol as **or** in this context, so the type is "Red or Blue or Green".

A union type is a form of *closed world* type. That is, it says every type that can possibly be a member of it. In contrast, object-oriented subtyping is usually *open world*, e.g. in Java, an interface can be implemented by any number of classes.

You can also declare constants like in C or Go like this, making use of `apply`, which can be omitted during call (will be discussed further in Sugar),

```
primitive Red    fun apply(): U32 => 0xFF0000FF
primitive Green  fun apply(): U32 => 0x00FF00FF
primitive Blue   fun apply(): U32 => 0x0000FFFF

type Colour is (Red | Blue | Green)
```

type-aliases-enumerations-apply.pony:15:19

or namespace them like this

```
primitive Colours
  fun red(): U32 => 0xFF0000FF
  fun green(): U32 => 0x00FF00FF
```

type-aliases-enumerations-namespace.pony

You might also want to iterate over the enumeration items like this to print their value for debugging purposes

Complex types

If a type is complicated, it can be nice to give it a mnemonic name. For example, if we want to say that a type must implement more than one interface, we could say:

```
interface HasName
  fun name(): String

interface HasAge
  fun age(): U32

interface HasFeelings
  fun feeling(): String

type Person is (HasName & HasAge & HasFeelings)
```

type-aliases-complex-types-interface.pony:6:15

This use of complex types applies to traits, not just interfaces:

```
trait HasName
  fun name(): String => "Bob"

trait HasAge
  fun age(): U32 => 42

trait HasFeelings
  fun feeling(): String => "Great!"

type Person is (HasName & HasAge & HasFeelings)
```

type-aliases-complex-types-trait.pony:6:15

There's another new concept here: the type has a `&` in it. This is similar to the `|` of a **union** type: it means this is an **intersection** type. That is, it's something that must be *all* of `HasName`, `HasAge` *and* `HasFeelings`.

But the use of `type` here is exactly the same as the enumeration example above, it's just providing a name for a type that is otherwise a bit tedious to type out over and over.

Another example, this time from the standard library, is `SetIs`. Here's the actual definition:

```
type SetIs[A] is HashSet[A, HashIs[A!]]
```

type-aliases-set-is.pony

Again there's something new here. After the name `SetIs` comes the name `A` in square brackets. That's because `SetIs` is a **generic type**. That is, you can give a `SetIs` another type as a parameter, to make specific kinds of set. If you've used Java or C#, this will be pretty familiar. If you've used C++, the equivalent concept is templates, but they work quite differently.

And again the use of `type` just provides a more convenient way to refer to the type we're aliasing:

```
HashSet[A, HashIs[A!]]
```

type-aliases-hash-set.pony

That's another **generic type**. It means a `SetIs` is really a kind of `HashSet`. Another concept has snuck in, which is `!` types. This is a type that is the **alias** of another type. That's tricky stuff that you only need when writing complex generic types, so we'll leave it for later.

One more example, again from the standard library, is the `Map` type that gets used a lot. It's actually a type alias. Here's the real definition of `Map`:

```
type Map[K: (Hashable box & Comparable[K] box), V] is HashMap[K, V, HashEq[K]]
```

type-aliases-map.pony

Unlike our previous example, the first type parameter, `K`, has a type associated with it. This is a **constraint**, which means when you parameterise a `Map`, the type you pass for `K` must be a subtype of the constraint.

Also, notice that `box` appears in the type. This is a **reference capability**. It means there is a certain class of operations we need to be able to do with a `K`. We'll cover this in more detail later.

Just like our other examples, all this really means is that `Map` is really a kind of `HashMap`.

Other stuff

Type aliases get used for a lot of things, but this gives you the general idea. Just remember that a type alias is always a convenience: you could replace every use of the type alias with the full type after the `is`.

In fact, that's exactly what the compiler does.

Type Expressions

The types we've talked about so far can also be combined in **type expressions**. If you're used to object-oriented programming, you may not have seen these before, but they are common in functional programming. A **type expression** is also called an **algebraic data type**.

There are three kinds of type expression: **tuples**, **unions**, and **intersections**.

Tuples

A **tuple** type is a sequence of types. For example, if we wanted something that was a `String` followed by a `U64`, we would write this:

```
var x: (String, U64)
x = ("hi", 3)
x = ("bye", 7)
```

type-expressions-tuple-declaration.pony

All type expressions are written in parentheses, and the elements of a tuple are separated by a comma. We can also destructure a tuple using assignment:

```
(var y, var z) = x
```

type-expressions-tuple-destructuring.pony

Or we can access the elements of a tuple directly:

```
var y = x._1
var z = x._2
```

type-expressions-tuple-direct-access.pony

Note that there's no way to assign to an element of a tuple. Instead, you can just reassign the entire tuple, like this:

```
x = ("wombat", x._2)
```

type-expressions-tuple-reassignment.pony

Why use a tuple instead of a class? Tuples are a way to express a collection of values that doesn't have any associated code or expected behaviour. Basically, if you just need a quick collection of things, maybe to return more than one value from a function, for example, you can use a tuple.

Unions

A **union** type is written like a **tuple**, but it uses a `|` (pronounced "or" when reading the type) instead of a `,` between its elements. Where a tuple represents a collection of values, a union represents a *single* value that can be any of the specified types.

Unions can be used for tons of stuff that require multiple concepts in other languages. For example, optional values, enumerations, marker values, and more.

```
var x: (String | None)
```

type-expressions-union.pony

Here we have an example of using a union to express an optional type, where `x` might be a `String`, but it also might be `None`.

Intersections

An **intersection** uses a `&` (pronounced "and" when reading the type) between its elements. It represents the exact opposite of a union: it is a *single* value that is *all* of the specified types, at the same time!

This can be very useful for combining traits or interfaces, for example. Here's something from the standard library:

```
type Map[K: (Hashable box & Comparable[K] box), V] is HashMap[K, V, HashEq[K]]
```

type-expressions-intersection.pony

That's a fairly complex type alias, but let's look at the constraint of `K`. It's `(Hashable box & Comparable[K] box)`, which means `K` is `Hashable` *and* it is `Comparable[K]`, at the same time.

Combining type expressions

Type expressions can be combined into more complex types. Here's another example from the standard library:

```
var _array: Array[((K, V) | _MapEmpty | _MapDeleted)]
```

type-expressions-combined.pony

Here we have an array where each element is either a tuple of `(K, V)` or a `_MapEmpty` or a `_MapDeleted`.

Because every type expression has parentheses around it, they are actually easy to read once you get the hang of it. However, if you use a complex type expression often, it can be nice to provide a type alias for it.

```
type Number is (Signed | Unsigned | Float)

type Signed is (I8 | I16 | I32 | I64 | I128)

type Unsigned is (U8 | U16 | U32 | U64 | U128)

type Float is (F32 | F64)
```

type-expressions-type-alias.pony

Those are all type aliases used by the standard library.

Is `Number` a type alias for a type expression that contains other type aliases? Yes! Fun, and convenient.

Expressions

This chapter covers the various expressions that make up Pony. From variables to control structures and more.

Literals

What do we want?

Values!

Where do we want them?

In our Pony programs!

Say no more

Every programming language has literals to encode values of certain types, and so does Pony.

In Pony you can express booleans, numeric types, characters, strings and arrays as literals.

Boolean Literals

There is `true`, there is `false`. That's it.

Numeric Literals

Numeric literals can be used to encode any signed or unsigned integer or floating point number.

In most cases Pony is able to infer the concrete type of the literal from the context where it is used (e.g. assignment to a field or local variable or as argument to a method/behaviour call).

It is possible to help the compiler determine the concrete type of the literal using a constructor of one of the numeric types:

- U8, U16, U32, U64, U128, USize, ULong
- I8, I16, I32, I64, I128, ISize, ILong
- F32, F64

```
let my_explicit_unsigned: U32 = 42_000
let my_constructor_unsigned = U8(1)
let my_constructor_float = F64(1.234)
```

literals-numeric-typing.pony

Integer literals can be given as decimal, hexadecimal or binary:

```
    let my_decimal_int: I32 = 1024
    let my_hexadecimal_int: I32 = 0x400
    let my_binary_int: I32 = 0b10000000000
```

literals-number-types.pony:3:5

Floating Point literals are expressed as standard floating point or scientific notation:

```
let my_double_precision_float: F64 = 0.009999999776482582092285156250
let my_scientific_float: F32 = 42.12e-4
```

literals-floats.pony

Character Literals

Character literals are enclosed with single quotes (').

Character literals, unlike String literals, encode to a single numeric value. Usually this is a single byte, a U8. But they can be coerced to any integer type:

```
    let big_a: U8 = 'A'                 // 65
    let hex_escaped_big_a: U8 = '\x41'  // 65
    let newline: U32 = '\n'             // 10
```

literals-character-literals.pony:3:5

The following escape sequences are supported:

- \x4F hex escape sequence with 2 hex digits (up to 0xFF)
- \a, \b, \e, \f, \n, \r, \t, \v, \\, \0, \'

Multibyte Character literals

It is possible to have character literals that contain multiple characters. The resulting integer value is constructed byte by byte with each character representing a single byte in the resulting integer, the last character being the least significant byte:

```
let multiByte: U64 = 'ABCD' // 0x41424344
```

literals-multibyte-character-literals.pony

String Literals

String literals are enclosed with double quotes `"` or triple-quoted `"""`. They can contain any kind of bytes and various escape sequences:

- `\u00FE` Unicode escape sequence with 4 hex digits encoding one code point
- `\u10FFFE` Unicode escape sequence with 6 hex digits encoding one code point
- `\x4F` hex escape sequence for Unicode letters with 2 hex digits (up to `0xFF`)
- `\a`, `\b`, `\e`, `\f`, `\n`, `\r`, `\t`, `\v`, `\\`, `\0`, `\"`

Each escape sequence encodes a full character, not byte.

```
use "format"

actor Main
  new create(env: Env) =>

    let pony = " "
    let pony_hex_escaped = "p\xF6n\xFF"
    let pony_unicode_escape = "\U01F40E"

    env.out.print(pony + " " + pony_hex_escaped + " " + pony_unicode_escape)
    for b in pony.values() do
      env.out.print(Format.int[U8](b, FormatHex))
    end
```

literals-string-literals.pony

All string literals support multi-line strings:

```
let stacked_ponies = "



"
```

literals-multi-line-string-literals.pony

String Literals and Encodings

String Literals contain the bytes that were read from their source code file. Their actual value thus depends on the encoding of their source.

Consider the following example:

```
    let u_umlaut = "ü"
```

literals-string-literals-encoding.pony:3:3

If the file containing this code is encoded as `UTF-8` the byte-value of `u_umlaut` will be: `\xc3\xbc`. If the file is encoded with *ISO-8559-1* (Latin-1) its value will be `\xfc`.

Triple quoted Strings

For embedding multi-line text in string literals, there are triple quoted strings.

```
let triple_quoted_string_docs =
  """
  Triple quoted strings are the way to go for long multi-line text.
  They are extensively used as docstrings which are turned into api documentation.

  They get some special treatment, in order to keep Pony code readable:

  * The string literal starts on the line after the opening triple quote.
  * Common indentation is removed from the string literal
    so it can be conveniently aligned with the enclosing indentation
    e.g. each line of this literal will get its first two whitespaces removed
  * Whitespace after the opening and before the closing triple quote will be
    removed as well. The first line will be completely removed if it only
    contains whitespace. e.g. this strings first character is `T` not `\n`.
  """
```

literals-triple-quoted-string-literals.pony

String Literal Instances

When a single string literal is used several times in your Pony program, all of them will be converted to a single common instance. This means they will always be equal based on identity.

Array Literals

Array literals are enclosed by square brackets. Array literal elements can be any kind of expressions. They are separated by semicolon or newline:

```
let my_literal_array =
  [
    "first"; "second"
    "third one on a new line"
  ]
```

literals-array-literals.pony:3:7

Type inference

If the type of the array is not specified, the resulting type of the literal array expression is `Array[T] ref` where `T` (the type of the elements) is inferred as the union of all the element types:

```
let my_heterogenous_array =
  [
    U64(42)
    "42"
    U64.min_value()
  ]
```

literals-type-inference-union.pony

In the above example the resulting array type will be `Array[(U64|String)] ref` because the array contains `String` and `U64` elements.

If the variable or call argument the array literal is assigned to has a type, the literal is coerced to that type:

```
let my_stringable_array: Array[Stringable] ref =
  [
    U64(0xA)
    "0xA"
  ]
```

literals-type-inference-coercion.pony

Here `my_stringable_array` is coerced to `Array[Stringable] ref`. This works because `Stringable` is a trait that both `String` and `U64` implement.

It is also possible to return an array with a different Reference Capability than `ref` just by specifying it on the type:

```
let my_immutable_array: Array[Stringable] val =
  [
    U64(0xBEEF)
    "0xBEEF"
  ]
```

literals-type-inference-reference-capabilities.pony

This way array literals can be used for creating arrays of any Reference Capability.

As Expression

It is also possible to give the literal a hint on what kind of type it should coerce the array elements to using an `as` Expression. The expression with the desired array element type needs to be added right after the opening square bracket, delimited by a colon:

```
let my_as_array =
  [ as Stringable:
    U64(0xFFEF)
    "0xFFEF"
    U64(1 + 1)
  ]
```

literals-as-expression.pony

This array literal is coerced to be an `Array[Stringable] ref` according to the `as` expression.

If a type is specified on the left-hand side, it needs to exactly match the type in the `as` expression.

Arrays and References

Constructing an array with a literal creates new references to its elements. Thus, to be 100% technically correct, array literal elements are inferred to be the alias of the actual element type. If all elements are of type `T` the array literal will be inferred as `Array[T!] ref` that is as an array of aliases of the type `T`.

It is thus necessary to use elements that can have more than one reference of the same type (e.g. types with `val` or `ref` capability) or use ephemeral types for other capabilities (as returned from constructors or the consume expression).

Variables

Like most other programming languages Pony allows you to store data in variables. There are a few different kinds of variables which have different lifetimes and are used for slightly different purposes.

Local variables

Local variables in Pony work very much as they do in other languages, allowing you to store temporary values while you perform calculations. Local variables live within a chunk of code (they are *local* to that chunk) and are created every time that code chunk executes and disposed of when it completes.

To define a local variable the `var` keyword is used (`let` can also be used, but we'll get to that later). Right after the `var` comes the variable's name, and then you can (optionally) put a `:` followed by the variable's type. For example:

```
var x: String = "Hello"
```

variables-local-variables.pony:1:1

Here, we're assigning the **string literal** `"Hello"` to `x`.

You don't have to give a value to the variable when you define it: you can assign one later if you prefer. If you try to read the value from a variable before you've assigned one, the compiler will complain instead of allowing the dreaded *uninitialised variable* bug.

Every variable has a type, but you don't have to specify it in the declaration if you provide an initial value. The compiler will automatically use the type of the initial value of the variable.

The following definitions of `x`, `y` and `z` are all effectively identical.

```
var x: String = "Hello"

var y = "Hello"

var z: String
z = "Hello"
```

variables-local-variables.pony

Can I miss out both the type and initial value for a variable? No. The compiler will complain that it can't figure out a type for that variable.

All local variable names start with a lowercase letter. If you want to you can end them with a *prime* `'` (or more than one) which is useful when you need a second variable with almost the same meaning as the first. For example, you might have one variable called `time` and another called `time'`.

The chunk of code that a variable lives in is known as its **scope**. Exactly what its scope is depends on where it is defined. For example, the scope of a variable defined within the `then` expression of an `if` statement is that `then` expression. We haven't looked at `if` statements yet, but they're very similar to every other language.

```
if a > b then
  var x = "a is bigger"
  env.out.print(x)  // OK
end
```

```
env.out.print(x)  // Illegal
```

variables-scope.pony:6:11

Variables only exist from when they are defined until the end of the current scope. For our variable `x` this is the `end` at the end of the then expression: after that, it cannot be used.

Var vs. let

Local variables are declared with either a `var` or a `let`. Using `var` means the variable can be assigned and reassigned as many times as you like. Using `let` means the variable can only be assigned once.

```
var x: U32 = 3
let y: U32 = 4
x = 5  // OK
y = 6  // Error, y is let
```

variables-var-vs-let.pony:3:6

Using `let` instead of `var` also means the variable has to be assigned immediately.

```
let x: U32 = 3 // Ok
let y: U32 // Error, can't declare a let local without assigning to it
y = 6 // Error, can't reassign to a let local
```

variables-let-reassignment.pony:3:5

Note that a variable having been declared with `let` only restricts reassignment, and does not influence the mutability of the object it references. This is the job of reference capabilities, explained later in this tutorial.

You never have to declare variables as `let`, but if you know you're never going to change what a variable references then using `let` is a good way to catch errors. It can also serve as a useful comment, indicating what is referenced is not meant to be changed.

Fields

In Pony, fields are variables that live within objects. They work like fields in other object-oriented languages.

Fields have the same lifetime as the object they're in, rather than being scoped. They are set up by the object constructor and disposed of along with the object.

If the name of a field starts with `_`, it's **private**. That means only the type the field is in can have code that reads or writes that field. Otherwise, the field is **public** and can be read or written from anywhere.

Just like local variables, fields can be `var` or `let`. Nevertheless, rules for field assignment differ a bit from variable assignment. No matter the type of the field (either `var` or `let`), either:

1. an initial value has to be assigned in their definition or
2. an initial value has to be assigned in the constructor method.

In the example below, the initial value of the two fields of the class `Wombat` is assigned at the definition level:

```
class Wombat
  let name: String = "Fantastibat"
  var _hunger_level: U32 = 0
```

variables-fields-definition-assignment.pony:6:8

Alternatively, these fields could be assigned in the constructor method:

```
class Wombat
  let name: String
  var _hunger_level: U32

  new create(hunger: U32) =>
    name = "Fantastibat"
    _hunger_level = hunger
```

variables-fields-constructor-assignment.pony:6:12

If the assignment is not done at the definition level or in the constructor, an error is raised by the compiler. This is true for both `var` and `let` fields.

Please note that the assignment of a value to a field has to be explicit. The below example raises an error when compiled, even when the field is of `var` type:

```
class Wombat
  let name: String
  var _hunger_level: U64

  new ref create(name': String, level: U64) =>
    name = name'
    set_hunger_level(level)
    // Error: field _hunger_level left undefined in constructor

  fun ref set_hunger_level(hunger_level: U64) =>
    _hunger_level = hunger_level
```

variables-fields-implicit-assignment.pony

We will see later in the Methods section that a class can have several constructors. For now, just remember that if the assignment of a field is not done at the definition level, it has to be done in each constructor of the class the field belongs to.

As for variables, using `var` means a field can be assigned and reassigned as many times as you like in the class. Using `let` means the field can only be assigned once.

```
class Wombat
  let name: String
  var _hunger_level: U64

  new ref create(name': String, level: U64) =>
    name = name'
    _hunger_level = level

  fun ref set_hunger_level(hunger_level: U64) =>
    _hunger_level = hunger_level // Ok, _hunger_level is of var type
  fun ref set_name(name' : String) =>
    name = name' // Error, can't assign to a let definition more than once
```

variables-fields-let-reassignment.pony:5:17

Can field declarations appear after methods? No. If `var` or `let` keywords appear after a `fun` or `be` declaration, they will be treated as variables within the method body rather than fields within the type declaration. As a result, fields must appear prior to methods in the type declaration

Embedded Fields

Unlike local variables, some types of fields can be declared using `embed`. Specifically, only classes or structs can be embedded - interfaces, traits, primitives and numeric types cannot. A field declared using `embed` is similar to one declared using `let`, but at the implementation level, the memory for the embedded class is laid out directly within the outer class. Contrast this with `let` or `var`, where the implementation uses pointers to reference the field class. Embedded fields can be passed to other functions in exactly the same way as `let` or `var` fields. Embedded fields must be initialised from a constructor expression.

Why would I use `embed`? `embed` avoids a pointer indirection when accessing a field and a separate memory allocation when creating that field. By default, it is advised to use `embed` if possible. However, since an embedded field is allocated alongside its parent object, exterior references to the field forbids garbage collection of the parent, which can result in higher memory usage if a field outlives its parent. Use `let` if this is a concern for you.

Global variables

Some programming languages have **global variables** that can be accessed from anywhere in the code. What a bad idea! Pony doesn't have global variables at all.

Shadowing

Some programming languages let you declare a variable with the same name as an existing variable, and then there are rules about which one you get. This is called *shadowing*, and it's a source of bugs. If you accidentally shadow a variable in Pony, the compiler will complain.

If you need a variable with *nearly* the same name, you can use a prime `'`.

Operators

Infix Operators

Infix operators take two operands and are written between those operands. Arithmetic and comparison operators are the most common:

```
1 + 2
a < b
```

operators-infix-operator.pony

Pony has pretty much the same set of infix operators as other languages.

Operator aliasing

Most infix operators in Pony are actually aliases for functions. The left operand is the receiver the function is called on and the right operand is passed as an argument. For example, the following expressions are equivalent:

```
x + y
x.add(y)
```

operators-operator-aliasing.pony

This means that `+` is not a special symbol that can only be applied to magic types. Any type can provide its own `add` function and the programmer can then use `+` with that type if they want to.

When defining your own `add` function there is no restriction on the types of the parameter or the return type. The right side of the `+` will have to match the parameter type and the whole `+` expression will have the type that `add` returns.

Here's a full example for defining a type which allows the use of `+`. This is all you need:

It is possible to overload infix operators to some degree using union types or f-bounded polymorphism, but this is beyond the scope of this tutorial. See the Pony standard library for further information.

You do not have to worry about any of this if you don't want to. You can simply use the existing infix operators for numbers just like any other language and not provide them for your own types.

The full list of infix operators that are aliases for functions is:

Operator	Method	Description	Note
`+`	`add()`	Addition	
`-`	`sub()`	Subtraction	
`*`	`mul()`	Multiplication	
`/`	`div()`	Division	
`%`	`rem()`	Remainder	
`%%`	`mod()`	Modulo	Starting with version `0.26.1`
`<<`	`shl()`	Left bit shift	
`>>`	`shr()`	Right bit shift	
`and`	`op_and()`	And, both bitwise and logical	
`or`	`op_or()`	Or, both bitwise and logical	
`xor`	`op_xor()`	Xor, both bitwise and logical	
`==`	`eq()`	Equality	
`!=`	`ne()`	Non-equality	
`<`	`lt()`	Less than	
`<=`	`le()`	Less than or equal	
`>=`	`ge()`	Greater than or equal	
`>`	`gt()`	Greater than	
`>~`	`gt_unsafe()`	Unsafe greater than	
`+~`	`add_unsafe()`	Unsafe Addition	
`-~`	`sub_unsafe()`	Unsafe Subtraction	
`*~`	`mul_unsafe()`	Unsafe Multiplication	
`/~`	`div_unsafe()`	Unsafe Division	

Operator	Method	Description	Note
`%~`	`rem_unsafe()`	Unsafe Remainder	
`%%~`	`mod_unsafe()`	Unsafe Modulo	Starting with version `0.26.1`
`<<~`	`shl_unsafe()`	Unsafe left bit shift	
`>>~`	`shr_unsafe()`	Unsafe right bit shift	
`==~`	`eq_unsafe()`	Unsafe equality	
`!=~`	`ne_unsafe()`	Unsafe non-equality	
`<~`	`lt_unsafe()`	Unsafe less than	
`<=~`	`le_unsafe()`	Unsafe less than or equal	
`>=~`	`ge_unsafe()`	Unsafe greater than or equal	
`+?`	`add_partial()?`	Partial Addition	
`-?`	`sub_partial()?`	Partial Subtraction	
`*?`	`mul_partial()?`	Partial Multiplication	
`/?`	`div_partial()?`	Partial Division	
`%?`	`rem_partial()?`	Partial Remainder	
`%%?`	`mod_partial()?`	Partial Modulo	Starting with version `0.26.1`

Short circuiting

The `and` and `or` operators use **short circuiting** when used with boolean variables. This means that the first operand is always evaluated, but the second is only evaluated if it can affect the result.

For `and`, if the first operand is **false** then the second operand is not evaluated since it cannot affect the result.

For `or`, if the first operand is **true** then the second operand is not evaluated since it cannot affect the result.

This is a special feature built into the compiler, it cannot be used with operator aliasing for any other type.

Unary operators

The unary operators are handled in the same manner, but with only one operand. For example, the following expressions are equivalent:

```
-x
x.neg()
```

operators-unary-operators.pony

The full list of unary operators that are aliases for functions is:

Operator	Method	Description
`-`	`neg()`	Arithmetic negation
`not`	`op_not()`	Not, both bitwise and logical
`-~`	`neg_unsafe()`	Unsafe arithmetic negation

Precedence

In Pony, unary operators always bind stronger than any infix operators: `not a == b` will be interpreted as `(not a) == b` instead of `not (a == b)`.

When using infix operators in complex expressions a key question is the **precedence**, i.e. which operator is evaluated first. Given this expression:

```
1 + 2 * 3  // Compilation failed.
```

operators-precedence-without-parentheses.pony:3:3

We will get a value of 9 if we evaluate the addition first and 7 if we evaluate the multiplication first. In mathematics, there are rules about the order in which to evaluate operators and most programming languages follow this approach.

The problem with this is that the programmer has to remember the order and people aren't very good at things like that. Most people will remember to do multiplication before addition, but what about left bit shifting versus bitwise and? Sometimes people misremember (or guess wrong) and that leads to bugs. Worse, those bugs are often very hard to spot.

Pony takes a different approach and outlaws infix precedence. Any expression where more than one infix operator is used **must** use parentheses to remove the ambiguity. If you fail to do this the compiler will complain.

This means that the example above is illegal in Pony and should be rewritten as:

```
1 + (2 * 3)  // 7
```

operators-precedence-with-parentheses.pony

Repeated use of a single operator, however, is fine:

```
1 + 2 + 3  // 6
```

operators-precedence-single-operator.pony

Meanwhile, mixing unary and infix operators do not need additional parentheses as unary operators always bind more closely, so if our example above used a negative three:

```
1 + 2 * -3  // Compilation failed.
```

operators-precedence-infix-and-unary-operators-without-parentheses.pony:3:3

We would still need parentheses to remove the ambiguity for our infix operators like we did above, but not for the unary arithmetic negative (`-`):

```
1 + (2 * -3)  // -5
```

operators-precedence-infix-and-unary-operators-with-parentheses.pony

We can see that it makes more sense for the unary operator to be applied before either infix as it only acts on a single number in the expression so it is never ambiguous.

Unary operators can also be applied to parentheses and act on the result of all operations in those parentheses prior to applying any infix operators outside the parentheses:

```
1 + -(2 * -3)  // 7
```

operators-precedence-unary-operator-with-parentheses.pony

Arithmetic

Arithmetic is about the stuff you learn to do with numbers in primary school: Addition, Subtraction, Multiplication, Division and so on. Piece of cake. We all know that stuff. We nonetheless want to spend a whole section on this topic, because when it comes to computers the devil is in the details.

As introduced in Primitives numeric types in Pony are represented as a special kind of primitive that maps to machine words. Both integer types and floating point types support a rich set of arithmetic and bit-level operations. These are expressed as Infix Operators that are implemented as plain functions on the numeric primitive types.

Pony focuses on two goals, performance and safety. From time to time, these two goals collide. This is true especially for arithmetic on integers and floating point numbers. Safe code should check for overflow, division by zero and other error conditions on each operation where it can happen. Pony tries to enforce as many safety invariants at compile time as it possibly can, but checks on arithmetic operations can only happen at runtime. Code focused on performance should execute integer arithmetic as fast and with as few CPU cycles as possible. Checking for overflow is expensive, doing plain dangerous arithmetic that is possibly subject to overflow is cheap.

Pony provides different ways of doing arithmetic to give programmers the freedom to chose which operation suits best for them, the safe but slower operation or the fast one, because performance is crucial for the use case.

Integers

Pony's default Integer Arithmetic

Doing arithmetic on integer types in Pony with the well known operators like `+`, `-`, `*`, `/` etc. tries to balance the needs for performance and correctness. All default arithmetic operations do not expose any undefined behaviour or error conditions. That means it handles both the cases for overflow/underflow and division by zero. Overflow/Underflow are handled with proper wrap around semantics, using one's complement on signed integers. In that respect we get behaviour like:

```
// unsigned wrap-around on overflow
U32.max_value() + 1 == 0

// signed wrap-around on overflow/underflow
I32.min_value() - 1 == I32.max_value()
```

arithmetic-default-integer-arithmetic.pony

Division by zero is a special case, which affects the division `/` and remainder `%` operators. In Mathematics, division by zero is undefined. In order to avoid either defining division as partial, throwing an error on division by zero or introducing undefined behaviour for that case, the *normal* division is defined to be `0` when the divisor is `0`. This might lead to silent errors, when used without care. Choose Partial and checked Arithmetic to detect division by zero.

In contrast to Unsafe Arithmetic default arithmetic comes with a small runtime overhead because unlike the unsafe variants, it does detect and handle overflow and division by zero.

Operator	Method	Description
`+`	`add()`	wrap around on over-/underflow
`-`	`sub()`	wrap around on over-/underflow
`*`	`mul()`	wrap around on over-/underflow
`/`	`div()`	`x / 0 = 0`
`%`	`rem()`	`x % 0 = 0`
`%%`	`mod()`	`x %% 0 = 0`
`-`	`neg()`	wrap around on over-/underflow
`>>`	`shr()`	filled with zeros, so `x >> 1 == x/2` is true
`<<`	`shl()`	filled with zeros, so `x << 1 == x*2` is true

Unsafe Integer Arithmetic

Unsafe integer arithmetic comes close to what you can expect from integer arithmetic in C. No checks, *raw speed*, possibilities of overflow, underflow or division by zero. Like in C, overflow, underflow and division by zero scenarios are undefined. Don't rely on the results in these cases. It could be anything and is highly platform specific. Division by zero might even crash your program with a `SIGFPE`. Our suggestion is to use these operators only if you can make sure you can exclude these cases.

Here is a list with all unsafe operations defined on Integers:

Operator	Method	Undefined in case of
`+~`	`add_unsafe()`	Overflow E.g. `I32.max_value() +~ I32(1)`
`-~`	`sub_unsafe()`	Overflow
`*~`	`mul_unsafe()`	Overflow.
`/~`	`div_unsafe()`	Division by zero and overflow. E.g. `I32.min_value() / I32(-1)`
`%~`	`rem_unsafe()`	Division by zero and overflow.
`%%~`	`mod_unsafe()`	Division by zero and overflow.
`-~`	`neg_unsafe()`	Overflow. E.g. `-~I32.max_value()`
`>>~`	`shr_unsafe()`	If non-zero bits are shifted out. E.g. `I32(1) >>~ U32(2)`
`<<~`	`shl_unsafe()`	If bits differing from the final sign bit are shifted out.

Unsafe Conversion Converting between integer types in Pony needs to happen explicitly. Each numeric type can be converted explicitly into every other type.

```
// converting an I32 to a 32 bit floating point
I32(12).f32()
```

arithmetic-explicit-numeric-conversion.pony

For each conversion operation there exists an unsafe counterpart, that is much faster when converting from and to floating point numbers. All these unsafe conversion between numeric types are undefined if the target type is smaller than the source type, e.g. if we convert from `I64` to `F32`.

```
// converting an I32 to a 32 bit floating point, the unsafe way
I32(12).f32_unsafe()

// an example for an undefined unsafe conversion
I64.max_value().f32_unsafe()

// an example for an undefined unsafe conversion, that is actually safe
I64(1).u8_unsafe()
```

arithmetic-unsafe-conversion.pony

Here is a full list of all available conversions for numeric types:

Safe conversion	Unsafe conversion
`u8()`	`u8_unsafe()`
`u16()`	`u16_unsafe()`
`u32()`	`u32_unsafe()`
`u64()`	`u64_unsafe()`
`u128()`	`u128_unsafe()`
`ulong()`	`ulong_unsafe()`
`usize()`	`usize_unsafe()`
`i8()`	`i8_unsafe()`
`i16()`	`i16_unsafe()`
`i32()`	`i32_unsafe()`
`i64()`	`i64_unsafe()`
`i128()`	`i128_unsafe()`
`ilong()`	`ilong_unsafe()`
`isize()`	`isize_unsafe()`
`f32()`	`f32_unsafe()`
`f64()`	`f64_unsafe()`

Partial and Checked Arithmetic

If overflow or division by zero are cases that need to be avoided and performance is no critical priority, partial or checked arithmetic offer great safety during runtime. Partial arithmetic operators error on overflow/underflow and division by zero. Checked arithmetic methods return a tuple of the result of the operation and a `Boolean` indicating overflow or other exceptional behaviour.

Partial as well as checked arithmetic comes with the burden of handling exceptions on every case and incurs some performance overhead, be warned.

Partial Operator	Method	Description
`+?`	`add_partial()`	errors on overflow/underflow
`-?`	`sub_partial()`	errors on overflow/underflow
`*?`	`mul_partial()`	errors on overflow/underflow
`/?`	`div_partial()`	errors on overflow/underflow and division by zero
`%?`	`rem_partial()`	errors on overflow/underflow and division by zero

Partial Operator	Method	Description
`%%?`	`mod_partial()`	errors on overflow/underflow and division by zero

Checked arithmetic functions all return the result of the operation and a `Boolean` flag indicating overflow/underflow or division by zero in a tuple.

Checked Method	Description
`addc()`	Checked addition, second tuple element is `true` on overflow/underflow.
`subc()`	Checked subtraction, second tuple element is `true` on overflow/underflow.
`mulc()`	Checked multiplication, second tuple element is `true` on overflow.
`divc()`	Checked division, second tuple element is `true` on overflow or division by zero.
`remc()`	Checked remainder, second tuple element is `true` on overflow or division by zero.
`modc()`	Checked modulo, second tuple element is `true` on overflow or division by zero.
`fldc()`	Checked floored division, second tuple element is `true` on overflow or division by zero.

Floating Point

Pony default arithmetic on floating point numbers (`F32`, `F64`) behave as defined in the floating point standard IEEE 754.

That means e.g. that division by `+0` returns `Inf` and by `-0` returns `-Inf`.

Unsafe Floating Point Arithmetic

Unsafe Floating Point operations do not necessarily comply with IEEE 754 for every input or every result. If any argument to an unsafe operation or its result are `+/-Inf` or `NaN`, the result is actually undefined.

This allows more aggressive optimizations and for faster execution, but only yields valid results for values different that the exceptional values `+/-Inf` and `NaN`. We suggest to only use unsafe arithmetic on floats if you can exclude those cases.

Operator	Method
`+~`	`add_unsafe()`
`-~`	`sub_unsafe()`
`*~`	`mul_unsafe()`
`/~`	`div_unsafe()`
`%~`	`rem_unsafe()`
`%%~`	`mod_unsafe()`
`-~`	`neg_unsafe()`
`<~`	`lt_unsafe()`
`>~`	`gt_unsafe()`
`<=~`	`le_unsafe()`

Operator	Method
`>=~`	`ge_unsafe()`
`=~`	`eq_unsafe()`
`!=~`	`ne_unsafe()`

Additionally `sqrt_unsafe()` is undefined for negative values.

Control Structures

To do real work in a program you have to be able to make decisions, iterate through collections of items and perform actions repeatedly. For this, you need control structures. Pony has control structures that will be familiar to programmers who have used most languages, such as `if`, `while` and `for`, but in Pony, they work slightly differently.

Conditionals

The simplest control structure is the good old `if`. It allows you to perform some action only when a condition is true. In Pony it looks like this:

```
if a > b then
  env.out.print("a is bigger")
end
```

control-structures-conditionals-if.pony:5:7

Can I use integers and pointers for the condition like I can in C? No. In Pony `if` conditions must have type `Bool`, i.e. they are always true or false. If you want to test whether a number `a` is not 0, then you need to explicitly say `a != 0`. This restriction removes a whole category of potential bugs from Pony programs.

If you want some alternative code for when the condition fails just add an `else`:

```
if a > b then
  env.out.print("a is bigger")
else
  env.out.print("a is not bigger")
end
```

control-structures-conditionals-if-else.pony:5:9

Often you want to test more than one condition in one go, giving you more than two possible outcomes. You can nest `if` statements, but this quickly gets ugly:

```
if a == b then
  env.out.print("they are the same")
else
  if a > b then
    env.out.print("a is bigger")
  else
    env.out.print("b bigger")
  end
end
```

control-structures-conditionals-nested-if-else.pony:5:13

As an alternative Pony provides the `elseif` keyword that combines an `else` and an `if`. This works the same as saying `else if` in other languages and you can have as many `elseif`s as you like for each `if`.

```
if a == b then
  env.out.print("they are the same")
elseif a > b then
  env.out.print("a is bigger")
else
  env.out.print("b bigger")
end
```

control-structures-conditionals-if-elseif-else.pony:5:11

Why can't I just say "else if" like I do in C? Why the extra keyword? The relationship between `if` and `else` in C, and other similar languages, is ambiguous. For example:

```
// C code
if(a)
  if(b)
    printf("a and b\n");
else
  printf("not a\n");
```

Here it is not obvious whether the `else` is an alternative to the first or the second `if`. In fact here the `else` relates to the `if(b)` so our example contains a bug. Pony avoids this type of bug by handling `if` and `else` differently and the need for `elseif` comes out of that.

Control structures are expressions

The big difference for control structures between Pony and other languages is that in Pony everything is an expression. In languages like C++ and Java `if` is a statement, not an expression. This means that you can't have an `if` inside an expression, there has to be a separate conditional operator '?'.

In Pony control flow statements like this are all expressions that hand back a value. Your `if` statement hands you back a value. Your `for` loop (which we'll get to a bit later) hands you back a value.

This means you can use `if` directly in a calculation:

```
x = 1 + if lots then 100 else 2 end
```

control-structures-conditionals-expressions.pony:5:5

This will give `x` a value of either 3 or 101, depending on the variable **lots**.

If the `then` and `else` branches of an `if` produce different types then the `if` produces a **union** of the two.

```
var x: (String | Bool) =
  if friendly then
    "Hello"
  else
    false
  end
```

control-structures-conditionals-expression-union-type.pony:4:9

But what if my if doesn't have an else? Any `else` branch that doesn't exist gives an implicit `None`.

```
var x: (String | None) =
  if friendly then
    "Hello"
  end
```

control-structures-conditionals-expression-implicit-none.pony:4:7

The same rules that apply to the value of an `if` expression applies to loops as well. Let's take a look at what a loop value would look like:

```
actor Main
  new create(env: Env) =>
    var x: (String | None) =
      for name in ["Bob"; "Fred"; "Sarah"].values() do
        name
      end
    match x
    | let s: String => env.out.print("x is " + s)
    | None => env.out.print("x is None")
    end
```

control-structures-loop-expression.pony

This will give **x** the value "Sarah" as it is the last name in our list. If our loop has 0 iterations, then the value of its `else` block will be the value of **x**. Or if there is no `else` block, the value will be `None`.

```
actor Main
  new create(env: Env) =>
    var x: (String | None) =
      for name in Array[String].values() do
        name
      end
    match x
    | let s: String => env.out.print("x is " + s)
    | None => env.out.print("x is None")
    end
```

control-structures-loop-expression-none.pony

Here **x** would be `None`.

You can also avoid needing `None` at all by providing a **default value** for when the loop has **0 iterations** by providing an `else` block.

```
actor Main
  new create(env: Env) =>
    var x: String =
      for name in Array[String].values() do
        name
      else
        "no names!"
      end
    env.out.print("x is " + x)
```

control-structures-loop-expression-else.pony

And finally, here the value of `x` is “no names!”

Loops

`if` allows you to choose what to do, but in order to do something more than once, you want a loop.

While

Pony `while` loops are very similar to those in other languages. A condition expression is evaluated and if it’s true we execute the code inside the loop. When we’re done we evaluate the condition again and keep going until it’s false.

Here’s an example that prints out the numbers 1 to 10:

```
var count: U32 = 1

while count <= 10 do
  env.out.print(count.string())
  count = count + 1
end
```

control-structures-loops-while.pony:3:8

Just like `if` expressions, `while` is also an expression. The value returned is just the value of the expression inside the loop the last time we go round it. For this example that will be the value given by `count = count + 1` when count is incremented to 11. Since Pony assignments hand back the *old* value our `while` loop will return 10.

But what if the condition evaluates to false the first time we try, then we don’t go round the loop at all? In Pony `while` expressions can also have an `else` block. In general, Pony `else` blocks provide a value when the expression they are attached to doesn’t. A `while` doesn’t have a value to give if the condition evaluates to false the first time, so the `else` provides it instead.

So is this like an else block on a while loop in Python? No, this is very different. In Python, the `else` is run when the `while` completes. In Pony the `else` is only run when the expression in the `while` isn’t.

Break

Sometimes you want to stop part-way through a loop and give up altogether. Pony has the `break` keyword for this and it is very similar to its counterpart in languages like C++, C#, and Python.

`break` immediately exits from the innermost loop it’s in. Since the loop has to return a value `break` can take an expression. This is optional, and if it’s left out, the value from the `else` block is returned.

Let’s have an example. Suppose you want to go through a list of names, looking for either “Jack” or “Jill”. If neither of those appear, you’ll just take the last name you’re given, and if you’re not given any names at all, you’ll use “Herbert”.

```
var name =
  while moreNames() do
    var name' = getName()
    if (name' == "Jack") or (name' == "Jill") then
      break name'
    end
    name'
  else
    "Herbert"
  end
```

control-structures-loops-while-break-else.pony:10:19

So first we ask if there are any more names to get. If there are then we get a name and see if it's "Jack" or "Jill". If it is we're done and we break out of the loop, handing back the name we've found. If not we try again.

The line `name'` appears at the end of the loop so that will be our value returned from the last iteration if neither "Jack" nor "Jill" is found.

The `else` block provides our value of "Herbert" if there are no names available at all.

Can I break out of multiple, nested loops like the Java labeled break? No, Pony does not support that. If you need to break out of multiple loops you should probably refactor your code or use a worker function.

Continue

Sometimes you want to stop part-way through one loop iteration and move onto the next. Like other languages, Pony uses the `continue` keyword for this.

`continue` stops executing the current iteration of the innermost loop it's in and evaluates the condition ready for the next iteration.

If `continue` is executed during the last iteration of the loop then we have no value to return from the loop. In this case, we use the loop's `else` expression to get a value. As with the `if` expression, if no `else` expression is provided, `None` is returned.

Can I continue an outer, nested loop like the Java labeled continue? No, Pony does not support that. If you need to continue an outer loop you should probably refactor your code.

For

For iterating over a collection of items Pony uses the `for` keyword. This is very similar to `foreach` in C#, `for`..`in` in Python and `for` in Java when used with a collection. It is very different to `for` in C and C++.

The Pony `for` loop iterates over a collection of items using an iterator. On each iteration, round the loop, we ask the iterator if there are any more elements to process, and if there are, we ask it for the next one.

For example, to print out all the strings in an array:

```
for name in ["Bob"; "Fred"; "Sarah"].values() do
  env.out.print(name)
end
```

control-structures-loops-for.pony:3:5

Note the call to `values()` on the array — this is because the loop needs an iterator, not an array.

The iterator does not have to be of any particular type, but needs to provide the following methods:

```
fun has_next(): Bool
fun next(): T?
```

control-structures-iterator-methods.pony

where T is the type of the objects in the collection. You don't need to worry about this unless you're writing your own iterators. To use existing collections, such as those provided in the standard library, you can just use `for` and it will all work. If you do write your own iterators, note that we use structural typing, so your iterator doesn't need to declare that it provides any particular type.

You can think of the above example as being equivalent to:

```
let iterator = ["Bob"; "Fred"; "Sarah"].values()
while iterator.has_next() do
  let name = iterator.next()?
  env.out.print(name)
end
```

control-structures-loops-for-while-comparison.pony:4:8

Note that the variable **name** is declared *let*, so you cannot assign to the control variable within the loop.

Can I use break and continue with for loops? Yes, `for` loops can have `else` expressions attached and can use `break` and `continue` just as for `while`.

Repeat

The final loop construct that Pony provides is `repeat until`. Here we evaluate the expression in the loop and then evaluate a condition expression to see if we're done or we should go round again.

This is similar to `do while` in C++, C# and Java, except that the termination condition is reversed; i.e. those languages terminate the loop when the condition expression is false, but Pony terminates the loop when the condition expression is true.

The differences between `while` and `repeat` in Pony are:

1. We always go around the loop at least once with `repeat`, whereas with `while` we may not go round at all.
2. The termination condition is reversed.

Suppose we're trying to create something and we want to keep trying until it's good enough:

```
actor Main
  new create(env: Env) =>
    var counter = U64(1)
    repeat
      env.out.print("hello!")
      counter = counter + 1
    until counter > 7 end
```

control-structures-loops-repeat.pony

Just like `while` loops, the value given by a `repeat` loop is the value of the expression within the loop on the last iteration, and `break` and `continue` can be used.

Since you always go round a repeat loop at least once, do you ever need to give it an else expression? Yes, you may need to. A `continue` in the last iteration of a `repeat` loop needs to get a value from somewhere, and an `else` expression is used for that.

Match Expressions

If we want to compare an expression to a value then we use an `if`. But if we want to compare an expression to a lot of values this gets very tedious. Pony provides a powerful pattern matching facility, combining matching on values and types, without any special code required.

Matching: the basics

Here's a simple example of a match expression that produces a string.

```
match x
| 2 => "int"
| 2.0 => "float"
| "2" => "string"
else
  "something else"
end
```

match-expression.pony

If you're used to functional languages this should be very familiar.

For those readers more familiar with the C and Java family of languages, think of this like a switch statement. But you can switch on values other than just integers, like Strings. In fact, you can switch on any type that provides a comparison function, including your own classes. And you can also switch on the runtime type of an expression.

A match starts with the keyword `match`, followed by the expression to match, which is known as the match **operand**. In this example, the operand is just the variable `x`, but it can be any expression.

Most of the match expression consists of a series of **cases** that we match against. Each case consists of a pipe symbol ('|'), the **pattern** to match against, an arrow ('=>') and the expression to evaluate if the case matches.

We go through the cases one by one until we find one that matches. (Actually, in practice the compiler is a lot more intelligent than that and uses a combination of sequential checks and jump tables to be as efficient as possible.)

Note that each match case has an expression to evaluate and these are all independent. There is no "fall through" between cases as there is in languages such as C.

If the value produced by the match expression isn't used then the cases can omit the arrow and expression to evaluate. This can be useful for excluding specific cases before a more general case.

Else cases

As with all Pony control structures, the else case for a match expression is used if we have no other value, i.e. if none of our cases match. The else case, if there is one, **must** come at the end of the match, after all of the specific cases.

If the value the match expression results in is used then you need to have an else case, except in cases where the compiler recognizes that the match is exhaustive and that the else case can never actually be reached. If you omit it a default will be added which evaluates to `None`.

The compiler recognizes a match as exhaustive when the union of the types for all patterns that match on type alone is a supertype of the matched expression type. In other words, when your cases cover all possible types for the matched expression, the compiler will not add an implicit `else None` to your match statement.

Matching on values

The simplest match expression just matches on value.

```
  fun f(x: U32): String =>
    match x
    | 1 => "one"
    | 2 => "two"
    | 3 => "three"
    | 5 => "not four"
    else
      "something else"
    end
```

match-values.pony:6:14

For value matching the pattern is simply the value we want to match to, just like a C switch statement. The case with the same value as the operand wins and we use its expression.

The compiler calls the `eq()` function on the operand, passing the pattern as the argument. This means that you can use your own types as match operands and patterns, as long as you define an `eq()` function.

```
class Foo
  var _x: U32

  new create(x: U32) =>
    _x = x

  fun eq(that: Foo): Bool =>
    _x == that._x

actor Main
  new create(env: Env) =>
    None

  fun f(x: Foo): String =>
    match x
    | Foo(1) => "one"
    | Foo(2) => "two"
    | Foo(3) => "three"
    | Foo(5) => "not four"
    else
      "something else"
    end
```

match-custom-eq-operand.pony

Matching on type and value

Matching on value is fine if the match operand and case patterns have all the same type. However, match can cope with multiple different types. Each case pattern is first checked to see if it is the same type as the runtime type of the operand. Only then will the values be compared.

```
  fun f(x: (U32 | String | None)): String =>
    match x
    | None => "none"
    | 2 => "two"
    | 3 => "three"
    | "5" => "not four"
    else
      "something else"
    end
```

match-type-and-value.pony:6:14

In many languages using runtime type information is very expensive and so it is generally avoided whenever possible.

In Pony it's cheap. Really cheap. Pony's "whole program" approach to compilation means the compiler can work out as much as possible at compile time. The runtime cost of each type check is generally a single pointer comparison. Plus of course, any checks which can be fully determined at compile time are. So for upcasts there's no runtime cost at all.

When are case patterns for value matching evaluated? Each case pattern expression that matches the type of the match operand, needs to be evaluated **each time** the `match` expression is evaluated until one case matches (further case patterns are ignored). This can lead to creating lots of objects unintentionally for the sole purpose of checking for equality. If case patterns actually only need to differentiate by type, Captures should be used instead, these boil down to simple type checks at runtime.

At first sight it is easy to confuse a value matching pattern for a type check. Consider the following example:

```
class Foo is Equatable[Foo]

actor Main

  fun f(x: (Foo | None)): String =>
    match x
    | Foo => "foo"
    | None => "bar"
    else
      ""
    end
  new create(env: Env) =>
    f(Foo)
```

match-value-pattern-matching-vs-type-check.pony

Both case patterns actually **do not** check for the match operand `x` being an instance of `Foo` or `None`, but check for equality with the instance created by evaluating the case pattern (each time). `None` is a primitive and thus there is only one instance at all, in which case this value pattern kind of does the expected thing, but not quite. If `None` had a custom `eq` function that would not use identity equality, this could lead to surprising results.

Remember to always use Captures if all you need is to differentiate by type. Only use value matching if you need a full blown equality check, be it for structural equality or identity equality.

Captures

Sometimes you want to be able to match the type, for any value of that type. For this, you use a **capture**. This defines a local variable, valid only within the case, containing the value of the operand. If the operand is not of the specified type then the case doesn't match.

Captures look just like variable declarations within the pattern. Like normal variables, they can be declared as var or let. If you're not going to reassign them within the case expression it is good practice to use let.

```
  fun f(x: (U32 | String | None)): String =>
    match x
    | None => "none"
    | 2 => "two"
    | 3 => "three"
    | let u: U32 => "other integer"
    | let s: String => s
    end
```

match-captures.pony:6:13

Can I omit the type from a capture, like I can from a local variable? Unfortunately no. Since we match on type and value the compiler has to know what type the pattern is, so it can't be inferred.

Implicit matching on capabilities in the context of union types

In union types, when we pattern match on individual classes or traits, we also implicitly pattern match on the corresponding capabilities. In the example provided below, if `_x` has static type `(A iso | B ref | None)` and dynamically matches `A`, then we also know that it must be an `A iso`.

```
class A
  fun ref sendable() =>
    None

class B
  fun ref update() =>
    None

actor Main
  var _x: (A iso | B ref | None)
  new create(env: Env) =>
    _x = None

  be f(a': A iso) =>
    match (_x = None) // type of this expression: (A iso^ | B ref | None)
    | let a: A iso => f(consume a)
    | let b: B ref => b.update()
    end
```

match-capabilities.pony

Note that using a match expression to differentiate solely based on capabilities at runtime is not possible, that is:

```
class A
  fun ref sendable() =>
    None

actor Main
  var _x: (A iso | A ref | None)

  new create(env: Env) =>
    _x = None

  be f() =>
    match (_x = None)
    | let a1: A iso => None
    | let a2: A ref => None
    end
```

match-capabilities-only.pony

does not type check.

Matching tuples

If you want to match on more than one operand at once then you can simply use a tuple. Cases will only match if **all** the tuple elements match.

```
  fun f(x: (String | None), y: U32): String =>
    match (x, y)
    | (None, let u: U32) => "none"
    | (let s: String, 2) => s + " two"
    | (let s: String, 3) => s + " three"
    | (let s: String, let u: U32) => s + " other integer"
    else
      "something else"
    end
```

match-tuples.pony:5:13

Do I have to specify all the elements in a tuple? No, you don't. Any tuple elements in a pattern can be marked as "don't care" by using an underscore ('_'). The first and fourth cases in our example don't actually care about the U32 element, so we can ignore it.

```
fun f(x: (String | None), y: U32): String =>
  match (x, y)
  | (None, _) => "none"
  | (let s: String, 2) => s + " two"
  | (let s: String, 3) => s + " three"
  | (let s: String, _) => s + " other integer"
  else
    "something else"
  end
```

match-tuples-ignore-elements.pony:5:13

Guards

In addition to matching on types and values, each case in a match can also have a guard condition. This is simply an expression, evaluated **after** type and value matching has occurred, that must give the value true for the case to match. If the guard is false then the case doesn't match and we move onto the next in the usual way.

Guards are introduced with the `if` keyword.

A guard expression may use any captured variables from that case, which allows for handling ranges and complex functions.

```
fun f(x: (String | None), y: U32): String =>
  match (x, y)
  | (None, _) => "none"
  | (let s: String, 2) => s + " two"
  | (let s: String, 3) => s + " three"
  | (let s: String, let u: U32) if u > 14 => s + " other big integer"
  | (let s: String, _) => s + " other small integer"
  else
    "something else"
  end
```

match-guards.pony:6:15

As Operator

The `as` operator in Pony has two related uses. First, it provides a safe way to increase the specificity of an object's type. Second, it gives the programmer a way to specify the type of the items in an array literal.

Expressing a different type of an object

In Pony, each object is an instance of a single concrete type, which is the most specific type for that object. But the object can also be held as part of a "wider" abstract type such as an interface, trait, or type union which the concrete type is said to be a subtype of.

`as` (like `match`) allows a program to check the runtime type of an abstract-typed value to see whether or not the object matches a given type which is more specific. If it doesn't match the more specific type, then a runtime `error` is raised. For example:

```
class Cat
  fun pet() =>
```

```
      ...

  type Animal is (Cat | Fish | Snake)

  fun pet(animal: Animal) =>
    try
      // raises error if not a Cat
      let cat: Cat = animal as Cat
      cat.pet()
    end
```

as-operator-more-specific-type.pony

In the above example, within `pet` our current view of `animal` is via the type union `Animal`. To treat `animal` as a cat, we need to do a runtime check that the concrete type of the object instance is `Cat`. If it is, then we can pet it. This example is a little contrived, but hopefully elucidates how `as` can be used to take a type that is a union and get a specific concrete type from it.

Note that the type requested as the `as` argument must exist as a type of the object instance, unlike C casting where one type can be forced to become another type. Coercion from one concrete type to another is not possible using `as`, so one can not do `let value:F64 = F32(1.0) as F64`. `F32` and `F64` are both concrete types and each object can only have a single concrete type. Many concrete types do provide methods that allow you to convert them to another concrete type, for example, `F32(1.0).f64()` to convert an `F32` to an `F64` or `F32(1.0).string()` to convert to a string.

In addition to using `as` with a union of disjoint types, we can also express an intersected type of the object, meaning the object has a type that the alias we have for the object is not directly related to the type we want to express. For example:

```
trait Alive

trait Well

class Person is (Alive & Well)

class LifeSigns
  fun is_all_good(alive: Alive)? =>
    // if the instance 'alive' is also of type 'Well' (such as a Person instance). raises e
    let well: Well = alive as Well
```

as-operator-unrelated-type.pony:1:10

`as` can also be used to get a more specific type of an object from an alias to it that is an interface or a trait. Let's say, for example, that you have a library for doing things with furry, rodent-like creatures. It provides a `Critter` interface which programmers can then use to create specific types of critters.

```
interface Critter
  fun wash(): String
```

as-operator-more-specific-interface.pony:1:2

The programmer uses this library to create a `Wombat` and a `Capybara` class. But the `Capybara` class provides a new method, `swim()`, that is not part of the `Critter` class. The programmer wants to store all of the critters in an array, in order to carry out actions on groups of critters. Now assume that when capybaras finish washing they want to go for a swim. The programmer can accomplish that by using `as` to attempt to use each `Critter` object in the `Array[Critter]` as a `Capybara`. If this fails because the `Critter` is not a `Capybara`, then an error is raised; the program can swallow this error and go on to the next item.

```
interface Critter
  fun wash(): String

class Wombat is Critter
  fun wash(): String => "I'm a clean wombat!"

class Capybara is Critter
  fun wash(): String => "I feel squeaky clean!"
  fun swim(): String => "I'm swimming like a fish!"

actor Main
  new create(env: Env) =>
    let critters = Array[Critter].>push(Wombat).>push(Capybara)
    for critter in critters.values() do
      env.out.print(critter.wash())
      try
        env.out.print((critter as Capybara).swim())
      end
    end
```

as-operator-more-specific-interface.pony

You can do the same with interfaces as well. In the example below, we have an Array of `Any` which is an interface where we want to try wash any entries that conform to the `Critter` interface.

```
actor Main
  new create(env: Env) =>
    let anys = Array[Any ref].>push(Wombat).>push(Capybara)
    for any in anys.values() do
      try
        env.out.print((any as Critter).wash())
      end
    end
```

as-operator-more-specific-interface-with-reference-capability.pony:11:18

Note, All the `as` examples above could be written using a `match` statement where a failure to match results in `error`. For example, our last example written to use `match` would be:

```
actor Main
  new create(env: Env) =>
    let anys = Array[Any ref].>push(Wombat).>push(Capybara)
    for any in anys.values() do
      try
        match any
        | let critter: Critter =>
          env.out.print(critter.wash())
        else
```

```
          error
        end
      end
    end
```

as-operator-match-statement-comparison.pony:11:23

Thinking of the `as` keyword as "an attempt to match that will error if not matched" is a good mental model to have. If you don't care about handling the "not matched" case that causes an error when using `as`, you can rewrite an `as` to use match without an error like:

```
actor Main
  new create(env: Env) =>
    let anys = Array[Any ref].>push(Wombat).>push(Capybara)
    for any in anys.values() do
      match any
      | let critter: Critter =>
        env.out.print(critter.wash())
      end
    end
```

as-operator-match-statement-without-try.pony:11:19

You can learn more about matching on type in the captures section of the match documentation.

Specify the type of items in an array literal

The `as` operator can also be used to tell the compiler what type to use for the items in an array literal. In many cases, the compiler can infer the type, but sometimes it is ambiguous.

For example, in the case of the following program, the method `foo` can take either an `Array[U32] ref` or an `Array[U64] ref` as an argument. If a literal array is passed as an argument to the method and no type is specified then the compiler cannot deduce the correct one because there are two equally valid ones.

```
actor Main
  fun foo(xs: (Array[U32] ref | Array[U64] ref)): Bool =>
    // do something boring here
    true

  new create(env: Env) =>
    foo([as U32: 1; 2; 3])
    // the compiler would complain about this:
    //   foo([1; 2; 3])
```

as-operator-array-literal.pony

The requested type must be a valid type for the items in the array. Since these types are checked at compile time they are guaranteed to work, so there is no need for the programmer to handle an error condition.

Methods

All Pony code that actually does something, rather than defining types etc, appears in named blocks which are referred to as methods. There are three kinds of methods: functions, constructors, and behaviours. All methods are attached to type definitions (e.g. classes) - there are no global functions.

Behaviours are used for handling asynchronous messages sent to actors, which we've seen in the "Types" chapter when we talked about actors.

Can I have some code outside of any methods like I do in Python? No. All Pony code must be within a method.

Functions

Pony functions are quite like functions (or methods) in other languages. They can have 0 or more parameters and 0 or 1 return values. If the return type is omitted then the function will have a return value of `None`.

```
class C
  fun add(x: U32, y: U32): U32 =>
    x + y

  fun nop() =>
    add(1, 2)  // Pointless, we ignore the result
```

methods-functions.pony:6:11

The function parameters (if any) are specified in parentheses after the function name. Functions that don't take any parameters still need to have the parentheses.

Each parameter is given a name and a type. In our example function `add` has 2 parameters, `x` and `y`, both of which are type `U32`. The values passed to a function call (the `1` and `2` in our example) are called arguments and when the call is made they are evaluated and assigned to the parameters. Parameters may not be assigned to within the function - they are effectively declared `let`.

After the parameters comes the return type. If nothing will be returned this is simply omitted.

After the return value, there's a `=>` and then finally the function body. The value returned is simply the value of the function body (remember that everything is an expression), which is simply the value of the last command in the function.

If you want to exit a function early then use the `return` command. If the function has a return type then you need to provide a value to return. If the function does not have a return type then `return` should appear on its own, without a value.

Can I overload functions by argument type? No, you cannot have multiple methods with the same name in the same type.

Constructors

Pony constructors are used to initialise newly created objects, as in many languages. However, unlike many languages, Pony constructors are named so you can have as many as you like, taking whatever parameters you like. By convention, the main constructor of each type (if there is such a thing for any given type) is called `create`.

```
class Foo
  var _x: U32

  new create() =>
```

```
    _x = 0

  new from_int(x: U32) =>
    _x = x
```

methods-constructors.pony:6:13

The purpose of a constructor is to set up the internal state of the object being created. To ensure this is done constructors must initialise all the fields in the object being constructed.

Can I exit a constructor early? Yes. Just then use the `return` command without a value. The object must already be in a legal state to do this.

Calling

As in many other languages, methods in Pony are called by providing the arguments within parentheses after the method name. The parentheses are required even if there are no arguments being passed to the method.

```
class Foo
  fun hello(name: String): String =>
    "hello " + name

  fun f() =>
    let a = hello("Fred")
```

methods-functions-calling.pony

Constructors are usually called "on" a type, by specifying the type that is to be created. To do this just specify the type, followed by a dot, followed by the name of the constructor you want to call.

```
class Foo
  var _x: U32

  new create() =>
    _x = 0

  new from_int(x: U32) =>
    _x = x

class Bar
  fun f() =>
    var a: Foo = Foo.create()
    var b: Foo = Foo.from_int(3)
```

methods-constructors-calling.pony

Functions are always called on an object. Again just specify the object, followed by a dot, followed by the name of the function to call. If the object to call on is omitted then the current object is used (i.e. `this`).

```
class Foo
  var _x: U32

  new create() =>
    _x = 0
  new from_int(x: U32) =>
    _x = x

  fun get(): U32 =>
    _x

class Bar
  fun f() =>
    var a: Foo = Foo.from_int(3)
    var b: U32 = a.get()
    var c: U32 = g(b)

  fun g(x: U32): U32 =>
    x + 1
```

methods-functions-calling-implicit-this.pony

Constructors can also be called on an expression. Here an object is created of the same type as the specified expression - this is equivalent to directly specifying the type.

```
class Foo
  var _x: U32

  new create() =>
    _x = 0

  new from_int(x: U32) =>
    _x = x

class Bar
  fun f() =>
    var a: Foo = Foo.create()
    var b: Foo = a.from_int(3)
```

methods-constructors-calling-on-expression.pony

We can even reuse the variable name in the assignment expression to call the constructor.

```
class Bar
  fun f() =>
    var a: Foo = a.create()
```

methods-constructors-calling-reuse-variable-name.pony

Here we specify that `var a` is type `Foo`, then proceed to use `a` to call the constructor, `create()`, for objects of type `Foo`.

Default arguments

When defining a method you can provide default values for any of the arguments. The caller then has the choice to use the values you have provided or to provide their own. Default argument values are specified with a `=` after the parameter name.

```
class Coord
  var _x: U32
  var _y: U32

  new create(x: U32 = 0, y: U32 = 0) =>
    _x = x
    _y = y

class Bar
  fun f() =>
    var a: Coord = Coord.create()     // Contains (0, 0)
    var b: Coord = Coord.create(3)    // Contains (3, 0)
    var c: Coord = Coord.create(3, 4) // Contains (3, 4)
```

methods-default-arguments.pony

Do I have to provide default values for all of my arguments? No, you can provide defaults for as many, or as few, as you like.

Named arguments

So far, when calling methods we have always given all the arguments in order. This is known as using **positional** arguments. However, we can also specify the arguments in any order by specifying their names. This is known as using **named** arguments.

To call a method using named arguments the `where` keyword is used, followed by the named arguments and their values.

```
class Coord
  var _x: U32
  var _y: U32

  new create(x: U32 = 0, y: U32 = 0) =>
    _x = x
    _y = y

class Bar
  fun f() =>
    var a: Coord = Coord.create(3, 4) // Contains (3, 4)
    var b: Coord = Coord.create(where y = 4, x = 3) // Contains (3, 4)
```

methods-named-arguments.pony

Note how in `b` above, the arguments were given out of order by using `where` followed using the name of the arguments.

Should I specify `where` for each named argument? No. There must be only one `where` in a method call.

Named and positional arguments can be used together in a single call. Just start with the positional arguments you want to specify, then a `where` and finally the named arguments. But be careful, each argument must be specified only once.

Default arguments can also be used in combination with positional and named arguments - just miss out any for which you want to use the default.

```
class Foo
  fun f(a: U32 = 1, b: U32 = 2, c: U32 = 3, d: U32 = 4, e: U32 = 5): U32  =>
    0
  fun g() =>
    f(6, 7 where d = 8)
    // Equivalent to:
    f(6, 7, 3, 8, 5)
```

methods-named-and-positional-arguments-combined.pony

Can I call using positional arguments but miss out the first one? No. If you use positional arguments they must be the first ones in the call.

Chaining

Method chaining allows you to chain calls on an object without requiring the method to return its receiver. The syntax to call a method and chain the receiver is `object.>method()`, which is roughly equivalent to `(object.method() ; object)`. Chaining a method discards its normal return value.

```
primitive Printer
  fun print_two_strings(out: StdStream, s1: String, s2: String) =>
    out.>print(s1).>print(s2)
    // Equivalent to:
    out.print(s1)
    out.print(s2)
    out
```

methods-chaining.pony

Note that the last `.>` in a chain can be a `.` if the return value of the last call matters.

```
interface Factory
  fun add_option(o: Option)
  fun make_object(): Object

primitive Foo
  fun object_wrong(f: Factory, o1: Option, o2: Option): Object =>
    f.>add_option(o1).>add_option(o2).>make_object() // Error! The expression returns a Fact

  fun object_right(f: Factory, o1: Option, o2: Option): Object =>
    f.>add_option(o1).>add_option(o2).make_object() // Works. The expression returns an Obje
```

methods-chaining-return-value.pony

Anonymous methods

Pony has anonymous methods (or Lambdas). They look like this:

```
use "collections"

actor Main
  new create(env: Env) =>
    let list_of_numbers = List[U32].from([1; 2; 3; 4])
    let is_odd = {(n: U32): Bool => (n % 2) == 1}
    for odd_number in list_of_numbers.filter(is_odd).values() do
      env.out.print(odd_number.string())
    end
```

methods-anonymous-methods.pony

They are presented more in-depth in the Object Literals section.

Privacy

In Pony, method names start either with a lower case letter or with an underscore followed by a lowercase letter. Methods with a leading underscore are private. This means they can only be called by code within the same package. Methods without a leading underscore are public and can be called by anyone.

Can I start my method name with 2 (or more) underscores? No. If the first character is an underscore then the second one MUST be a lower case letter.

Precedence

We have talked about precedence of operators before, and in Pony, method calls and field accesses have higher precedence than any operators.

To sum up, in complex expressions,

1. Method calls and field accesses have higher precedence than any operators.
2. Unary operator have higher precedence than infix operators.
3. When mixing infix operators in complex expressions, we must use parentheses to specify any precedence explicitly.

Errors

Pony doesn't feature exceptions as you might be familiar with them from languages like Python, Java, C++ et al. It does, however, provide a simple partial function mechanism to aid in error handling. Partial functions and the `error` keyword used to raise them look similar to exceptions in other languages but have some important semantic differences. Let's take a look at how you work with Pony's error and then how it differs from the exceptions you might be used to.

Raising and handling errors

An error is raised with the command `error`. At any point, the code may decide to declare an `error` has occurred. Code execution halts at that point, and the call chain is unwound until the nearest enclosing error handler is found. This is all checked at compile time so errors cannot cause the whole program to crash.

Error handlers are declared using the `try-else` syntax.

```
try
  callA()
  if not callB() then error end
  callC()
else
  callD()
end
```

errors-try-else.pony:9:15

In the above code `callA()` will always be executed and so will `callB()`. If the result of `callB()` is true then we will proceed to `callC()` in the normal fashion and `callD()` will not then be executed.

However, if `callB()` returns false, then an error will be raised. At this point, execution will stop and the nearest enclosing error handler will be found and executed. In this example that is, our else block and so `callD()` will be executed.

In either case, execution will then carry on with whatever code comes after the `try end`.

Do I have to provide an error handler? No. The `try` block will handle any errors regardless. If you don't provide an error handler then no error handling action will be taken - execution will simply continue after the `try` expression.

If you want to do something that might raise an error, but you don't care if it does you can just put it in a `try` block without an `else`.

```
try
  // Do something that may raise an error
end
```

errors-try-without-else.pony

Is there anything my error handler has to do? No. If you provide an error handler then it must contain some code, but it is entirely up to you what it does.

What's the resulting value of a try block? The result of a `try` block is the value of the last statement in the `try` block, or the value of the last statement in the `else` clause if an error was raised. If an error was raised and there was no `else` clause provided, the result value will be `None`.

Partial functions

Pony does not require that all errors are handled immediately as in our previous examples. Instead, functions can raise errors that are handled by whatever code calls them. These are called partial functions (this is a mathematical term meaning a function that does not have a defined result for all possible inputs, i.e. arguments). Partial functions **must** be marked as such in Pony with a `?`, both in the function signature (after the return type) and at the call site (after the closing parentheses).

For example, a somewhat contrived version of the factorial function that accepts a signed integer will error if given a negative input. It's only partially defined over its valid input type.

```
  fun factorial(x: I32): I32 ? =>
    if x < 0 then error end
    if x == 0 then
      1
    else
      x * factorial(x - 1)?
    end
```

errors-partial-functions.pony:16:22

Everywhere that an error can be generated in Pony (an error command, a call to a partial function, or certain built-in language constructs) must appear within a `try` block or a function that is marked as partial. This is checked at compile time, ensuring that an error cannot escape handling and crash the program.

Prior to Pony 0.16.0, call sites of partial functions were not required to be marked with a `?`. This often led to confusion about the possibilities for control flow when reading code. Having every partial function call site clearly marked makes it very easy for the reader to immediately understand everywhere that a block of code may jump away to the nearest error handler, making the possible control flow paths more obvious and explicit.

Partial constructors and behaviours

Class constructors may also be marked as partial. If a class constructor raises an error then the construction is considered to have failed and the object under construction is discarded without ever being returned to the caller.

When an actor constructor is called the actor is created and a reference to it is returned immediately. However, the constructor code is executed asynchronously at some later time. If an actor constructor were to raise an error it would already be too late to report this to the caller. For this reason, constructors for actors may not be partial.

Behaviours are also executed asynchronously and so cannot be partial for the same reason.

Try-then blocks

In addition to an `else` error handler, a `try` command can have a `then` block. This is executed after the rest of the `try`, whether or not an error is raised or handled. Expanding our example from earlier:

```
try
  callA()
  if not callB() then error end
  callC()
else
  callD()
then
  callE()
end
```

errors-try-then.pony:9:17

The `callE()` will always be executed. If `callB()` returns true then the sequence executed is `callA()`, `callB()`, `callC()`, `callE()`. If `callB()` returns false then the sequence executed is `callA()`, `callB()`, `callD()`, `callE()`.

Do I have to have an else error handler to have a then block? No. You can have a `try-then` block without an `else` if you like.

Will my then block really always be executed, even if I return inside the try? Yes, your `then` expression will **always** be executed when the `try` block is complete. The only way it won't be is if the `try` never completes (due to an infinite loop), the machine is powered off, or the process is killed (and then, maybe).

With blocks

A `with` expression can be used to ensure disposal of an object when it is no longer needed. A common case is a database connection which needs to be closed after use to avoid resource leaks on the server. For example:

```
with obj = SomeObjectThatNeedsDisposing() do
  // use obj
end
```

errors-with-blocks.pony

`obj.dispose()` will be called whether the code inside the `with` block completes successfully or raises an error. To take part in a `with` expression, the object that needs resource clean-up must, therefore, provide a `dispose()` method:

```
class SomeObjectThatNeedsDisposing
  // constructor, other functions

  fun dispose() =>
    // release resources
```

errors-dispose.pony

Multiple objects can be set up for disposal:

```
with obj = SomeObjectThatNeedsDisposing(), other = SomeOtherDisposableObject() do
  // use obj and other
end
```

errors-dispose-multiple.pony

The value of a `with` expression is the value of the last expression in the block.

Language constructs that can raise errors

The only language construct that can raise an error, other than the `error` command or calling a partial method, is the `as` command. This converts the given value to the specified type if it can be. If it can't then an error is raised. This means that the `as` command can only be used inside a try block or a partial method.

Comparison to exceptions in other languages

Pony errors behave very much the same as those in C++, Java, C#, Python, and Ruby. The key difference is that Pony errors do not have a type or instance associated with them. This makes them the same as C++ exceptions would be if a fixed literal was always thrown, e.g. `throw 3;`. This difference simplifies error handling for the programmer and allows for much better runtime error handling performance.

The `else` handler in a `try` expression is just like a `catch(...)` in C++, `catch(Exception e)` in Java or C#, `except:` in Python, or `rescue` in Ruby. Since exceptions do not have types there is no need for handlers to specify types or to have multiple handlers in a single try block.

The `then` block in a `try` expression is just like a `finally` in Java, C#, or Python and `ensure` in Ruby.

If required, error handlers can "reraise" by using the `error` command within the handler.

Equality in Pony

Pony features two forms of equality: by structure and by identity.

Identity equality

Identity equality checks in Pony are done via the `is` keyword. `is` verifies that the two items are the same.

```
if None is None then
  // TRUE!
  // There is only 1 None so the identity is the same
end

let a = Foo("hi")
let b = Foo("hi")

if a is b then
  // NOPE. THIS IS FALSE
end

let c = a
if a is c then
  // YUP! TRUE!
end
```

equality-identity-equality.pony

Structural equality

Structural equality checking in Pony is done via the infix operator `==`. It verifies that two items have the same value. If the identity of the items being compared is the same, then by definition they have the same value.

You can define how structural equality is checked on your object by implementing `fun eq(that: box->Foo): Bool`. Remember, since `==` is an infix operator, `eq` must be defined on the left operand, and the right operand must be of type `Foo`.

```
class Foo
  let _a: String

  new create(a: String) =>
    _a = a

  fun eq(that: box->Foo): Bool =>
    this._a == that._a

actor Main
  new create(e: Env) =>
    let a = Foo("hi")
    let b = Foo("bye")
    let c = Foo("hi")

    if a == b then
      // won't print
      e.out.print("1")
    end

    if a == c then
      // will print
      e.out.print("2")
    end

    if a is c then
      // won't print
      e.out.print("3")
    end
```

equality-structural-equality.pony

If you don't define your own `eq`, you will inherit the default implementation that defines equal by value as being the same as by identity.

```
interface Equatable[A: Equatable[A] #read]
  fun eq(that: box->A): Bool => this is that
  fun ne(that: box->A): Bool => not eq(that)
```

equality-equatable-default-implementation.pony

Primitives and equality

As you might remember from Chapter 2, primitives are the same as classes except for two important differences:

- A primitive has no fields.
- There is only one instance of a user-defined primitive.

This means, that every primitive of a given type, is always structurally equal and equal based on identity. So, for example, None is always None.

```
if None is None then
  // this is always true
end

if None == None then
  // this is also always true
end
```

equality-primitives.pony

Sugar

Pony allows you to omit certain small details from your code and will put them back in for you. This is done to help make your code less cluttered and more readable. Using sugar is entirely optional, you can always write out the full version if you prefer.

Apply

Many Pony classes have a function called `apply` which performs whatever action is most common for that type. Pony allows you to omit the word `apply` and just attempt to do a call directly on the object. So:

```
var foo = Foo.create()
foo()
```

sugar-apply-implicit.pony

becomes:

```
var foo = Foo.create()
foo.apply()
```

sugar-apply-explicit.pony

Any required arguments can be added just like normal method calls.

```
var foo = Foo.create()
foo(x, 37 where crash = false)
```

sugar-apply-with-arguments-implicit.pony

becomes:

```
var foo = Foo.create()
foo.apply(x, 37 where crash = false)
```

sugar-apply-with-arguments-explicit.pony

Do I still need to provide the arguments to apply? Yes, only the `apply` will be added for you, the correct number and type of arguments must be supplied. Default and named arguments can be used as normal.

How do I call a function foo if apply is added? The `apply` sugar is only added when calling an object, not when calling a method. The compiler can tell the difference and only adds the `apply` when appropriate.

Create

To create an object you need to specify the type and call a constructor. Pony allows you to miss out the constructor and will insert a call to `create()` for you. So:

```
var foo = Foo
```

sugar-create-implicit.pony

becomes:

```
var foo = Foo.create()
```

sugar-create-explicit.pony

Normally types are not valid things to appear in expressions, so omitting the constructor call is not ambiguous. Remember that you can easily spot that a name is a type because it will start with a capital letter.

If arguments are needed for `create` these can be provided as if calling the type. Default and named arguments can be used as normal.

```
var foo = Foo(x, 37 where crash = false)
```

sugar-create-with-arguments-implicit.pony

becomes:

```
var foo = Foo.create(x, 37 where crash = false)
```

sugar-create-with-arguments-explicit.pony

What if I want to use a constructor that isn't named create? Then the sugar can't help you and you have to write it out yourself.

If the create I want to call takes no arguments can I still put in the parentheses? No. Calls of the form `Type()` use the combined create-apply sugar (see below). To get `Type.create()` just use `Type`.

Combined create-apply

If a type has a create constructor that takes no arguments then the create and apply sugar can be used together. Just call on the type and calls to create and apply will be added. The call to create will take no arguments and the call to apply will take whatever arguments are supplied.

```
var foo = Foo()
var bar = Bar(x, 37 where crash = false)
```

sugar-create-apply-combined-implicit.pony

becomes:

```
var foo = Foo.create().apply()
var bar = Bar.create().apply(x, 37 where crash = false)
```

sugar-create-apply-combined-explicit.pony

What if the create has default arguments? Do I get the combined create-apply sugar if I want to use the defaults? The combined create-apply sugar can only be used when the `create` constructor has no arguments. If there are default arguments then this sugar cannot be used.

Update

The `update` sugar allows any class to use an assignment to accept data. Many languages allow this for assigning into collections, for example, a simple C array, `a[3] = x;`.

In any assignment where the left-hand side is a function call, Pony will translate this to a call to update, with the value from the right-hand side as an extra argument. So:

```
foo(37) = x
```

sugar-update-implicit.pony:18:18

becomes:

```
foo.update(37 where value = x)
```

sugar-update-explicit.pony:18:18

The value from the right-hand side of the assignment is always passed to a parameter named `value`. Any object can allow this syntax simply by providing an appropriate function `update` with an argument `value`.

Does my update function have to have a single parameter that takes an integer? No, you can define update to take whatever parameters you like, as long as there is one called `value`. The following are all fine:

```
foo1(2, 3) = x
foo2() = x
foo3(37, "Hello", 3.5 where a = 2, b = 3) = x
```

sugar-update-additional-parameters.pony:23:25

Does it matter where `value` appears in my parameter list? Whilst it doesn't strictly matter it is good practice to put `value` as the last parameter. That way all of the others can be specified by position.

See also

- Lambdas (*Sugar for an object with an `apply()` method*)
- Capability constraints (*Sugar for reference capability combinations in the context of generic types*)
- Default reference capabilities (*Sugar for implicit default values in the context of generic types*)

Object Literals

Sometimes it's really convenient to be able to write a whole object inline. In Pony, this is called an object literal, and it does pretty much exactly what an object literal in JavaScript does: it creates an object that you can use immediately.

But Pony is statically typed, so an object literal also creates an anonymous type that the object literal fulfills. This is similar to anonymous classes in Java and C#. In Pony, an anonymous type can provide any number of traits and interfaces.

What's this look like, then?

It basically looks like any other type definition, but with some small differences. Here's a simple one:

```
object
  fun apply(): String => "hi"
end
```

object-literals-object-literal.pony:4:6

Ok, that's pretty trivial. Let's extend it so that it explicitly provides an interface so that the compiler will make sure the anonymous type fulfills that interface. You can use the same notation to provide traits as well.

```
object is Hashable
  fun apply(): String => "hi"
  fun hash(): USize => this().hash()
end
```

object-literals-object-literal-with-interface.pony:6:9

What we can't do is specify constructors in an object literal, because the literal *is* the constructor. So how do we assign to fields? Well, we just assign to them. For example:

```
use "collections"

class Foo
  fun foo(str: String): Hashable =>
    object is Hashable
      let s: String = str
      fun apply(): String => s
      fun hash(): USize => s.hash()
    end
```

object-literals-fields-assignment.pony:1:9

When we assign to a field in the constructor, we are *capturing* from the lexical scope the object literal is in. Pretty fun stuff! It lets us have arbitrarily complex **closures** that can even have multiple entry points (i.e. functions you can call on a closure).

An object literal with fields is returned as a `ref` by default unless an explicit reference capability is declared by specifying the capability after the `object` keyword. For example, an object with sendable captured references can be declared as `iso` if needed:

```
use "collections"

class Foo
  fun foo(str: String): Hashable iso^ =>
    object iso is Hashable
      let s: String = str
      fun apply(): String => s
      fun hash(): USize => s.hash()
    end
```

object-literals-reference-capability.pony:1:9

We can also implicitly capture values from the lexical scope by using them in the object literal. Sometimes values that aren't local variables, aren't fields, and aren't parameters of a function are called *free variables*. By using them in a function, we are *closing over* them - that is, capturing them. The code above could be written without the field `s`:

```
use "collections"

class Foo
  fun foo(str: String): Hashable iso^ =>
    object iso is Hashable
      fun apply(): String => str
      fun hash(): USize => str.hash()
    end
```

object-literals-closing-over-values.pony:1:8

Lambdas

Arbitrarily complex closures are nice, but sometimes we just want a simple closure. In Pony, you can use the lambdas for that. A lambda is written as a function (implicitly named `apply`) enclosed in curly brackets:

```
{(s: String): String => "lambda: " + s }
```

object-literals-lambda.pony:4:4

This produces the same code as:

```
object
  fun apply(s: String): String => "lambda: " + s
end
```

object-literals-lambda-as-explicit-object-literal.pony:4:6

The reference capability of the lambda object can be declared by appending it after the closing curly bracket:

```
{(s: String): String => "lambda: " + s } iso
```

object-literals-lambda-with-reference-capability.pony:4:4

This produces the same code as:

```
object iso
  fun apply(s: String): String => "lambda: " + s
end
```

object-literals-lambda-with-reference-capability-as-explicit-object-literal.pony:4:6

Lambdas can be used to capture from the lexical scope in the same way as object literals can assign from the lexical scope to a field. This is done by adding a second argument list after the parameters:

```
class Foo
  new create(env: Env) =>
    foo({(s: String)(env) => env.out.print(s) })

  fun foo(f: {(String)}) =>
    f("Hello World")
```

object-literals-lambda-capture-values.pony:5:10

It's also possible to use a *capture list* to create new names for things. A capture list is a second parenthesised list after the parameters:

```
  new create(env: Env) =>
    foo({(s: String)(myenv = env) => myenv.out.print(s) })
```

object-literals-lambda-capture-and-rename-values.pony:6:7

The type of a lambda is also declared using curly brackets. Within the brackets, the function parameter types are specified within parentheses followed by an optional colon and return type. The example above uses `{(String)}` to be the type of a lambda function that takes a `String` as an argument and returns nothing.

If the lambda object is not declared with a specific reference capability, the reference capability is inferred from the structure of the lambda. If the lambda does not have any captured references, it will be `val` by default; if it does have captured references, it will be `ref` by default. The following is an example of a `val` lambda object:

```
use "collections"

actor Main
  new create(env: Env) =>
    let l = List[U32]
    l.>push(10).>push(20).>push(30).push(40)
    let r = reduce(l, 0, {(a:U32, b:U32): U32 => a + b })
    env.out.print("Result: " + r.string())

  fun reduce(l: List[U32], acc: U32, f: {(U32, U32): U32} val): U32 =>
    try
      let acc' = f(acc, l.shift()?)
      reduce(l, acc', f)
    else
      acc
    end
```

object-literals-lambda-reference-capabilities.pony

The `reduce` method in this example requires the lambda type for the `f` parameter to require a reference capability of `val`. The lambda object passed in as an argument does not need to declare an explicit reference capability because `val` is the default for a lambda that does not capture anything.

As mentioned previously the lambda desugars to an object literal with an `apply` method. The reference capability for the `apply` method defaults to `box` like any other method. In a lambda that captures references, this needs to be `ref` if the function needs to modify any of the captured variables or call `ref` methods on them. The reference capability for the method (versus the reference capability for the object which was described above) is defined by putting the capability before the parenthesized argument list.

```
use "collections"

actor Main
  new create(env: Env) =>
    let l = List[String]
    l.>push("hello").push("world")
    var count = U32(0)
    for_each(l, {ref(s:String) =>
      env.out.print(s)
      count = count + 1
    })
    // Displays '0' as the count
    env.out.print("Count: " + count.string())

  fun for_each(l: List[String], f: {ref(String)} ref) =>
    try
      f(l.shift()?)
      for_each(l, f)
    end
```

object-literals-lambda-reference-capabilities-2.pony

This example declares the type of the apply function that is generated by the lambda expression as being `ref`. The lambda type declaration for the `f` parameter in the `for_each` method also declares it as `ref`. The reference capability of the lambda type must also be `ref` so that the method can be called. The lambda object does not need to declare an explicit reference capability because `ref` is the default for a lambda that has captures.

The above example also notes a subtle reality of captured references. At first glance one might expect `count` to have been incremented by the application of `f`. However, reassigning a reference, `count = count + 1`, inside a lambda or object literal can never cause a reassignment in the outer scope. If `count` were an object with reference capabilities permitting mutation, the captured reference could be modified with for example `count.increment()`. The resulting mutation would be visible to any location holding a reference to the same object as `count`.

Actor literals

Normally, an object literal is an instance of an anonymous class. To make it an instance of an anonymous actor, just include one or more behaviours in the object literal definition.

```
object
  be apply() => env.out.print("hi")
end
```

object-literals-actor-literal.pony:4:6

An actor literal is always returned as a `tag`.

Primitive literals

When an anonymous type has no fields and no behaviours (like, for example, an object literal declared as a lambda literal), the compiler generates it as an anonymous primitive, unless a non-`val` reference capability is explicitly given. This means no memory allocation is needed to generate an instance of that type.

In other words, in Pony, a lambda that doesn't close over anything has no memory allocation overhead. Nice.

A primitive literal is always returned as a `val`.

Partial Application

Partial application lets us supply *some* of the arguments to a constructor, function, or behaviour, and get back something that lets us supply the rest of the arguments later.

A simple case

A simple case is to create a "callback" function. For example:

```
class Foo
  var _f: F64 = 0

  fun ref addmul(add: F64, mul: F64): F64 =>
    _f = (_f + add) * mul

class Bar
  fun apply() =>
    let foo: Foo = Foo
    let f = foo~addmul(3)
    f(4)
```

partial-application-callback-function-with-some-arguments-bound.pony

This is a bit of a silly example, but hopefully, the idea is clear. We partially apply the `addmul` function on `foo`, binding the receiver to `foo` and the `add` argument to `3`. We get back an object, `f`, that has an `apply` method that takes a `mul` argument. When it's called, it in turn calls `foo.addmul(3, mul)`.

We can also bind all the arguments:

```
let f = foo~addmul(3, 4)
f()
```

partial-application-callback-function-with-all-arguments-bound.pony

Or even none of the arguments:

```
let f = foo~addmul()
f(3, 4)
```

partial-application-callback-function-with-no-arguments-bound.pony

Out of order arguments

Partial application with named arguments allows binding arguments in any order, not just left to right. For example:

```
let f = foo~addmul(where mul = 4)
f(3)
```

partial-application-callback-function-with-out-of-order-arguments.pony

Here, we bound the `mul` argument but left `add` unbound.

Partial application is just a lambda

Under the hood, we're assembling an object literal for partial application, just as if you had written a lambda yourself. It captures aliases of some of the lexical scope as fields and has an `apply` function that takes some, possibly reduced, number of arguments. This is actually done as sugar, by rewriting the abstract syntax tree for partial application to be an object literal, before code generation.

That means partial application results in an anonymous class and returns a `ref`. If you need another reference capability, you can wrap partial application in a `recover` expression. It also means that we can't consume unique fields for a lambda, as the apply method might be called many times.

Partially applying a partial application

Since partial application results in an object with an apply method, we can partially apply the result!

```
let f = foo~addmul()
let f2 = f~apply(where mul = 4)
f2(3)
```

partial-application-partially-applying-a-partial-application.pony

Reference Capabilities

We've covered the basics of Pony's type system and then expressions, this chapter about reference capabilities will cover another feature of Pony's type system. There aren't currently any mainstream programming languages that feature reference capabilities. What is a reference capability?

Well, a reference capability is built on the idea of "a capability". A capability is the ability to do "something". Usually that "something" involves an external resource that you might want access to; like the file system or the network. This usage of capability is called an object capability and is discussed in the next chapter.

Pony also features a different kind of capability, called a "reference capability". Where object capabilities are about being granted the ability to do things with objects, reference capabilities are about denying you the ability to do things with memory references. For example, "you can have access to this memory BUT ONLY for reading it. You can not write to it". That's a reference capability and it's denying you access to do things.

Reference capabilities are core to what makes Pony special. You might remember in the introduction to this tutorial what we said about Pony:

- It's type safe. Really type safe. There's a mathematical proof and everything.
- It's memory safe. Ok, this comes with type safe, but it's still interesting. There are no dangling pointers, no buffer overruns, heck, the language doesn't even have the concept of *null*!
- It's exception safe. There are no runtime exceptions. All exceptions have defined semantics, and they are *always* handled.
- It's data-race-free. Pony doesn't have locks or atomic operations or anything like that. Instead, the type system ensures *at compile time* that your concurrent program can never have data races. So you can write highly concurrent code and never get it wrong.
- It's deadlock free. This one is easy because Pony has no locks at all! So they definitely don't deadlock, because they don't exist.

Reference capabilities are what make all that awesome possible.

Code examples in this chapter might be kind of sparse, because we're largely dealing with higher-level concepts. Try to read through the chapter at least once before starting to put the ideas into practice. By the time you finish this chapter, you should start to have a handle on what reference capabilities are and how you can use them. Don't worry if you struggle with them at first. For most people, it's a new way of thinking about your code and takes a while to grasp. If you get stuck trying to get your capabilities right, definitely reach out for help. Once you've used them for a couple weeks, problems with capabilities start to melt away, but before that can be a real struggle. Don't worry, we all went through that struggle. In fact, there's a section of the Pony website dedicated to resources that can help in learning reference capabilities. And by all means, reach out to the Pony community for help. We are here to help you get over the reference capabilities learning curve. It's not easy. We know that. It's a new way of thinking for folks, so do please reach out. We're waiting to hear from you.

Scared? Don't be. Ready? Good. Let's get started.

Reference Capabilities

So if the object *is* the capability, what controls what we can do with the object? How do we express our *access rights* on that object?

In Pony, we do it with *reference capabilities*.

Rights are part of a capability

If you open a file in UNIX and get a file descriptor back, that file descriptor is a token that designates an object - but it isn't a capability. To be a capability, we need to open that file with some permission - some access right. For example:

```
int fd = open("/etc/passwd", O_RDWR);
```

Now we have a token and a set of rights.

In Pony, every reference has both a type and a reference capability. In fact, the reference capability is *part* of its type. These allow you to specify which of your objects can be shared with other actors and allow the compiler to check that what you're doing is concurrency safe.

Basic concepts

There are a few simple concepts you need to understand before reference capabilities will make any sense. We've talked about some of these already, and some may already be obvious to you, but it's worth recapping here.

Shared mutable data is hard

The problem with concurrency is shared mutable data. If two different threads have access to the same piece of data then they might try to update it at the same time. At best this can lead to the two threads having different versions of the data. At worst the updates can interact badly resulting in the data being overwritten with garbage. The standard way to avoid these problems is to use locks to prevent data updates from happening at the same time. This causes big performance hits and is very difficult to get right, so it causes lots of bugs.

Immutable data can be safely shared

Any data that is immutable (i.e. it cannot be changed) is safe to use concurrently. Since it is immutable it is never updated and it's the updates that cause concurrency problems.

Isolated data is safe

If a block of data has only one reference to it then we call it **isolated**. Since there is only one reference to it, isolated data cannot be **shared** by multiple threads, so there are no concurrency problems. Isolated data can be passed between multiple threads. As long as only one of them has a reference to it at a time then the data is still safe from concurrency problems.

Isolated data may be complex

An isolated piece of data may be a single byte. But it can also be a large data structure with multiple references between the various objects in that structure. What matters for the data to be isolated is that there is only a single reference to that structure as a whole. We talk about the **isolation boundary** of a data structure. For the structure to be isolated:

1. There must only be a single reference outside the boundary that points to an object inside.
2. There can be any number of references inside the boundary, but none of them must point to an object outside.

Every actor is single threaded

The code within a single actor is never run concurrently. This means that, within a single actor, data updates cannot cause problems. It's only when we want to share data between actors that we have problems.

OK, safely sharing data concurrently is tricky. How do reference capabilities help?

By sharing only immutable data and exchanging only isolated data we can have safe concurrent programs without locks. The problem is that it's very difficult to do that correctly. If you accidentally hang on to a reference to some isolated data you've handed over or change something you've shared as immutable then everything goes wrong. What you need is for the compiler to force you to live up to your promises. Pony reference capabilities allow the compiler to do just that.

Type qualifiers

If you've used C/C++, you may be familiar with `const`, which is a *type qualifier* that tells the compiler not to allow the programmer to *mutate* something.

A reference capability is a form of *type qualifier* and provides a lot more guarantees than `const` does!

In Pony, every use of a type has a reference capability. These capabilities apply to variables, rather than to the type as a whole. In other words, when you define a class `Wombat`, you don't pick a reference capability for all instances of the class. Instead, `Wombat` variables each have their own reference capability.

As an example, in some languages, you have to define a type that represents a mutable `String` and another type that represents an immutable `String`. For example, in Java, there is a `String` and a `StringBuilder`. In Pony, you can define a single class `String` and have some variables that are `String ref` (which are mutable) and other variables that are `String val` (which are immutable).

The list of reference capabilities

There are six reference capabilities in Pony and they all have strict definitions and rules on how they can be used. We'll get to all of that later, but for now here are their names and what you use them for:

Isolated, written `iso`. This is for references to isolated data structures. If you have an `iso` variable then you know that there are no other variables that can access that data. So you can change it however you like and give it to another actor.

Value, written `val`. This is for references to immutable data structures. If you have a `val` variable then you know that no-one can change the data. So you can read it and share it with other actors.

Reference, written `ref`. This is for references to mutable data structures that are not isolated, in other words, "normal" data. If you have a `ref` variable then you can read and write the data however you like and you can have multiple variables that can access the same data. But you can't share it with other actors.

Box. This is for references to data that is read-only to you. That data might be immutable and shared with other actors or there may be other variables using it in your actor that can change the data. Either way, the `box` variable can be used to safely read the data. This may sound a little pointless, but it allows you to write code that can work for both `val` and `ref` variables, as long as it doesn't write to the object.

Transition, written `trn`. This is used for data structures that you want to write to, while also holding read-only (`box`) variables for them. You can also convert the `trn` variable to a `val` variable later if you wish, which stops anyone from changing the data and allows it be shared with other actors.

Tag. This is for references used only for identification. You cannot read or write data using a `tag` variable. But you can store and compare `tag`s to check object identity and share `tag` variables with other actors.

Note that if you have a variable referring to an actor then you can send messages to that actor regardless of what reference capability that variable has.

How to write a reference capability

A reference capability comes at the end of a type. So, for example:

```
String iso // An isolated string
String trn // A transition string
String ref // A string reference
String val // A string value
String box // A string box
String tag // A string tag
```

reference-capabilities-string-capabilities.pony

What does it mean when a type doesn't specify a reference capability? It means you are using the *default* reference capability for that type, which is defined along with the type. Here's an example from the standard library:

```
class val String
```

reference-capabilities-string-default.pony

When we use a String we usually mean an immutable string value, so we make `val` the default reference capability for `String` (but not necessarily for `String` constructors, see below). For example, when we don't specify the capability in the following code, the compiler understands that we are using the default reference capability `val` specified in the type definition:

```
let a: String val = "Hello, world!"
let b: String = "I'm a wombat!" // Also a String val
```

reference-capabilities-default-vs-explicit.pony

So do I have to specify a reference capability when I define a type? Only if you want to. There are sensible defaults that most types will use. These are `ref` for classes, `val` for primitives (i.e. immutable references), and `tag` for actors.

How to create objects with different capabilities

When you write a constructor, by default, that constructor will either create a new object with `ref` or `tag` as the capability. In the case of actors, the constructor will always create a `tag`. For classes, it defaults to `ref` but you can create with other capabilities. Let's take a look at an example:

```
class Foo
  let x: U32

  new val create(x': U32) =>
    x = x'
```

reference-capability-specificy-a-capability-other-than-the-default.pony

Now when you call `Foo.create(1)`, you'll get a `Foo val` instead of `Foo ref`. But what if you want to create both `val` and `ref` `Foo`s? You could do something like this:

```
class Foo
  let x: U32

  new val create_val(x': U32) =>
    x = x'
  new ref create_ref(x': U32) =>
    x = x'
```

reference-capabilities-constructors-for-different-capabilities.pony

But, that's probably not what you'd really want to do. Better to use the capabilities recovery facilities of Pony that we'll cover later in the Recovering Capabilities section.

Reference Capability Guarantees

Since types are guarantees, it's useful to talk about what guarantees a reference capability makes.

What is denied

We're going to talk about reference capability guarantees in terms of what's *denied*. By this, we mean: what can other variables *not* do when you have a variable with a certain reference capability?

We need to distinguish between the actor that contains the variable in question and *other* actors.

This is important because data reads and writes from other actors may occur concurrently. If two actors can both read the same data and one of them changes it then it will change under the feet of the other actor. This leads to data-races and the need for locks. By ensuring this situation can never occur, Pony eliminates the need for locks.

All code within any one actor always executes sequentially. This means that data accesses from multiple variables within a single actor do not suffer from data-races.

Mutable reference capabilities

The **mutable** reference capabilities are `iso`, `trn` and `ref`. These reference capabilities are **mutable** because they can be used to both read from and write to an object.

- If an actor has an `iso` variable, no other variable can be used by *any* actor to read from or write to that object. This means an `iso` variable is the only variable anywhere in the program that can read from or write to that object. It is *read and write unique.*
- If an actor has a `trn` variable, no other variable can be used by *any* actor to write to that object, and no other variable can be used by *other* actors to read from or write to that object. This means a `trn` variable is the only variable anywhere in the program that can write to that object, but other variables held by the same actor may be able to read from it. It is *write unique.*
- If an actor has a `ref` variable, no other variable can be used by *other* actors to read from or write to that object. This means that other variables can be used to read from and write to the object, but only from within the same actor.

Why can they be used to write? Because they all stop *other* actors from reading from or writing to the object. Since we know no other actor will be reading, it's safe for us to write to the object, without having to worry about data-races. And since we know no other actor will be writing, it's safe for us to read from the object, too.

Immutable reference capabilities

The **immutable** reference capabilities are `val` and `box`. These reference capabilities are **immutable** because they can be used to read from an object, but not to write to it.

- If an actor has a `val` variable, no other variable can be used by *any* actor to write to that object. This means that the object can't *ever* change. It is *globally immutable*.
- If an actor has a `box` variable, no other variable can be used by *other* actors to write to that object. This means that other actors may be able to read the object and other variables in the same actor may be able to write to it (although not both). In either case, it is safe for us to read. The object is *locally immutable*.

Why can they be used to read but not write? Because these reference capabilities only stop *other* actors from writing to the object. That means there is no guarantee that *other* actors aren't reading from the object, which means it's not safe for us to write to it. It's safe for more than one actor to read from an object at the same time though, so we're allowed to do that.

Opaque reference capabilities

There's only one **opaque** reference capability, which is `tag`. A `tag` variable makes no guarantees about other variables at all. As a result, it can't be used to either read from or write to the object; hence the name **opaque**.

It's still useful though: you can do identity comparison with it, you can call behaviours on it, and you can call functions on it that only need a `tag` receiver.

Why can't `tag` be used to read or write? Because `tag` doesn't stop *other* actors from writing to the object. That means if we tried to read, we would have no guarantee that there wasn't some other actor writing to the object, so we might get a race condition.

Consume and Destructive Read

An important part of Pony's capabilities is being able to say "I'm done with this thing." We'll cover two means of handling this situation: consuming a variable and destructive reads.

Consuming a variable

Sometimes, you want to *move* an object from one variable to another. In other words, you don't want to make a *second* name for the object, you want to move the object from some existing name to a different one.

You can do this by using `consume`. When you `consume` a variable you take the value out of it, effectively leaving the variable empty. No code can read from that variable again until a new value is written to it. Consuming a local variable or a parameter allows you to move it to a new location, most importantly for `iso` and `trn`.

```
fun test(a: Wombat iso) =>
  var b: Wombat iso = consume a // Allowed!
```

consume-and-destructive-read-consuming-a-variable.pony:5:6

The compiler is happy with that because by consuming `a`, you've said the value can't be used again and the compiler will complain if you try to.

```
fun test(a: Wombat iso) =>
  var b: Wombat iso = consume a // Allowed!
  var c: Wombat tag = a // Not allowed!
```

consume-and-destructive-read-consuming-a-variable-failure.pony:5:7

Here's an example of that. When you try to assign `a` to `c`, the compiler will complain.

By default, a `consume` expression returns a type with the capability of the variable that you are assigning to. You can see this in the example above, where we say that `b` is `Wombat iso`, and as such the result of the `consume` expression is `Wombat iso`. We could also have said that `b` is a `Wombat val`, but we can instead give an explicit reference capability to the `consume` expression:

```
fun test(a: AnIncrediblyLongTypeName iso) =>
  var b = consume val a
```

consume-and-destructive-read-consuming-a-variable-and-change-its-reference-capability.pony

The expression in line 2 of the example above is equivalent to saying `var b: AnIncrediblyLongTypeName val = consume a`.

Can I `consume` a field? Definitely not! Consuming something means it is empty, that is, it has no value. There's no way to be sure no other alias to the object will access that field. If we tried to access a field that was empty, we would crash. But there's a way to do what you want to do: *destructive read*.

Destructive read

There's another way to *move* a value from one name to another. Earlier, we talked about how assignment in Pony returns the *old* value of the left-hand side, rather than the new value. This is called *destructive read*, and we can use it to do what we want to do, even with fields.

```
class Aardvark
  var buddy: Wombat iso

  new create() =>
    buddy = recover Wombat end

  fun ref test(a: Wombat iso) =>
    var b: Wombat iso = buddy = consume a // Allowed!
```

consume-and-destructive-read-moving-a-value.pony

Here, we consume `a`, assign it to the field `buddy`, and assign the *old* value of `buddy` to `b`.

Why is it ok to destructively read fields when we can't consume them? Because when we do a destructive read, we assign to the field so it always has a value. Unlike `consume`, there's no time when the field is empty. That means it's safe and the compiler doesn't complain.

Recovering Capabilities

A `recover` expression lets you "lift" the reference capability of the result. A mutable reference capability (`iso`, `trn`, or `ref`) can become *any* reference capability, and an immutable reference capability (`val` or `box`) can become any immutable or opaque reference capability.

Why is this useful?

This most straightforward use of `recover` is to get an `iso` that you can pass to another actor. But it can be used for many other things as well, such as:

- Creating a cyclic immutable data structure. That is, you can create a complex mutable data structure inside a `recover` expression, "lift" the resulting `ref` to a `val`.

- “Borrow” an `iso` as a `ref`, do a series of complex mutable operations on it, and return it as an `iso` again.
- “Extract” a mutable field from an `iso` and return it as an `iso`.

What does this look like?

The `recover` expression wraps a list of expressions and is terminated by an `end`, like this:

```
recover Array[String].create() end
```

recovering-capabilities-ref-to-iso.pony

This expression returns an `Array[String] iso`, instead of the usual `Array[String] ref` you would get. The reason it is `iso` and not any of the other mutable reference capabilities is because there is a default reference capability when you don’t specify one. The default for any mutable reference capability is `iso` and the default for any immutable reference capability is `val`.

Here’s a more complicated example from the standard library:

```
recover
  var s = String((prec + 1).max(width.max(31)))
  var value = x

  try
    if value == 0 then
      s.push(table(0)?)
    else
      while value != 0 do
        let index = ((value = value / base) - (value * base))
        s.push(table(index.usize())?)
      end
    end
  end

  _extend_digits(s, prec')
  s.append(typestring)
  s.append(prestring)
  _pad(s, width, align, fill)
  s
end
```

recovering-capabilities-format-int.pony

That’s from `format/_FormatInt`. It creates a `String ref`, does a bunch of stuff with it, and finally returns it as a `String iso`.

You can also give an explicit reference capability:

```
    let key = recover val line.substring(0, i).>strip() end
```

recovering-capabilities-with-explicit-reference-capability.pony:5:5

That’s from `net/http/_PayloadBuilder`. We get a substring of `line`, which is a `String iso^`, then we call strip on it, which returns itself. But since strip is a `ref` function, it returns itself as a `String ref^` - so we use a `recover val` to end up with a `String val`.

How does this work?

Inside the `recover` expression, your code only has access to **sendable** values from the enclosing lexical scope. In other words, you can only use `iso`, `val` and `tag` things from outside the `recover` expression.

This means that when the `recover` expression finishes, any aliases to the result of the expression other than `iso`, `val` and `tag` ones won't exist anymore. That makes it safe to "lift" the reference capability of the result of the expression.

If the `recover` expression could access **non-sendable** values from the enclosing lexical scope, "lifting" the reference capability of the result wouldn't be safe. Some of those values could "leak" into an `iso` or `val` result, and result in data races.

Automatic receiver recovery

When you have an `iso` or `trn` receiver, you normally can't call `ref` methods on it. That's because the receiver is also an argument to a method, which means both the method body and the caller have access to the receiver at the same time. And *that* means we have to alias the receiver when we call a method on it. The alias of an `iso` is a `tag` (which isn't a subtype of `ref`) and the alias of a `trn` is a `box` (also not a subtype of `ref`).

But we can get around this! If all the arguments to the method (other than the receiver, which is the implicit argument being recovered) *at the call-site* are **sendable**, and the return type of the method is either **sendable** or isn't used *at the call-site*, then we can "automatically recover" the receiver. That just means we don't have to alias the receiver - and *that* means we can call `ref` methods on an `iso` or `trn`, since `iso` and `trn` are both subtypes of `ref`.

Notice that this technique looks mostly at the call-site, rather than at the definition of the method being called. That makes it more flexible. For example, if the method being called wants a `ref` argument, and we pass it an `iso` argument, that's **sendable** at the call-site, so we can still do automatic receiver recovery.

This may sound a little complicated, but in practice, it means you can write code that treats an `iso` mostly like a `ref`, and the compiler will complain when it's wrong. For example:

```
let s = recover String end
s.append("hi")
```

recovering-capabilities-string-append.pony:3:4

Here, we create a `String iso` and then append some text to it. The append method takes a `ref` receiver and a `box` parameter. We can automatically recover the `iso` receiver since we pass a `val` parameter, so everything is fine.

Aliasing

Aliasing means having more than one reference to the same object, within the same actor. This can be the case for a variable or a field.

In most programming languages, aliasing is pretty simple. You just assign some variable to another variable, and there you go, you have an alias. The variable you assign to has the same type (or some supertype) as what's being assigned to it, and everything is fine.

In Pony, that works for some reference capabilities, but not all.

Aliasing and deny guarantees

The reason for this is that the `iso` reference capability denies other `iso` variables that point to the same object. That is, you can only have one `iso` variable pointing to any given object. The same goes for `trn`.

```
fun test(a: Wombat iso) =>
  var b: Wombat iso = a // Not allowed!
```

aliasing-multiple-references-to-an-iso-object.pony:5:6

Here we have some function that gets passed an isolated Wombat. If we try to alias `a` by assigning it to `b`, we'll be breaking reference capability guarantees, so the compiler will stop us. Instead, we can only store aliases that are compatible with the original capability.

What can I alias an `iso` as? Since an `iso` says no other variable can be used by *any* actor to read from or write to that object, we can only create aliases to an `iso` that can neither read nor write. Fortunately, we have a reference capability that does exactly that: `tag`. So we can do this and the compiler will be happy:

```
fun test(a: Wombat iso) =>
  var b: Wombat tag = a // Allowed!
```

aliasing-iso-to-tag.pony:5:6

What about aliasing `trn`? Since a `trn` says no other variable can be used by *any* actor to write to that object, we need something that doesn't allow writing but also doesn't prevent our `trn` variable from writing. Fortunately, we have a reference capability that does that too: `box`. So we can do this and the compiler will be happy:

```
fun test(a: Wombat trn) =>
  var b: Wombat box = a // Allowed!
```

aliasing-trn-to-box.pony:5:6

What about aliasing other stuff? For both `iso` and `trn`, the guarantees require that aliases must give up on some ability (reading and writing for `iso`, writing for `trn`). For the other capabilities (`ref`, `val`, `box` and `tag`), aliases allow for the same operations, so such a reference can just be aliased as itself.

What counts as making an alias?

There are three things that count as making an alias:

1. When you **assign** a value to a variable or a field.
2. When you **pass** a value as an argument to a method.
3. When you **call a method**, an alias of the receiver of the call is created. It is accessible as `this` within the method body.

In all three cases, you are making a new *name* for the object. This might be the name of a local variable, the name of a field, or the name of a parameter to a method.

Alias types

Occasionally we'll want to talk about the type of an alias generically. An alias type is a way of saying "whatever we can safely alias this thing as". We'll discuss generic types later, which will put this to use, but for now it will help us talk about aliases of capabilities in the future.

We indicate an alias type by putting a `!` at the end. Here's an example:

```
fun test(a: A) =>
  var b: A! = a
```

aliasing-alias-types.pony

Here, we're using `A` as a **type variable**, which we'll cover later. So `A!` means "an alias of whatever type `A` is". We can also use it to talk about capabilities: we could have written the statements about `iso` and `trn` as just `iso!` = `tag` and `trn!` = `box`.

Ephemeral types

In Pony, every expression has a type. So what's the type of `consume a`? It's not the same type as `a`, because it might not be possible to alias `a`. Instead, it's an **ephemeral** type. That is, it's a type for a value that currently has no name (it might have a name through some other alias, but not the one we just consumed or destructively read).

To show a type is ephemeral, we put a `^` at the end. For example:

```
fun test(a: Wombat iso): Wombat iso^ =>
  consume a
```

aliasing-ephemeral-types.pony

Here, our function takes an isolated Wombat as a parameter and returns an ephemeral isolated Wombat.

This is useful for dealing with `iso` and `trn` types, and for generic types, but it's also important for constructors. A constructor always returns an ephemeral type, because it's a new object.

Passing and Sharing References

Reference capabilities make it safe to both **pass** mutable data between actors and to **share** immutable data amongst actors. Not only that, they make it safe to do it with no copying, no locks, in fact, no runtime overhead at all.

Passing

For an object to be mutable, we need to be sure that no *other* actor can read from or write to that object. The three mutable reference capabilities (`iso`, `trn`, and `ref`) all make that guarantee.

But what if we want to pass a mutable object from one actor to another? To do that, we need to be sure that the actor that is *sending* the mutable object also *gives up* the ability to both read from and write to that object.

This is exactly what `iso` does. It is *read and write unique*, there can only be one reference at a time that can be used for reading or writing. If you send an `iso` object to another actor, you will be giving up the ability to read from or write to that object.

So I should use `iso` when I want to pass a mutable object between actors? Yes! If you don't need to pass it, you can just use `ref` instead.

Sharing

If you want to **share** an object amongst actors, then we have to make one of the following guarantees:

1. Either *no* actor can write to the object, in which case *any* actor can read from it, or
2. Only *one* actor can write to the object, in which case *other* actors can neither read from or write to the object.

The first guarantee is exactly what `val` does. It is *globally immutable*, so we know that *no* actor can ever write to that object. As a result, you can freely send `val` objects to other actors, without needing to give up the ability to read from that object.

So I should use `val` when I want to share an immutable object amongst actors? Yes! If you don't need to share it, you can just use `ref` instead, or `box` if you want it to be immutable.

The second guarantee is what `tag` does. Not the part about only one actor writing (that's guaranteed by any mutable reference capability), but the part about not being able to read from or write to an object. That means you can freely pass `tag` objects to other actors, without needing to give up the ability to read from or write to that object.

What's the point in sending a tag reference to another actor if it can't then read or write the fields? Because `tag` **can** be used to **identify** objects and sometimes that's all you need. Also, if the object is an actor you can call behaviours on it even though you only have a `tag`.

So I should use `tag` when I want to share the identity of a mutable object amongst actors? Yes! Or, really, the identity of anything, whether it's mutable, immutable, or even an actor.

Reference capabilities that can't be sent

You may have noticed we didn't mention `trn`, `ref`, or `box` as things you can send to other actors. That's because you can't do it. They don't make the guarantees we need in order to be safe.

So when should you use those reference capabilities?

- Use `ref` when you need to be able to change an object over time. On the other hand, if your program wouldn't be any slower if you used an immutable type instead, you may want to use a `val` anyway.
- Use `box` when you don't care whether the object is mutable or immutable. In other words, you want to be able to read it, but you don't need to write to it or share it with other actors.
- Use `trn` when you want to be able to change an object for a while, but you also want to be able to make it *globally immutable* later.

Capability Subtyping

Simple subtypes

Subtyping is about *substitutability*. That is, if we need to supply a certain type, what other types can we substitute instead? Reference capabilities are one important component. We can start by going through a few simpler cases, and then we will talk about the full chart.

First, let's focus on the four capabilities `ref`, `val`, `box`, and `tag`. These have a very useful property: they alias to themselves (and unalias to themselves, as well). This will make the subtyping a lot simpler to work with. Afterwards we can talk about `iso` and `trn`, whose subtyping is more intricate.

To keep things brief, let's add a small shorthand. We will use the `<:` symbol to mean "is a subtype of", or you can read it as "can be used as".

- `ref <: box`. A `ref` can be written to and read from, while `box` only needs the ability to read.
- `val <: box`. A `val` can be read from and is globally immutable, while `box` only requires the ability to read.
- `box <: tag`. A `box` can be read from, while a `tag` doesn't have any permissions at all. In fact, anything can be used as `tag`.

That's all there is to those four. A `ref` could have other mutable aliases, so it can't become `val`, which requires global uniqueness. Likewise, `val` can't become `ref` since it can't be used to write (and there could be other `val` aliases requiring immutability).

Also keep in mind, subtyping is *transitive*. That means that since `val <: box` and `box <: tag`, we also get `val <: tag`. The basic cases will be explained below, and transitivity can be used to derive all other subtyping relationships for capabilities.

Subtypes of unique capabilities

When it comes to talking about unique capabilities, the situation is a bit more complex. With variables, we only had the six basic capabilities, but we're talking about expressions here. We will have to work with aliased and unaliased forms of the capabilities.

From here, let's talk about ephemeral capabilities. Remember that the way to get an ephemeral capability is by *unaliasing*, that is, moving a value out of a named location with `consume` or destructive read.

Subtyping here is surprisingly simple: `iso^` is a sub-capability of absolutely everything, and `trn^` is a sub-capability of `ref` and `val`. Let's go through the interesting cases again with these two:

- `iso^ <: trn^`. An `iso^` guarantees there's no readable or writable aliases, whereas `trn^` just needs no writable aliases.
- `trn^ <: ref`. A `trn^` reference can be used to read and write, which is enough for `ref`.
- `trn^ <: val`. A `trn^` reference has no writable aliases. A `val` requires global immutability, so we can forget our writable access in order to get `val`, since we know no other aliases can write.

Temporary unique access

We talked about aliasing and consuming, but what about when we just use a variable without making a new alias? If `x` is `iso`, what type do we give to the expression `x`? It would be pretty useless if we could only use our `iso` variables as `tag`. We couldn't modify fields or call any methods.

What we get is the bare `iso` capability. Like `ref`, this allows us to read and write, *but* we will have to keep the destination isolated. We will get into what kind of things we can do with it later, but for now, we will talk about subtyping.

- `iso^` <: `iso`. As mentioned earlier, `iso^` can become *anything*. This isn't enormously useful, all told, but an `iso^` expression with no other names is stronger than a expression pointing to an existing `iso` name.
- `trn^` <: `trn`. Similarly, we may use an expression that has no writable aliases, as an expression which has one unique writeable alias.

- `iso` <: `tag`. We can't coerce `iso` to anything else since the original name is still around, but we can always drop down to `tag` (which is just `iso!`).
- `trn` <: `box`. This is quite similar, we can forget our ability to write and just get a new `box` alias to store.

Combining Capabilities

When we talked about fields in the classes and variables chapters, we passed over the detail of field capabilities. Fields, just like variables, have their own capabilities! A `val` field still refers to something permanently immutable. A `tag` field still can't be read from. An `iso` field is still globally unique: it can only be accessed except through this field of a single instance.

Once we have fields with capabilities inside objects with capabilities, now we have two capabilities to keep track of. When a field of an object is accessed or extracted, its reference capability depends both on the reference capability of the field and the reference capability of the **origin**, that is, the object the **field** is being read from. We have to pick a capability for the combination that maintains the guarantees for both the **origin** reference capability, and for the capability of the **field**.

Viewpoint adaptation

The process of combining origin and field capabilities is called **viewpoint adaptation**. That is, the **origin** has a **viewpoint**, and its fields can be "seen" only from that **viewpoint**.

Let's start with a table. This shows how a **field** of each capability looks when using an **origin** of each capability.

	`iso` field	`trn` field	`ref` field	`val` field	`box` field	`tag` field
`iso` origin	`iso`	`tag`	`tag`	`val`	`tag`	`tag`
`trn` origin	`iso`	`box`	`box`	`val`	`box`	`tag`
`ref` origin	`iso`	`trn`	`ref`	`val`	`box`	`tag`
`val` origin	`val`	`val`	`val`	`val`	`val`	`tag`
`box` origin	`tag`	`box`	`box`	`val`	`box`	`tag`
`tag` origin	n/a	n/a	n/a	n/a	n/a	n/a

For example, if you have a `trn` origin and you read a `ref` field, you get a `box` result:

Explaining why

That table will seem totally natural to you, eventually. But probably not yet. To help it seem natural, let's walk through each cell in the table and explain why it is the way it is.

Reading from an `iso` variable

Anything read through an `iso` origin has to maintain the isolation guarantee that the origin has. The key thing to remember is that the `iso` can be sent to another actor and it can also become any other reference capability. So when we read a field, we need to get a result that won't ever break the isolation guarantees that the origin makes, that is, *read and write uniqueness*.

An `iso` field makes the same guarantees as an `iso` origin, so that's fine to read. A `val` field is *globally immutable*, which means it's always ok to read it, no matter what the origin is (well, other than `tag`).

Everything else, though, can break our isolation guarantees. That's why other reference capabilities are seen as `tag`: it's the only type that is neither readable nor writable.

Reading from a `trn` variable

This is like `iso`, but with a weaker guarantee (*write uniqueness* as opposed to *read and write uniqueness*). That makes a big difference since now we can return something readable when we enforce our guarantees.

An `iso` field makes stronger guarantees than `trn`, and can't alias anything readable inside the `trn` origin, so it's perfectly safe to read.

On the other hand, `trn` and `ref` fields have to be returned as `box`. It might seem a bit odd that `trn` has to be returned as `box`, since after all it guarantees write uniqueness itself and we might expect it to behave like `iso`. The issue is that `trn`, unlike `iso`, *can* alias with some `box` variables in the origin. And that `trn` origin still has to make the guarantee that nothing else can write to fields that it can read. On the other hand, `trn` still can't be returned as `val`, because then we might leave the original field in place and create a `val` alias, while that field can still be used to write! So we have to view it as `box`.

Immutable and opaque capabilities, though, can never violate write uniqueness, so `val`, `box`, and `tag` are viewed as themselves.

Reading from a `ref` variable

A `ref` origin doesn't modify its fields at all. This is because a `ref` origin doesn't make any guarantees that are incompatible with its fields.

Reading from a `val` variable

A `val` origin is deeply and globally immutable, so all of its fields are also `val`. The only exception is a `tag` field. Since we can't read from it, we also can't guarantee that nobody can write to it, so it stays `tag`.

Reading from a `box` variable

A `box` variable is locally immutable. This means it's possible that it may be mutated through some other variable (a `trn` or a `ref`), but it's also possible that our `box` variable is an alias of some `val` variable.

When we read a field, we need to return a reference capability that is compatible with the field but is also locally immutable.

An `iso` field is returned as a `tag` because no locally immutable reference capability can maintain its isolation guarantees. A `val` field is returned as a `val` because global immutability is a stronger guarantee than local immutability. A `box` field makes the same local immutability guarantee as its origin, so that's also fine.

For `trn` and `ref` we need to return a locally immutable reference capability that doesn't violate any guarantees the field makes. In both cases, we can return `box`.

Reading from a `tag` variable

This one is easy: `tag` variables are opaque! They can't be read from.

Writing to the field of an object

Like reading the field of an object, writing to a field depends on the reference capability of the object reference being stored and the reference capability of the origin object containing the field. The reference capability of the object being stored must not violate the guarantees made by the origin object's reference capability. For example, a `val` object reference can be stored in an `iso` origin. This is because the `val` reference capability guarantees that no alias to that object exists which could violate the guarantees that the `iso` capability makes.

Here's a simplified version of the table above that shows which reference capabilities can be stored in the field of an origin object.

	`iso` object	`trn` object	`ref` object	`val` object	`box` object	`tag` object
`iso` origin						
`trn` origin						
`ref` origin						
`val` origin						
`box` origin						
`tag` origin						

The bottom half of this chart is empty, since only origins with a mutable capability can have their fields modified.

Arrow Types aka Viewpoints

When we talked about **reference capability composition** and **viewpoint adaptation**, we dealt with cases where we know the reference capability of the origin. However, sometimes we don't know the precise reference capability of the origin.

When that happens, we can write a **viewpoint adapted type**, which we call an **arrow type** because we write it with an `->`.

Using `this->` as a viewpoint

A function with a `box` receiver can be called with a `ref` receiver or a `val` receiver as well since those are both subtypes of `box`. Sometimes, we want to be able to talk about a type to take this into account. For example:

```
class Wombat
  var _friend: Wombat

  fun friend(): this->Wombat => _friend
```

arrow-types-this.pony

Here, we have a `Wombat`, and every `Wombat` has a friend that's also a `Wombat` (lucky `Wombat`). In fact, it's a `Wombat ref`, since `ref` is the default reference capability for a `Wombat` (since we didn't specify one). We also have a function that returns that friend. It's got a `box` receiver (because `box` is the default receiver reference capability for a function if we don't specify it).

So the return type would normally be a `Wombat box`. Why's that? Because, as we saw earlier, when we read a `ref` field from a `box` origin, we get a `box`. In this case, the origin is the receiver, which is a `box`.

But wait! What if we want a function that can return a `Wombat ref` when the receiver is a `ref`, a `Wombat val` when the receiver is a `val`, and a `Wombat box` when the receiver is a `box`? We don't want to have to write the function three times.

We use `this->`! In this case, `this->Wombat`. It means "a `Wombat ref` as seen by the receiver".

We know at the *call site* what the real reference capability of the receiver is. So when the function is called, the compiler knows everything it needs to know to get this right.

Using a type parameter as a viewpoint

We haven't covered generics yet, so this may seem a little weird. We'll cover this again when we talk about generics (i.e. parameterised types), but we're mentioning it here for completeness.

Another time we don't know the precise reference capability of something is if we are using a type parameter. Here's an example from the standard library:

```
class ListValues[A, N: ListNode[A] box] is Iterator[N->A]
```

arrow-types-type-parameter.pony

Here, we have a `ListValues` type that has two type parameters, `A` and `N`. In addition, `N` has a constraint: it has to be a subtype of `ListNode[A] box`. That's all fine and well, but we also say the `ListValues[A, N]` provides `Iterator[N->A]`. That's the interesting bit: we provide an interface that let's us iterate over values of the type `N->A`.

That means we'll be returning objects of the type `A`, but the reference capability will be the same as an object of type `N` would see an object of type `A`.

Using `box->` as a viewpoint

There's one more way we use arrow types, and it's also related to generics. Sometimes we want to talk about a type parameter as it is seen by some unknown type, *as long as that type can read the type parameter.*

In other words, the unknown type will be a subtype of `box`, but that's all we know. Here's an example from the standard library:

```
interface Comparable[A: Comparable[A] box]
  fun eq(that: box->A): Bool => this is that
  fun ne(that: box->A): Bool => not eq(that)
```

arrow-types-box.pony

Here, we say that something is `Comparable[A]` if and only if it has functions `eq` and `ne` and those functions have a single parameter of type `box->A` and return a `Bool`. In other words, whatever `A` is bound to, we only need to be able to read it.

Reference Capability Matrix

At this point, it's quite possible that you read the previous sections in this chapter and are still pretty confused about the relation between reference capabilities. It's okay! We have all struggled when learning this part of Pony, too. Once you start working on Pony code, you'll get a better intuition with them.

In the meantime, if you still feel like all these tidbits in the chapter are still scrambled in your head, there is one resource often presented with Pony that can give you a more visual representation: the **reference capability matrix**.

It is also the origin of the concept behind each capability in Pony, in the sense of how each capability denies certain properties to its reference – in other words, which guarantees a capability makes. We will explain what that actually means before presenting the matrix.

Local and global aliases

Before anything else, we want to clarify what we mean by "local" and "global" aliases.

A local alias is a reference to the same variable that exists in the same actor. Whenever you pass a value around, and it's not the argument of an actor's behavior, this is the kind of alias we are working with.

On the other hand, a global alias is a reference to the same variable that can exist in a *different* actor. That is, it describes the properties of how two or more actors could interact with the same reference.

Each reference capability in Pony is actually a pair of local guarantees and global guarantees. For instance, `ref` doesn't deny any read/write capabilities inside the actor, but denies other actors from reading or writing to that reference.

You may recall from the Reference Capability Guarantees section that mutable references cannot be safely shared between actors, while immutable references can be read by multiple actors. In general, global properties are always as restrictive or more restrictive than the local properties to that reference - what is denied globally must also be denied locally. For example, it's not possible to write to an immutable reference in either a global or local alias. It's also not possible to read from or write to an opaque reference, `tag`. Therefore, some combinations of local and global aliases are impossible, and have no designated capabilities.

Reference capability matrix

Without further ado, here's the reference capability matrix:

	Deny global read/write aliases	Deny global write aliases	Don't deny any global aliases
Deny local read/write aliases	`iso`		
Deny local write aliases	`trn`	`val`	
Don't deny any local aliases	`ref`	`box`	`tag`
	(Mutable)	*(Immutable)*	*(Opaque)*

In the context of the matrix, "denying a capability" means that any other alias to that reference is not allowed to do that action. For example, since `trn` denies other local write aliases (but allows reads), this is the only reference that allows writing to the object; and since it denies both read and write aliases to other actors, it's safe to write inside this actor, thus being mutable. And since `box` does not break any guarantees that `trn` makes (local reads are allowed, but global writes are forbidden), we can create `box` aliases to a `trn` reference.

You'll notice that the top-right side is empty. That's because, as previously discussed, we cannot make any local guarantees that are more restrictive than the global guarantees, or we'd end up with invalid capabilities that could be written to in this actor but read somewhere else at the same time.

The matrix also helps visualizing other concepts previously discussed in this chapter:

- **Sendable capabilities**. If we want to send references to a different actor, we must make sure that the global and local aliases make the same guarantees. It'd be unsafe to send a `trn` to another actor, since we could possibly hold `box` references locally. Only `iso`, `val`, and `tag` have the same global and local restrictions – all of which are in the main diagonal of the matrix.
- **Ephemeral subtyping**. If we have an ephemeral capability (for instance, `iso^` after consuming an isolated variable), we can be more permissive for the new alias, i.e. remove restrictions, such as allowing local aliases with read capabilities, and receive the reference into a `trn^`; or both read and write, which gives us `ref`. The same is true for more global alias, and we can get `val`, `box`, or `tag`. Visually, this would be equivalent to walking downwards and/or to-the-right starting from the capability in the matrix.
- **Recovering capabilities**. This is when we "lift" a capability, from a mutable reference to `iso` or an immutable reference to `val`. The matrix equivalent would be walking upwards starting from the capability – quite literally lifting in this case.
- **Aliasing**. With a bit more of imagination, it's possible to picture aliasing `iso` and `trn` as reflecting them on the secondary diagonal of the matrix onto `tag` and `box`, respectively. The reason for that lies on which restrictions arise from the local guarantees. An `iso` doesn't allow different aliases to read or write, which `tag` enforces; and `trn` doesn't allow different aliases to write but allows them to do local reads, fitting `box`'s restrictions.

We want to emphasize that trying to apply the reference capability matrix to some capabilities problems is not guaranteed to work (viewpoint adaptation is one example). The matrix is the original definition of the reference capabilities, presented here as a mnemonic device. Whenever you struggle with reference capabilities, we recommend that you reread the corresponding section of this chapter to understand why something is not allowed by the compiler.

Object Capabilities

If you are reading this tutorial in order, you've just finished the reference capabilities chapter and your brain probably hurts. We're sorry about that. Hopefully object capabilities, while a new concept, are less mind bending.

We touched on object capabilities previously in the tutorial, this chapter will dig in more. So, what is an object capability?

A capability is the ability to do "something". Usually that "something" involves an external resource that you might want access to; like the file system or the network. This is called an object capability. Object capabilities have appeared in a number of programming languages including E.

Object Capabilities

Pony's capabilities-secure type system is based on the object-capability model. That sounds complicated, but really it's elegant and simple. The core idea is this:

> A capability is an unforgeable token that (a) designates an object and (b) gives the program the authority to perform a specific set of actions on that object.

So what's that token? It's an address. A pointer. A reference. It's just… an object.

How is that unforgeable?

Since Pony has no pointer arithmetic and is both type-safe and memory-safe, object references can't be "invented" (i.e. forged) by the program. You can only get one by constructing an object or being passed an object.

What about the C-FFI? Using the C-FFI can break this guarantee. We'll talk about the **C-FFI trust boundary** later, and how to control it.

What about global variables?

They're bad! Because you can get them without either constructing them or being passed them.

Global variables are a form of what is called *ambient authority*. Another form of ambient authority is unfettered access to the file system.

Pony has no global variables and no global functions. That doesn't mean all ambient authority is magically gone - we still need to be careful about the file system, for example. Having no global variables is necessary, but not sufficient, to eliminate ambient authority.

How does this help?

Instead of having permissions lists, access control lists, or other forms of security, the object-capabilities model means that if you have a reference to an object, you can do things with that object. Simple and effective.

There's a great paper on how the object-capability model works, and it's pretty easy reading:

Capability Myths Demolished

Capabilities and concurrency

The object-capability model on its own does not address concurrency. It makes clear *what* will happen if there is simultaneous access to an object, but it does not prescribe a single method of controlling this.

Combining capabilities with the actor-model is a good start, and has been done before in languages such as E and Joule.

Pony does this and also uses a system of *reference capabilities* in the type system.

Delegating and restricting authority

Any interesting program will need to interact with the outside world, like accessing the network or the file system, or by creating and communicating with other programs. We call **ambient**

authority all those rights implicitly granted to the program to make these things possible, and because Pony makes this concept *explicit*, we need to take the time to talk about what it means and how it works. In other languages like for example C, you can always attempt to do an operation like I/O, and it will usually succeed save some runtime checks (your disk being full might make an operation fail, for example). But in Pony, any piece of code interacting with the outside world needs the authority to do so.

The operating system essentially knows nothing about the structure of any program it runs, in particular it is ignorant of any Pony specific concepts. In order to impose that some code is authorized for an operation like writing to disk, Pony requires a special argument to be passed, a capability bearing the authority required for the task at hand. The type system guarantees that inadequate capabilities for a task fail at compile time.

Recall the definition of capability from the Object Capabilities section:

> A capability is an unforgeable token that (a) designates an object and (b) gives the program the authority to perform a specific set of actions on that object.

In Pony, the `Main` actor is created with an `Env` object, which holds the unforgeable `AmbientAuth` token in its root field. This value is the capability that represents the ambient authority given to us by the system.

Here is a program that connects to example.com via TCP on port 80 and quits:

```
use "net"

class MyTCPConnectionNotify is TCPConnectionNotify
  let _out: OutStream

  new iso create(out: OutStream) =>
    _out = out

  fun ref connected(conn: TCPConnection ref) =>
    _out.print("connected")
    conn.close()

  fun ref connect_failed(conn: TCPConnection ref) =>
    _out.print("connect_failed")

actor Connect
  new create(out: OutStream, auth: TCPConnectAuth) =>
    TCPConnection(auth, MyTCPConnectionNotify(out), "example.com", "80")

actor Main
  new create(env: Env) =>
    Connect(env.out, TCPConnectAuth(env.root))
```

derived-authority-delegating-and-restricting-authority.pony

The `Main` actor authorizes the `Connect` actor by passing a `TCPConnectAuth` token created from the ambient authority token in `env.root`. The ambient authority token is unforgeable since the `AmbientAuth` constructor is private and the only existing instance is provided by the runtime itself.

The `Connect` actor uses this derived authority when it creates a TCP connection:

```
    TCPConnection(auth, MyTCPConnectionNotify(out), "example.com", "80")
```

derived-authority-delegating-and-restricting-authority.pony:18:18

The `TCPConnection` requires an authority as first parameter, and since the compiler checks that the correct type was passed, this guarantees that a `TCPConnection` can only be created by an actor holding the required authorization.

The implementation of the `TCPConnection` constructor does not even use the authorization parameter at run time, all it does is require it to be of the right type. The type checking done by the compiler is sufficient for this guarantee.

Restrict, then delegate your authority

In order to handle our own code and that of others more safely, and also to understand our code better, we want to split up the authority, and only grant the particular authority a piece of code actually requires.

The first parameter of the `TCPConnection` constructor has the type `TCPConnectAuth`. This is what we call "the most specific authority". All classes in the standard library that require an authority token only accept a single type of token; the token of "most specific authority". In the case of `TCPConnection`, this is `TCPConnectAuth`.

Now imagine we don't trust the `Connect` actor, so we don't want to provide it with more authority than needed. For example, there is no point in granting it filesystem access, since it is supposed to do network things (specifically, TCP), not access the filesystem. Instead of passing the entire `AmbientAuth` (the root of all authority), we "downgrade" that to a `TCPConnectAuth` (the most restrictive authority in `net`), pass it to the `Connect` actor, and have that pass it to the `TCPConnection` constructor:

```
actor Connect
  new create(out: OutStream, auth: TCPConnectAuth) =>
    TCPConnection(auth, MyTCPConnectionNotify(out), "example.com", "80")

actor Main
  new create(env: Env) =>
    try Connect(env.out, TCPConnectAuth(env.root)) end
```

derived-authority-restrict-then-delegate-your-authority.pony:16:22

Now we are sure it cannot access the filesystem or listen on a TCP or UDP port. Pay close mind to the authority that code you are calling is asking for. Never give `AmbientAuth` to **any** code you do not trust completely both now and in the future. You should always create the most specific authority and give the library that authority. If the library is asking for more authority than it needs, **do not use the library**.

Authorization-friendly interface

Consider the above example again, but this time let's think of the `Connect` actor being part of a 3rd party package that we are building. Our goal is to write the actor in such a way that users of our package can grant it only the authority necessary for it to function.

As the package author, it is then our responsibility to realize that the minimal authority possible is the `TCPConnectAuth`. We should only request `TCPConnectAuth` from our users. Our current implementation already satisfies this requirement. Rather than requesting a less specific authority like `AmbientAuth` from our users and creating the `TCPConnectAuth` in our library, we only ask for the `TCPConnetAuth` that is required.

Authority hierarchies

Let's have a look at the authorizations available in the standard library's `net` package.

```
primitive NetAuth
  new create(from: AmbientAuth) =>
    None

primitive DNSAuth
  new create(from: (AmbientAuth | NetAuth)) =>
    None

primitive UDPAuth
  new create(from: (AmbientAuth | NetAuth)) =>
    None

primitive TCPAuth
  new create(from: (AmbientAuth | NetAuth)) =>
    None

primitive TCPListenAuth
  new create(from: (AmbientAuth | NetAuth | TCPAuth)) =>
    None

primitive TCPConnectAuth
  new create(from: (AmbientAuth | NetAuth | TCPAuth)) =>
    None
```

derived-authority-authority-hierarchies.pony

Look at the constructor for `TCPConnectAuth`:

```
  new create(from: (AmbientAuth | NetAuth | TCPAuth)) =>
```

derived-authority-authority-hierarchies.pony:22:22

you might notice that this looks like a hierarchy of authorities:

```
AmbientAuth >> NetAuth >> TCPAuth >> TCPConnectAuth
```

where in this paragraph, "»" means "grants at least as much authority as". In fact, the `AmbientAuth` encompasses all ambient authority and is a strictly larger authority than `NetAuth`, which grants access to the network, which is more powerful than `TCPAuth` which is restricted to the TCP protocol. Finally, `TCPConnectAuth` is good only for creating a `TCPConnection`.

This hierarchy is established by means of the constructor of the weaker authority accepting one of the stronger authorities, for example:

```
primitive TCPAuth
  new create(from: (AmbientAuth | NetAuth)) =>
    None
```

derived-authority-authority-hierarchies.pony:13:15

Where TCPAuth grants less authority than NetAuth. NetAuth can be used to create any of the derived authorities DNSAuth, UDPAuth, TCPAuth, TCPListenAuth, TCPConnectAuth whereas TCPAuth can only be used to derive TCPListenAuth and TCPConnectAuth.

Trust Boundary

We mentioned previously that the C-FFI can be used to break pretty much every guarantee that Pony makes. This is because, once you've called into C, you are executing arbitrary machine code that can stomp memory addresses, write to anything, and generally be pretty badly behaved.

Trust boundaries

When we talk about trust, we don't mean things you trust because you think they are perfect. Instead, we mean things you *have* to trust in order to get things done, even though you know they are *imperfect*.

In Pony, when you use the C-FFI, you are basically declaring that you trust the C code that's being executed. That's fine, because you may need it to get work done. But what about trusting someone else's code to use the C-FFI? You may need to, but you definitely want to know that it's happening.

Safe packages

The normal way to handle that is to be sure you're using just the code you need to use in your program. Pretty simple! Don't use some random package from the internet without looking at the code and making sure it doesn't do nasty FFI stuff.

But we can do better than that.

In Pony, you can optionally declare a set of *safe* packages on the ponyc command line, like this:

```
ponyc --safe=files:net:process my_project
```

Here, we are declaring that only the files, net and process packages are allowed to use C-FFI calls. We've established our trust boundary: any other packages that try to use C-FFI calls will result in a compile-time error.

Generics

Often when writing code you want to create similar classes or functions that differ only in the type that they operate on. The classic example of this is collection classes. You want to be able to create an Array that can hold objects of a particular type without creating an IntArray, StringArray, etc. This is where generics step in.

Generic Classes

A generic class is a class that can have parameters, much like a method has parameters. The parameters for a generic class are types, including their reference capability. Parameters are introduced to a class using square brackets.

Take the following example of a non-generic class:

```
class Foo
  var _c: U32
  new create(c: U32) =>
    _c = c

  fun get(): U32 => _c

  fun ref set(c: U32) => _c = c

actor Main
  new create(env:Env) =>
    let a = Foo(42)
    env.out.print(a.get().string())
    a.set(21)
    env.out.print(a.get().string())
```

generics-foo-non-generic.pony

This class only works for the type `U32`, a 32 bit unsigned integer. We can make this work over other types by making the type a parameter to the class. For this example it looks like:

```
class Foo[A: Any val]
  var _c: A

  new create(c: A) =>
    _c = c

  fun get(): A => _c

  fun ref set(c: A) => _c = c

actor Main
  new create(env:Env) =>
    let a = Foo[U32](42)
    env.out.print(a.get().string())
    a.set(21)

    env.out.print(a.get().string())
    let b = Foo[F32](1.5)
    env.out.print(b.get().string())

    let c = Foo[String]("Hello")
    env.out.print(c.get().string())
```

generics-foo-with-any-val.pony

The first thing to note here is that the `Foo` class now takes a type parameter in square brackets, `[A: Any val]`. That syntax for the type parameter is:

```
Name: Constraint ReferenceCapability
```

In this case, the name is `A`, the constraint is `Any` and the reference capability is `val`. `Any` is used to mean that the type can be any type - it is not constrained. The remainder of the class definition replaces `U32` with the type name `A`.

The user of the class must provide a type when referencing the class name. This is done when creating it:

That tells the compiler what specific class to create, replacing `A` with the type provided. For example, a `Foo[String]` usage becomes equivalent to:

```
class FooString
  var _c: String val

  new create(c: String val) =>
    _c = c

  fun get(): String val => _c

  fun ref set(c: String val) => _c = c
```

generics-foo-string.pony:1:9

Type parameter defaults

Sometimes the same parameter type is used over and over again, and it is tedious to always specify it when using the generic class. The class `Bar` expects its type parameter to be a `USize val` by default:

```
class Bar[A: Any box = USize val]
  var _c: A

  new create(c: A) =>
    _c = c

  fun get(): A => _c

  fun ref set(c: A) => _c = c
```

generics-type-parameter-defaults.pony:1:9

Now, when the default type parameter is the desired one, it can simply be omitted. But it is still possible to be explicit or use a different type.

```
    let a = Bar(42)
    let b = Bar[USize](42)
    let c = Bar[F32](1.5)
```

generics-type-parameter-defaults.pony:13:15

Note that we could simply write `class Bar[A: Any box = USize]` because `val` is the default reference capability of the `USize` type.

Generic Methods

Methods can be generic too. They are defined in the same way as normal methods but have type parameters inside square brackets after the method name:

```
primitive Foo
  fun bar[A: Stringable val](a: A): String =>
    a.string()

actor Main
```

```
  new create(env:Env) =>
    let a = Foo.bar[U32](10)
    env.out.print(a.string())

    let b = Foo.bar[String]("Hello")
    env.out.print(b.string())
```

generics-generic-methods.pony

This example shows a constraint other than `Any`. The `Stringable` type is any type with a `string()` method to convert to a `String`.

These examples show the basic idea behind generics and how to use them. Real world usage gets quite a bit more complex and the following sections will dive deeper into how to use them.

Generics and Reference Capabilities

In the examples presented previously we've explicitly set the reference capability to `val`:

```
class Foo[A: Any val]
```

generics-foo-with-any-val.pony:1:1

If the capability is left out of the type parameter then the generic class or function can accept any reference capability. This would look like:

```
class Foo[A: Any]
```

generics-and-reference-capabilities-explicit-constraint-and-default-capability.pony

It can be made shorter because `Any` is the default constraint, leaving us with:

```
class Foo[A]
```

generics-and-reference-capabilities-default-capability-and-constraint.pony

This is what the example shown before looks like but with any reference capability accepted:

```
// Note - this won't compile
class Foo[A]
  var _c: A

  new create(c: A) =>
    _c = c

  fun get(): A => _c

  fun ref set(c: A) => _c = c

actor Main
  new create(env: Env) =>
    let a = Foo[U32](42)
    env.out.print(a.get().string())
    a.set(21)
    env.out.print(a.get().string())
```

generics-and-reference-capabilities-accept-any-reference-capability.pony

Unfortunately, this doesn't compile. For a generic class to compile it must be compilable for all possible types and reference capabilities that satisfy the constraints in the type parameter. In this case, that's any type with any reference capability. The class works for the specific reference capability of `val` as we saw earlier, but how well does it work for `ref`? Let's expand it and see:

```
// Note - this also won't compile
class Foo
  var _c: String ref

  new create(c: String ref) =>
    _c = c

  fun get(): String ref => _c

  fun ref set(c: String ref) => _c = c

actor Main
  new create(env: Env) =>
    let a = Foo(recover ref String end)
    env.out.print(a.get().string())
    a.set(recover ref String end)
    env.out.print(a.get().string())
```

generics-and-reference-capabilities-foo-ref.pony

This does not compile. The compiler complains that `get()` doesn't actually return a `String ref`, but `this->String ref`. We obviously need to simply change the type signature to fix this, but what is going on here? `this->String ref` is an arrow type. An arrow type with "this->" states to use the capability of the actual receiver (`ref` in our case), not the capability of the method (which defaults to `box` here). According to viewpoint adaption this will be `ref->ref` which is `ref`. Without this arrow type we would only see the field `_c` as `box` because we are in a `box` method.

So let's apply what we just learned:

```
class Foo
  var _c: String ref

  new create(c: String ref) =>
    _c = c

  fun get(): this->String ref => _c

  fun ref set(c: String ref) => _c = c

actor Main
  new create(env: Env) =>
    let a = Foo(recover ref String end)
    env.out.print(a.get().string())
    a.set(recover ref String end)
    env.out.print(a.get().string())
```

generics-and-reference-capabilities-foo-ref-and-this-ref.pony

That compiles and runs, so `ref` is valid now. The real test though is `iso`. Let's convert the class to `iso` and walk through what is needed to get it to compile. We'll then revisit our generic class to get it working:

An `iso` specific class

```
// Note - this won't compile
class Foo
  var _c: String iso

  new create(c: String iso) =>
    _c = c

  fun get(): this->String iso => _c

  fun ref set(c: String iso) => _c = c

actor Main
  new create(env: Env) =>
    let a = Foo(recover iso String end)
    env.out.print(a.get().string())
    a.set(recover iso String end)
    env.out.print(a.get().string())
```

generics-and-reference-capabilities-foo-iso.pony

This fails to compile. The first error is:

```
main.pony:5:8: right side must be a subtype of left side
    _c = c
       ^
    Info:
    main.pony:4:17: String iso! is not a subtype of String iso: iso! is not a subtype of iso
      new create(c: String iso) =>
                  ^
```

The error is telling us that we are aliasing the `String iso` - The `!` in `iso!` means it is an alias of an existing `iso`. Looking at the code shows the problem:

```
  new create(c: String iso) =>
    _c = c
```

generics-and-reference-capabilities-foo-iso.pony:5:6

We have `c` as an `iso` and are trying to assign it to `_c`. This creates two aliases to the same object, something that `iso` does not allow. To fix it for the `iso` case we have to `consume` the parameter. The correct constructor should be:

```
  new create(c: String iso) =>
    _c = consume c
```

generics-and-reference-capabilities-foo-iso-consume-iso-constructor-parameter.pony:4:5

A similar issue exists with the `set` method. Here we also need to consume the variable `c` that is passed in:

```
fun set(c: String iso) => _c = consume c
```

generics-and-reference-capabilities-foo-iso-consume-iso-function-parameter.pony

Now we have a version of Foo that is working correctly for iso. Note how applying the arrow type to the get method also works for iso. But here the result is a different one, by applying viewpoint adaptation we get from ref->iso (with ref being the capability of the receiver, the Foo object referenced by a) to iso. Through the magic of automatic receiver recovery we can call the string method on it:

```
class Foo
  var _c: String iso

  new create(c: String iso) =>
    _c = consume c

  fun get(): this->String iso => _c

  fun ref set(c: String iso) => _c = consume c

actor Main
  new create(env: Env) =>
    let a = Foo(recover iso String end)
    env.out.print(a.get().string())
    a.set(recover iso String end)
    env.out.print(a.get().string())
```

generics-and-reference-capabilities-foo-iso-consume-iso-constructor-parameter.pony

A capability generic class

Now that we have iso working we know how to write a generic class that works for iso and it will work for other capabilities too:

```
class Foo[A]
  var _c: A

  new create(c: A) =>
    _c = consume c

  fun get(): this->A => _c

  fun ref set(c: A) => _c = consume c

actor Main
  new create(env: Env) =>
    let a = Foo[String iso]("Hello".clone())
    env.out.print(a.get().string())

    let b = Foo[String ref](recover ref "World".clone() end)
    env.out.print(b.get().string())

    let c = Foo[U8](42)
    env.out.print(c.get().string())
```

generics-and-reference-capabilities-capability-generic-class.pony

It's quite a bit of work to get a generic class or method to work across all capability types, in particular for `iso`. There are ways of restricting the generic to subsets of capabilities and that's the topic of the next section.

Constraints

Capability Constraints

The type parameter constraint for a generic class or method can constrain to a particular capability as seen previously:

```
class Foo[A: Any val]
```

generics-foo-with-any-val.pony:1:1

Without the constraint, the generic must work for all possible capabilities. Sometimes you don't want to be limited to a specific capability and you can't support all capabilities. The solution for this is generic constraint qualifiers. These represent classes of capabilities that are accepted in the generic. The valid qualifiers are:

	Capabilities allowed	Description
`#read`	`ref`, `val`, `box`	Anything you can read from
`#send`	`iso`, `val`, `tag`	Anything you can send to an actor
`#share`	`val`, `tag`	Anything you can send to more than one actor
`#any`	`iso`, `trn`, `ref`, `val`, `box`, `tag`	Default of a constraint
`#alias`	`ref`, `val`, `box`, `tag`	Set of capabilities that alias as themselves (used by compiler)

In the previous section, we went through extra work to support `iso`. If there's no requirement for `iso` support we can use `#read` and support `ref`, `val`, and `box`:

```
class Foo[A: Any #read]
  var _c: A

  new create(c: A) =>
    _c = c

  fun get(): this->A => _c

  fun ref set(c: A) => _c = c

actor Main
  new create(env:Env) =>
    let a = Foo[String ref](recover ref "hello".clone() end)
    env.out.print(a.get().string())

    let b = Foo[String val]("World")
    env.out.print(b.get().string())
```

generic-constraints-foo-any-read.pony

Packages

Pony code is organised into **packages**. Each program and library is a single package, possibly using other packages.

The package structure

The package is the basic unit of code in Pony. It corresponds directly to a directory in the file system, all Pony source files within that directory are within that package. Note that this does not include files in any sub-directories.

Every source file is within exactly one package. Hence all Pony code is in packages.

A package is usually split into several source files, although it does not have to be. This is purely a convenience to allow better code organisation and the compiler treats all the code within a package as if it were from a single file.

The package is the privacy boundary for types and methods. That is:

1. Private types (those whose name starts with an underscore) can be used only within the package in which they are defined.
2. Private methods (those whose name starts with an underscore) can be called only from code within the package in which they are defined.

It follows that all code within a package is assumed to know and trust, all the rest of the code in the package.

There is no such concept as a sub-package in Pony. For example, the packages "foo/bar" and "foo/bar/wombat" will, presumably, perform related tasks but they are two independent packages. Package "foo/bar" does not contain package "foo/bar/wombat" and neither has access to the private elements of the other.

Use Statement

To use a package in your code you need to have a **use** command. This tells the compiler to find the package you need and make the types defined in it available to you. Every Pony file that needs to know about a type from a package must have a use command for it.

Use commands are a similar concept to Python and Java "import", C/C++ "#include" and C# "using" commands, but not exactly the same. They come at the beginning of Pony files and look like this:

```
use "collections"
```

use-statement-collections.pony

This will find all of the publicly visible types defined in the *collections* package and add them to the type namespace of the file containing the `use` command. These types are then available to use within that file, just as if they were defined locally.

For example, the standard library contains the package *time*. This contains the following definition (among others):

```
primitive Time
  fun now(): (I64, I64)
```

use-statement-time.pony

To access the *now* function just add a use command:

```
use "time"

class Foo
  fun f() =>
    (var secs, var nsecs) = Time.now()
```

use-statement-time-now.pony

Use names

As we saw above the use command adds all the public types from a package into the namespace of the using file. This means that using a package may define type names that you want to use for your own types. Furthermore, if you use two packages within a file they may both define the same type name, causing a clash in your namespace. For example:

```
// In package A
class Foo

// In package B
class Foo

// In your code
use "packageA"
use "packageB"

class Bar
  var _x: Foo
```

use-statement-use-names-conflict.pony

The declarations of _x is an error because we don't know which `Foo` is being referred to. Actually using 'Foo' is not even required, simply using both `packageA` and `packageB` is enough to cause an error here.

To avoid this problem the use command allows you to specify an alias. If you do this then only that alias is put into your namespace. The types from the used package can then be accessed using this alias as a qualifier. Our example now becomes:

```
// In package A
class Foo

// In package B
class Foo

// In your code
use a = "packageA"
use b = "packageB"

class Bar
  var _x: a.Foo  // The Foo from package A
  var _y: b.Foo  // The Foo from package B
```

use-statement-use-names-resolution.pony

If you prefer you can give an alias to only one of the packages. `Foo` will then still be added to your namespace referring to the unaliased package:

```
// In package A
class Foo

// In package B
class Foo

// In your code
use "packageA"
use b = "packageB"

class Bar
  var _x: Foo  // The Foo from package A
  var _y: b.Foo  // The Foo from package B
```

use-statement-use-names-resolution-alternative.pony

Can I just specify the full package path and forget about the use command, like I do in Java and C#? No, you can't do that in Pony. You can't refer to one package based on a `use` command for another package and you can't use types from a package without a use command for that package. Every package that you want to use must have its own `use` command.

Are there limits on the names I can use for an alias? Use alias names have to start with a lower case letter. Other than that you can use whatever name you want, as long as you're not using that name for any other purpose in your file.

Scheme indicators

The string we give to a `use` command is known as the *specifier*. This consists of a *scheme* indicator and a *locator*, separated by a colon. The scheme indicator tells the `use` command what we want it to do, for example, the scheme indicator for including a package is "package". If no colon is found within the specifier string then the use command assumes you meant "package".

The following two use commands are exactly equivalent:

```
use "foo"
use "package:foo"
```

use-statement-scheme-indicators-optional-package-scheme-specifier.pony

If you are using a locator string that includes a colon, for example, an absolute path in Windows, then you **have** to include the "package" scheme specifier:

```
use "C:/foo/bar"  // Error, scheme "C" is unknown
use "package:C:/foo/bar"  // OK
```

use-statement-scheme-indicators-required-package-scheme-specifier.pony

To allow use commands to be portable across operating systems, and to avoid confusion with escape characters, '/' should always be used as the path separator in use commands, even on Windows.

Standard Library

The Pony standard library is a collection of packages that can each be used as needed to provide a variety of functionality. For example, the **files** package provides file access and the **collections** package provides generic lists, maps, sets and so on.

There is also a special package in the standard library called **builtin**. This contains various types that the compiler has to treat specially and are so common that all Pony code needs to know about them. All Pony source files have an implicit `use "builtin"` command. This means all the types defined in the package builtin are automatically available in the type namespace of all Pony source files.

Documentation for the standard library is available online

Testing

Unto all code, good or bad, comes the needs to test it. Verification that our code does what we expect is very important. Over the last 20 years, there has been an explosion in different testing techniques and tools. This chapter will get you going with PonyTest, the current Pony testing tool.

Testing with PonyTest

PonyTest is Pony's unit testing framework. It is designed to be as simple as possible to use, both for the unit test writer and the user running the tests.

Each unit test is a class, with a single test function. By default, all tests run concurrently.

Each test run is provided with a helper object. This provides logging and assertion functions. By default log messages are only shown for tests that fail.

When any assertion function fails the test is counted as a fail. However, tests can also indicate failure by raising an error in the test function.

Example program

To use PonyTest simply write a class for each test and a `TestList` type that tells the PonyTest object about the tests. Typically the `TestList` will be Main for the package.

The following is a complete program with 2 trivial tests.

```
use "pony_test"

actor Main is TestList
  new create(env: Env) =>
    PonyTest(env, this)

  new make() =>
    None

  fun tag tests(test: PonyTest) =>
    test(_TestAdd)
    test(_TestSub)

class iso _TestAdd is UnitTest
```

```
  fun name(): String => "addition"

  fun apply(h: TestHelper) =>
    h.assert_eq[U32](4, 2 + 2)

class iso _TestSub is UnitTest
  fun name(): String => "subtraction"

  fun apply(h: TestHelper) =>
    h.assert_eq[U32](2, 4 - 2)
```

ponytest-example.pony

The make() constructor is not needed for this example. However, it allows for easy aggregation of tests (see below) so it is recommended that all test Mains provide it.

Main.create() is called only for program invocations on the current package. Main.make() is called during aggregation. If so desired extra code can be added to either of these constructors to perform additional tasks.

Test names

Tests are identified by names, which are used when printing test results and on the command line to select which tests to run. These names are independent of the names of the test classes in the Pony source code.

Arbitrary strings can be used for these names, but for large projects, it is strongly recommended to use a hierarchical naming scheme to make it easier to select groups of tests.

Aggregation

Often it is desirable to run a collection of unit tests from multiple different source files. For example, if several packages within a bundle each have their own unit tests it may be useful to run all tests for the bundle together.

This can be achieved by writing an aggregate test list class, which calls the list function for each package. The following is an example that aggregates the tests from packages `foo` and `bar`.

```
use "pony_test"
use foo = "foo"
use bar = "bar"

actor Main is TestList
  new create(env: Env) =>
    PonyTest(env, this)

  new make() =>
    None

  fun tag tests(test: PonyTest) =>
    foo.Main.make().tests(test)
    bar.Main.make().tests(test)
```

ponytest-aggregation.pony

Aggregate test classes may themselves be aggregated. Every test list class may contain any combination of its own tests and aggregated lists.

Long tests

Simple tests run within a single function. When that function exits, either returning or raising an error, the test is complete. This is not viable for tests that need to use actors.

Long tests allow for delayed completion. Any test can call `long_test()` on its `TestHelper` to indicate that it needs to keep running. When the test is finally complete it calls `complete()` on its `TestHelper`.

The `complete()` function takes a `Bool` parameter to specify whether the test was a success. If any asserts fail then the test will be considered a failure regardless of the value of this parameter. However, `complete()` must still be called.

Since failing tests may hang, a timeout must be specified for each long test. When the test function exits a timer is started with the specified timeout. If this timer fires before `complete()` is called the test is marked as a failure and the timeout is reported.

On a timeout, the `timed_out()` function is called on the unit test object. This should perform whatever test specific tidy up is required to allow the program to exit. There is no need to call `complete()` if a timeout occurs, although it is not an error to do so.

Note that the timeout is only relevant when a test hangs and would otherwise prevent the test program from completing. Setting a very long timeout on tests that should not be able to hang is perfectly acceptable and will not make the test take any longer if successful.

Timeouts should not be used as the standard method of detecting if a test has failed.

Exclusion groups

By default, all tests are run concurrently. This may be a problem for some tests, e.g. if they manipulate an external file or use a system resource. To fix this issue any number of tests may be put into an exclusion group.

No tests that are in the same exclusion group will be run concurrently.

Exclusion groups are identified by name, arbitrary strings may be used. Multiple exclusion groups may be used and tests in different groups may run concurrently. Tests that do not specify an exclusion group may be run concurrently with any other tests.

The command line option `--sequential` prevents any tests from running concurrently, regardless of exclusion groups. This is intended for debugging rather than standard use.

Tear down

Each unit test object may define a `tear_down()` function. This is called after the test has finished allowing the tearing down of any complex environment that had to be set up for the test.

The `tear_down()` function is called for each test regardless of whether it passed or failed. If a test times out tear_down() will be called after `timed_out()` returns.

When a test is in an exclusion group, the `tear_down()` call is considered part of the tests run. The next test in the exclusion group will not start until after `tear_down()` returns on the current test.

The test's `TestHelper` is handed to `tear_down()` and it is permitted to log messages and call assert functions during tear down.

Additional resources

You can learn more about PonyTest specifics by checking out the API documentation. There's also a testing section in the Pony Patterns book.

Testing with `PonyCheck`

PonyCheck is Pony's property based testing framework. It is designed to work seamlessly with PonyTest, Pony's unit testing framework. How is property based testing different than unit testing? Why does Pony include both?

In traditional unit testing, it is the duty and burden of the developer to provide and craft meaningful input examples for the unit under test (be it a class, a function or whatever) and check if some output conditions hold. This is a tedious and error-prone activity.

Property based testing leaves generation of test input samples to the testing engine which generates random examples taken from a description how to do so, so called `Generators`. The developer needs to define a `Generator` and describe the condition that should hold for each and every input sample.

Property based testing first came up as `QuickCheck` in Haskell. It has the nice property of automatically inferring `Generators` from the type of the property parameter, the test input sample.

PonyCheck is heavily inspired by QuickCheck and other great property based testing libraries, namely:

- Hypothesis
- Theft
- ScalaCheck

Usage

Writing property based tests in PonyCheck is done by implementing the trait `Property1`. A `Property1` needs to define a type parameter for the type of the input sample, a `Generator` and a property function. Here is a minimal example:

```
use "pony_check"

class _MyFirstProperty is Property1[String]
  fun name(): String =>
    "my_first_property"

  fun gen(): Generator[String] =>
    Generators.ascii()

  fun property(arg1: String, ph: PropertyHelper) =>
    ph.assert_eq[String](arg1, arg1)
```

ponycheck-usage.pony:2:12

A `Property1` needs a name for identification in test output. We created a `Generator` by using one of the many convenience factory methods and combinators defined in the `Generators` primitive and we used `PropertyHelper` to assert on a condition that should hold for all samples

Below are two classic list reverse properties from the QuickCheck paper adapted to Pony arrays:

```
use "pony_check"
use "collections"

class _ListReverseProperty is Property1[Array[USize]]
  fun name(): String => "list/reverse"

  fun gen(): Generator[Array[USize]] =>
    Generators.seq_of[USize, Array[USize]](Generators.usize())

  fun property(arg1: Array[USize], ph: PropertyHelper) =>
    ph.assert_array_eq[USize](arg1, arg1.reverse().reverse())

class _ListReverseOneProperty is Property1[Array[USize]]
  fun name(): String => "list/reverse/one"

  fun gen(): Generator[Array[USize]] =>
    Generators.seq_of[USize, Array[USize]](Generators.usize() where min = 1, max = 1)

  fun property(arg1: Array[USize], ph: PropertyHelper) =>
    ph.assert_array_eq[USize](arg1, arg1.reverse())
```

ponycheck-usage-quickcheck.pony

Integration with PonyTest

PonyCheck properties need to be executed. The test runner for PonyCheck is PonyTest. To integrate `Property1` into PonyTest, `Property1` needs to be wrapped inside a `Property1UnitTest` and passed to the PonyTest `apply` method as a regular PonyTest `UnitTest`:

It is also possible to integrate any number of properties directly into one `UnitTest` using the `PonyCheck.for_all` convenience function:

```
class _ListReverseProperties is UnitTest
  fun name(): String => "list/properties"

  fun apply(h: TestHelper) ? =>
    let gen1 = Generators.seq_of[USize, Array[USize]](Generators.usize())
    PonyCheck.for_all[Array[USize]](gen1, h)({
      (arg1: Array[USize], ph: PropertyHelper) =>
        ph.assert_array_eq[USize](arg1, arg1.reverse().reverse())
    })
    let gen2 = Generators.seq_of[USize, Array[USize]](1, Generators.usize())
    PonyCheck.for_all[Array[USize]](gen2, h)({
      (arg1: Array[USize], ph: PropertyHelper) =>
        ph.assert_array_eq[USize](arg1, arg1.reverse())
    })
```

ponycheck-ponytest-for-all.pony

Additional resources

You can learn more about PonyCheck specifics by checking out the API documentation. You can also find some example tests in ponyc GitHub repository.

To learn more about testing in Pony in general, there's a testing section in the Pony Patterns book which isn't specific to `PonyCheck`.

C-FFI

Pony supports integration with other native languages through the Foreign Function Interface (FFI). The FFI library provides a stable and portable API and high-level programming interface allowing Pony to integrate with native libraries easily.

Note that calling C (or other low-level languages) is inherently dangerous. C code fundamentally has access to all memory in the process and can change any of it, either deliberately or due to bugs. This is one of the language's most useful, but also most dangerous, features. Calling well written, bug-free, C code will have no ill effects on your program. However, calling buggy or malicious C code or calling C incorrectly can cause your Pony program to go wrong, including corrupting data and crashing. Consequently, all of the Pony guarantees regarding not crashing, memory safety and concurrent correctness can be voided by calling FFI functions.

Calling C from Pony

FFI is built into Pony and native libraries may be directly referenced in Pony code. There is no need to code or configure bindings, wrappers or interfaces.

Safely does it

It is VERY important that when calling FFI functions you MUST get the parameter and return types right. The compiler has no way to know what the native code expects and will just believe whatever you do. Errors here can cause invalid data to be passed to the FFI function or returned to Pony, which can lead to program crashes.

To help avoid bugs, Pony requires you to specify the type signatures of FFI functions in advance. While the compiler will trust that you specify the correct types in the signature, it will check the arguments you provide at each FFI call site against the declared signature. This means that you must get the types right only once, in the declaration. A declaration won't help you if the argument types the native code expects are different to what you think they are, but it will protect you against trivial mistakes and simple typos.

Here's an example of an FFI signature and call from the standard library:

```
use @_mkdir[I32](dir: Pointer[U8] tag) if windows
use @mkdir[I32](path: Pointer[U8] tag, mode: U32) if not windows

class val FilePath
  fun val mkdir(must_create: Bool = false): Bool =>
    // ...
      let r = ifdef windows then
        @_mkdir(element.cstring())
      else
        @mkdir(element.cstring(), 0x1FF)
      end
```

calling-c-file-path.pony

FFI functions have the @ symbol before its name, and FFI signatures are declared using the `use` command. The types specified here are considered authoritative, and any FFI calls that use different parameter types will result in a compile error.

The use @ command can take a condition just like other `use` commands. This is useful in this case, since the `_mkdir` function only exists in Windows.

If the name of the C function that you want to call is also a reserved keyword in Pony (such as `box`), you will need to wrap the name in double quotes (`@"box"`). If you forget to do so, your program will not compile.

An FFI signature is public to all Pony files inside the same package, so you only need to write them once.

C types

Many C functions require types that don't have an exact equivalent in Pony. A variety of features is provided for these.

For FFI functions that have no return value (i.e. they return `void` in C) the return value specified should be `None`.

In Pony, a String is an object with a header and fields, while in C a `char*` is simply a pointer to character data. The `.cstring()` function on String provides us with a valid pointer to hand to C. Our `mkdir` example above makes use of this for the first argument.

Pony classes and structs correspond directly to pointers to the class or struct in C.

For C pointers to simple types, such as U64, the Pony `Pointer[]` polymorphic type should be used, with a `tag` reference capability. To represent `void*` arguments, you should use the `Pointer[None] tag` type, which will allow you to pass a pointer to any type, including other pointers. This is needed to write declarations for certain POSIX functions, such as `memcpy`:

```
// The C type is void* memcpy(void *restrict dst, const void *restrict src, size_t n);
use @memcpy[Pointer[U8]](dst: Pointer[None] tag, src: Pointer[None] tag, n: USize)

// Now we can use memcpy with any Pointer type
let out: Pointer[Pointer[U8] tag] tag = // ...
let outlen: Pointer[U8] tag = // ...
let ptr: Pointer[U8] tag = // ...
let size: USize = // ...
// ...
@memcpy(out, addressof ptr, size.bitwidth() / 8)
@memcpy(outlen, addressof size, 1)
```

calling-c-memcpy.pony

When dealing with `void*` return types from C, it is good practice to try to narrow the type down to the most specific Pony type that you expect to receive. In the example above, we chose `Pointer[U8]` as the return type, since we can use such a pointer to construct Pony Arrays and Strings.

To pass pointers to values to C the `addressof` operator can be used (previously `&`), just like taking an address in C. This is done in the standard library to pass the address of a `U32` to an FFI function that takes a `int*` as an out parameter:

```
use @frexp[F64](value: F64, exponent: Pointer[U32])
// ...
var exponent: U32 = 0
var mantissa = @frexp(this, addressof exponent)
```

calling-c-addressof.pony

Get and Pass Pointers to FFI

If you want to receive a pointer to an opaque C type, using a pointer to a primitive can be useful:

```
use @XOpenDisplay[Pointer[_XDisplayHandle]](name: Pointer[U8] tag)
use @eglGetDisplay[Pointer[_EGLDisplayHandle]](disp: Pointer[_XDisplayHandle])

primitive _XDisplayHandle
primitive _EGLDisplayHandle

let x_dpy = @XOpenDisplay(Pointer[U8])
if x_dpy.is_null() then
  env.out.print("XOpenDisplay failed")
end

let e_dpy = @eglGetDisplay(x_dpy)
if e_dpy.is_null() then
  env.out.print("eglGetDisplay failed")
end
```

calling-c-pointer-to-opaque-c-type.pony

The above example would also work if we used `Pointer[None]` for all the pointer types. By using a pointer to a primitive, we are adding a level of type safety, as the compiler will ensure that we don't pass a pointer to any other type as a parameter to `eglGetDisplay`. It is important to note that these primitives should **not be used anywhere except as a type parameter** of `Pointer[]`, to avoid misuse.

Working with Structs: from Pony to C

Like we mentioned above, Pony classes and structs correspond directly to pointers to the class or struct in C. This means that in most cases we won't need to use the `addressof` operator when passing struct types to C. For example, let's imagine we want to use the `writev` function from Pony on Linux:

As you saw, a `IOVec` instance in Pony is equivalent to `struct iovec*`. In some cases, like the above example, it can be cumbersome to define a `struct` type in Pony if you only want to use it in a single place. You can also use a pointer to a tuple type as a shorthand for a struct: let's rework the above example:

In the example above, the type `Pointer[(Pointer[U8] tag, USize)] tag` is equivalent to the `IOVec` struct type we defined earlier. That is, *a struct type is equivalent to a pointer to a tuple type with the fields of the struct as elements, in the same order as the original struct type defined them.*

Can I pass struct types by value, instead of passing a pointer? Not at the moment. This is a known limitation of the current FFI system, but it is something the Pony team is interested in fixing. If you'd like to work on adding support for passing structs by value, contact us on the Zulip.

Working with Structs: from C to Pony

A common pattern in C is to pass a struct pointer to a function, and that function will fill in various values in the struct. To do this in Pony, you make a `struct` and then use a `NullablePointer`, which denotes a possibly-null type:

A `NullablePointer` type can only be used with `structs`, and is only intended for output parameters (like in the example above) or for return types from C. You don't need to use a `NullablePointer` if you are only passing a `struct` as a regular input parameter.

If you are using a C function that returns a struct, remember, that the C function needs to return a pointer to the struct. The following in Pony should be read as **returns a pointer to struct `Rect`**:

```
use @from_c[Rect]()

struct Rect
  var length: U16
  var width: U16
```

calling-c-from-c-struct.pony:1:5

As we saw earlier, you can also use a `Pointer[(U16, U16)]` as well. It is the equivalent to our `Rect`.

Can I return struct types by value, instead of passing a pointer? Not at the moment. This is a known limitation of the current FFI system, but it is something the Pony team is interested in fixing. If you'd like to work on adding support for returning structs by value, contact us on the Zulip.

Return-type Polymorphism

We mentioned before that you should use the `Pointer[None]` type in Pony when dealing with values of `void*` type in C. This is very useful for function parameters, but when we use `Pointer[None]` for the return type of a C function, we won't be able to access the value that the pointer points to. Let's imagine a generic list in C:

```c
struct List;

struct List* list_create();
void list_free(struct List* list);

void list_push(struct List* list, void *data);
void* list_pop(struct List* list);
```

Following the advice from previous sections, we can write the following Pony declarations:

```
use @list_create[Pointer[_List]]()
use @list_free[None](list: Pointer[_List])

use @list_push[None](list: Pointer[_List], data: Pointer[None])
use @list_pop[Pointer[None]](list: Pointer[_List])

primitive _List
```

calling-c-generic-list.pony

We can use these declarations to create lists of different types, and insert elements into them:

```
struct Point
  var x: U64 = 0
  var y: U64 = 0

let list_of_points = @list_create()
@list_push(list_of_points, NullablePointer[Point].create(Point))

let list_of_strings = @list_create()
@list_push(list_of_strings, "some data".cstring())
```

calling-c-different-types-of-lists.pony

We can also get elements out of the list, although we won't be able to do anything with them:

```
// Compiler error: couldn't find 'x' in 'Pointer'
let point_x = @list_pop(list_of_points)
point.x

// Compiler error: wanted Pointer[U8 val] ref^, got Pointer[None val] ref
let head = String.from_cstring(@list_pop(list_of_strings))
```

calling-c-access-list-entry-without-return-type.pony

We can fix this problem by adding an explicit return type when calling `list_pop`:

```
// OK
let point = @list_pop[Point](list_of_points)
let x_coord = point.x

// OK
let pointer = @list_pop[Pointer[U8]](list_of_strings)
let data = String.from_cstring(pointer)
```

calling-c-access-list-entry-with-explicit-return-type.pony

Note that the declaration for `list_pop` is still needed: if we don't add an explicit return type when calling `list_pop`, the default type will be the return type of the declaration.

When specifying a different return type for an FFI function, make sure that the new type is compatible with the type specified in the declaration.

Variadic C functions

Some C functions are variadic, that is, they can take a variable number of parameters. To interact with these functions, you should also specify that fact in the FFI signature:

In the example above, the compiler will type-check the first argument to `printf`, but will not be able to check any other argument, since it lacks the necessary type information. It is **very** important that you use `...` in the FFI signature if the corresponding C function is variadic: if you don't, the compiler might generate a program that is incorrect or crash on some platforms while appearing to work correctly on others.

FFI functions raising errors

Some FFI functions might raise Pony errors. Functions in existing C libraries are very unlikely to do this, but support libraries specifically written for use with Pony may well do.

FFI calls to functions that **might** raise an error **must** mark it as such by adding a ? after its declaration. The FFI call site must mark it as well. For example:

```
use @pony_os_send[USize](event: AsioEventID, buffer: Pointer[U8] tag, size: USize) ?
// ...
// May raise an error
@pony_os_send(_event, data.cpointer(), data.size()) ?
```

calling-c-ffi-functions-raising-errors.pony

If you're writing a C library that wants to raise a Pony error, you should do so using the `pony_error` function. Here's an example from the Pony runtime:

```
// In pony.h
PONY_API void pony_error();

// In socket.c
PONY_API size_t pony_os_send(asio_event_t* ev, const char* buf, size_t len)
{
  ssize_t sent = send(ev->fd, buf, len, 0);

  if(sent < 0)
  {
    if(errno == EWOULDBLOCK || errno == EAGAIN)
      return 0;

    pony_error();
  }

  return (size_t)sent;
}
```

A function that calls the `pony_error` function should only be called from inside a `try` block in Pony. If this is not done, the call to `pony_error` will result in a call to C's `abort` function, which will terminate the program.

Type signature compatibility

Since type signature declarations are scoped to a single Pony package, separate packages might define different FFI signatures for the same C function. In this case, as well as the case where you specify a different return type for an FFI call, the compiler will make sure that all calls and declarations are compatible with each other. Two functions are compatible if their arguments and return types are compatible. Two types are compatible with each other if they have the same ABI size and they can be safely cast to each other. The compiler allows the following type casts:

- Any `struct` type can be cast to any other `struct` (as they are both pointer types)
- Pointers and integers can be cast to each other.

Consider the following example:

```
// In library lib_a
use @memcmp[I32](dst: Pointer[None] tag, src: Pointer[None] tag, len: USize)

// In library lib_b
use @memcmp[I32](dst: Pointer[None] tag, src: USize, len: U64)
```

calling-c-type-signature-compatibility.pony

These two declarations have different types for the `src` and `len` parameters. In the case of `src`, the types are compatible since an integer can be cast as a pointer, and vice versa. For `len`, the types will not be compatible on 32 bit platforms, where `USize` is equivalent to `U32`. It is important to take the rules around casting into account when writing type declarations in libraries that will be used by others, as it will avoid any compatibility problems with other libraries.

Calling FFI functions from Interfaces or Traits

We mentioned in the previous section that FFI declarations are scoped to a single Pony package, with separate packages possibly defining different FFI signatures for the same C function. Importing an external package will not import any FFI declarations, since any name collisions would produce multiple declarations for the same C function name, and thus deciding which declaration to use would be ambiguous.

Given the above fact, if you define any default methods (or behaviors) in an interface or trait, you will not be able to perform an FFI call from them. For example, the code below will fail to compile:

```
use @printf[I32](fmt: Pointer[None] tag, ...)

trait Foo
  fun apply() =>
    // Error: Can't call an FFI function in a default method or behavior
    @printf("Hello from trait Foo\n".cstring())

actor Main is Foo
  new create(env: Env) =>
    this.apply()
```

calling-c-default-method-in-trait.pony

If the trait `Foo` above was part of the public API of a package, allowing its `apply` method to perform an FFI call would render `Foo` unusable for any external users, given that the declaration for `printf` would not be in scope.

Fortunately, avoiding this limitation is relatively painless. Whenever you need to call an FFI function from a default method implementation, consider moving said function to a separate type:

```
use @printf[I32](fmt: Pointer[None] tag, ...)

trait Foo
  fun apply() =>
    // OK
    Printf("Hello from trait Foo\n")

primitive Printf
```

```
  fun apply(str: String) =>
    @printf(str.cstring())

actor Main is Foo
  new create(env: Env) =>
    this.apply()
```

calling-c-default-method-in-primitive.pony

By making the change above, we avoid exposing the call to `printf` to any consumers of our trait, thus making it usable by external users.

Linking to C Libraries

If Pony code calls FFI functions, then those functions, or rather the libraries containing them, must be linked into the Pony program.

Use for external libraries

To link an external library to Pony code another variant of the use command is used. The `lib` specifier is used to tell the compiler you want to link to a library. For example:

```
use "lib:foo"
```

linking-c-use-lib-foo.pony

As with other `use` commands a condition may be specified. This is particularly useful when the library has slightly different names on different platforms.

Here's a real example from the standard library:

```
use "path:/usr/local/opt/libressl/lib" if osx
use "lib:ssl" if not windows
use "lib:crypto" if not windows
use "lib:libssl-32" if windows
use "lib:libcrypto-32" if windows

use @SSL_load_error_strings[None]()
use @SSL_library_init[I32]()

primitive _SSLInit
  """
  This initialises SSL when the program begins.
  """
  fun _init() =>
    @SSL_load_error_strings()
    @SSL_library_init()
```

linking-c-use-with-condition.pony

On Windows, we use the libraries `libssl-32` and `libcrypto-32` and on other platforms we use `ssl` and `crypto`. These contain the FFI functions `SSL_library_init` and `SSL_load_error_strings` (amongst others).

By default the Pony compiler will look for the libraries to link in the standard places, however, that is defined on the build platform. However, it may be necessary to look in extra places.

The `use "path:..."` command allows this. The specified path is added to the library search paths for the remainder of the current file. The example above uses this to add the path `/usr/local/opt/libressl/lib` for MacOS. This is required because the library is provided by brew, which installs things outside the standard library search paths.

If you are integrating with existing libraries, that is all you need to do.

C ABI

The FFI support in Pony uses the C application binary interface (ABI) to interface with native code. The C ABI is a calling convention, one of many, that allow objects from different programming languages to be used together.

Writing a C library for Pony

Writing your own C library for use by Pony is almost as easy as using existing libraries.

Let's look at a complete example of a C function we may wish to provide to Pony. Let's consider a pure Pony implementation of a Jump Consistent Hash:

```
// Jump consistent hashing in Pony, with an inline pseudo random generator
// https://arxiv.org/abs/1406.2294

fun jch(key: U64, buckets: U32): I32 =>
  var k = key
  var b = I64(0)
  var j = I64(0)

  while j < buckets.i64() do
    b = j
    k = (k * 2862933555777941757) + 1
    j = ((b + 1).f64() * (I64(1 << 31).f64() / ((k >> 33) + 1).f64())).i64()
  end
```

c-abi-jump-consistent-hashing.pony:5:18

Let's say we wish to compare the pure Pony performance to an existing C function with the following header:

```
#ifndef __JCH_H_
#define __JCH_H_

extern "C"
{
  int32_t jch_chash(uint64_t key, uint32_t num_buckets);
}

#endif
```

Note the use of `extern "C"`. If the library is built as C++ then we need to tell the compiler not to mangle the function name, otherwise, Pony won't be able to find it. For libraries built as C, this is not needed, of course.

The implementation of the previous header would be something like:

```c
#include <stdint.h>

// A fast, minimal memory, consistent hash algorithm
// https://arxiv.org/abs/1406.2294
int32_t jch_chash(uint64_t key, uint32_t num_buckets)
{
  int b = -1;
  uint64_t j = 0;

  do {
    b = j;
    key = key * 2862933555777941757ULL + 1;
    j = (b + 1) * ((double)(1LL << 31) / ((double)(key >> 33) + 1));
  } while(j < num_buckets);

  return (int32_t)b;
}
```

We need to compile the native code to a shared library. This example is for MacOS. The exact details may vary on other platforms.

```
clang -fPIC -Wall -Wextra -O3 -g -MM jch.c >jch.d
clang -fPIC -Wall -Wextra -O3 -g  -c -o jch.o jch.c
clang -shared -lm -o libjch.dylib jch.o
```

The Pony code to use this new C library is just like the code we've already seen for using C libraries.

```
"""
This is an example of Pony integrating with native code via the built-in FFI
support
"""

use "lib:jch"
use "collections"
use @jch_chash[I32](hash: U64, bucket_size: U32)

actor Main
  var _env: Env

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

    let bucket_size: U32 = 1000000

    _env.out.print("C implementation:")
    for i in Range[U64](1, 20) do
      let hash = @jch_chash(i, bucket_size)
      _env.out.print(i.string() + ": " + hash.string())
    end

    _env.out.print("Pony implementation:")
    for i in Range[U64](1, 20) do
```

```
      let hash = jch(i, bucket_size)
      _env.out.print(i.string() + ": " + hash.string())
    end

  fun jch(key: U64, buckets: U32): I32 =>
    var k = key
    var b = I64(0)
    var j = I64(0)

    while j < buckets.i64() do
      b = j
      k = (k * 2862933555777941757) + 1
      j = ((b + 1).f64() * (I64(1 << 31).f64() / ((k >> 33) + 1).f64())).i64()
    end

    b.i32()
```

c-abi-pony-use-native-jump-consistent-hashing-c-implementation.pony

We can now use ponyc to compile a native executable integrating Pony and our C library. And that's all we need to do.

Callbacks

Some C APIs let the programmer specify functions that should be called to do pieces of work. For example, the SQLite API has a function called `sqlite3_exec` that executes an SQL statement and calls a function given by the programmer on each row returned by that statement. The functions that are supplied by the programmer are known as "callback functions". Some specific Pony functions can be passed as callback functions.

Bare functions

Classic Pony functions have a receiver, which acts as an implicit argument to the function. Because of this, classic functions can't be used as callbacks with many C APIs. Instead, you can use *bare functions*, which are functions with no receiver.

You can define a bare function by prefixing the function name with the @ symbol.

```
class C
  fun @callback() =>
    ...
```

c-ffi-callbacks-bare-functions.pony

The function can then be passed as a callback to a C API with the `addressof` operator.

```
@setup_callback(addressof C.callback)
```

c-ffi-callbacks-bare-functions-pass-to-c-api.pony

Note that it is possible to use an object reference instead of a type as the left-hand side of the method access.

Since bare methods have no receiver, they cannot reference the `this` identifier in their body (either explicitly or implicitly through field access), cannot use `this` viewpoint adapted types, and cannot specify a receiver capability.

Bare lambdas

Bare lambdas are special lambdas defining bare functions. A bare lambda or bare lambda type is specified using the same syntax as other lambda types, with the small variation that it is prefixed with the @ symbol. The underlying value of a bare lambda is equivalent to a C function pointer, which means that a bare lambda can be directly passed as a callback to a C function. The partial application of a bare method yields a bare lambda.

```
let callback = @{() => ... }
@setup_callback(callback)
```

c-ffi-callbacks-bare-lambda-callback.pony

Bare lambdas can also be used to define structures containing function pointers. For example:

```
struct S
  var fun_ptr: @{()}
```

c-ffi-callbacks-bare-lambda-struct.pony

This Pony structure is equivalent to the following C structure:

```
struct S
{
  void(*fun_ptr)();
};
```

In the same vein as bare functions, bare lambdas cannot specify captures, cannot use `this` neither as an identifier nor as a type, and cannot specify a receiver capability. In addition, a bare lambda object always has a `val` capability.

Classic lambda types and bare lambda types can never be subtypes of each other.

An example

Consider SQLite, mentioned earlier. When the client code calls `sqlite3_exec`, an SQL query is executed against a database, and the callback function is called for each row returned by the SQL statement. Here's the signature for `sqlite3_exec`:

```
typedef int (*sqlite3_callback)(void*,int,char**, char**);

...

SQLITE_API int SQLITE_STDCALL sqlite3_exec(
sqlite3 *db,                /* The database on which the SQL executes */
const char *zSql,           /* The SQL to be executed */
sqlite3_callback xCallback, /* Invoke this callback routine */
void *pArg,                 /* First argument to xCallback() */
char **pzErrMsg             /* Write error messages here */
)
{
  ...
  xCallback(pArg, nCol, azVals, azCols)
  ...
}
```

`sqlite3_callback` is the type of the callback function that will be called by `sqlite3_exec` for each row returned by the `sql` statement. The first argument to the callback function is the pointer pArg that was passed to sqlite3_exec, the second argument is the number of columns in the row being processed, the third argument is data for each column, and the fourth argument is the name of each column.

Here's the skeleton of some Pony code that uses sqlite3_exec to query an SQLite database, with examples of both the bare method way and the bare lambda way:

```
use @sqlite3_exec[I32](db: Pointer[None] tag, sql: Pointer[U8] tag,
  callback: Pointer[None], data: Pointer[None], err_msg: Pointer[Pointer[U8] tag] tag)

class SQLiteClient
  fun client_code() =>
    ...
    @sqlite3_exec(db, sql.cstring(), addressof this.method_callback,
                  this, addressof zErrMsg)
    ...

  fun @method_callback(client: SQLiteClient, argc: I32,
    argv: Pointer[Pointer[U8]], azColName: Pointer[Pointer[U8]]): I32
  =>
    ...
```

c-ffi-callbacks-sqlite3-callback.pony

```
use @sqlite3_exec[I32](db: Pointer[None] tag, sql: Pointer[U8] tag,
  callback: Pointer[None], data: Pointer[None], err_msg: Pointer[Pointer[U8] tag] tag)

class SQLiteClient
  fun client_code() =>
    ...
    let lambda_callback =
      @{(client: SQLiteClient, argc: I32, argv: Pointer[Pointer[U8]],
        azColName: Pointer[Pointer[U8]]): I32
      =>
        ...
      }

    @sqlite3_exec(db, sql.cstring(), lambda_callback, this,
                  addressof zErrMsg)
    ...
```

c-ffi-callbacks-sqlite3-callback-2.pony

Focusing on the callback-related parts, the callback function is passed using addressof this.method_callback (resp. by directly passing the bare lambda) as the third argument to sqlite3_exec. The fourth argument is this, which will end up being the first argument when the callback function is called. The callback function is called in sqlite3_exec by the call to xCallback.

Gotchas

Every programming language has gotchas. Those "what the heck" moments that make us all laugh when someone does a presentation on them. They often shoot to the top of sites like Hacker News and Reddit. It's all in good fun, except, it isn't. Each of those gotchas and the laughs we get from them, hide someone's pain. This chapter covers some common Pony gotchas that new Pony programmers often stumble across with painful results. Probably the best way to approach this chapter is to imagine each section has a giant flashing "DO NOT DO THIS" sign.

Divide by Zero

What's 1 divided by 0? How about 10 divided by 0? What is the result you get in your favorite programming language?

In math, divide by zero is undefined. There is no answer to that question as the expression 1/0 has no meaning. In many programming languages, the answer is a runtime exception that the user has to handle. In Pony, things are a bit different.

Divide by zero in Pony

In Pony, *integer division by zero results in zero*. That's right,

```
let x = I64(1) / I64(0)
```

divide-by-zero.pony:3:3

results in `0` being assigned to `x`. Baffling right? Well, yes and no. From a mathematical standpoint, it is very much baffling. From a practical standpoint, it is very much not.

While Pony has Partial division:

```
let x =
  try
    I64(1) /? I64(0)
  else
    // handle division by zero
  end
```

divide-by-zero-partial.pony

Defining division as partial leads to code littered with `try`s attempting to deal with the possibility of division by zero. Even if you had asserted that your denominator was not zero, you'd still need to protect against divide by zero because, at this time, the compiler can't detect that value dependent typing.

Pony also offers Unsafe Division, which declares division by zero as undefined, as in C:

```
// the value of x is undefined
let x = I64(1) /~ I64(0)
```

divide-by-zero-unsafe.pony

But declaring this case as undefined does not help us out here. As a programmer you'd still need to guard that case in order to not poison your program with undefined values or risking terminating your program with a `SIGFPE`. So, in order to maintain a practical API and avoid undefined behaviour, *normal* division on integers in Pony is defined to be `0`. To avoid `0`s silently creeping through your divisions, use Partial or Checked Division.

Divide by zero on floating points

In conformance with IEEE 754, *floating point division by zero results in `inf` or `-inf`*, depending on the sign of the numerator.

If you can assert that your denominator cannot be `0`, it is possible to use Unsafe Division to gain some performance:

```
    let x = F64(1.5) /~ F64(0.5)
```

divide-by-zero-floats.pony:4:4

Garbage Collection

There's a common GC anti-pattern that many new Pony programmers accidentally stumble across. Usually, this results in a skyrocketing of memory usage in their test program and questions on Zulip as to why Pony isn't working correctly. It is, in fact, working correctly, albeit not obviously.

Garbage Collection in the world at large

Garbage collection, in most languages, can run at any time. Your program can be paused so that memory can be freed up. This sucks if you want predictable completion of sections of code. Most of the time, your function will finish in less than a millisecond, but every now and then, it's paused during execution to GC. There are advantages to this approach. Whenever you run low on memory, the GC can attempt to free some memory and get you more. In general, this is how people expect Pony's garbage collector to work. As you might guess though, it doesn't work that way.

Garbage Collection in Pony

Garbage collection is never attempted on any actor while it is executing a behavior. This gives you very predictable performance when executing behaviors, but also makes it easy to grab way more memory than you intend to. Let's take a look at how that can happen via the "long-running behavior problem."

Long running behaviors and memory

Here's a typical "I'm learning Pony" program:

```
use "collections"

actor Main
  new create(env: Env) =>
    for i in Range(1, 2_000_000) do
      ... something that uses up heap ...
    end
```

garbage-collection.pony

This program will never garbage collect before exiting. `create` is run as a behavior on actors, which means that no garbage collection will occur while it's running. Long loops in behaviors are a good way to exhaust memory. Don't do it. If you want to execute something in such a fashion, use a Timer.

Scheduling

The Pony scheduler is not preemptive. This means that your actor has to yield control of the scheduler thread in order for another actor to execute. The normal way to do this is for your behavior to end. If your behavior doesn't end, you will continue to monopolize a scheduler thread and bad things will happen.

FFI and monopolizing the scheduler

An easy way to monopolize a scheduler thread is to use the FFI facilities of Pony to kick off code that doesn't return for an extended period of time. You do not want to do this. Do not call FFI code that doesn't return in a reasonable amount of time.

Long running behaviors

Another way to monopolize a scheduler thread is to write a behavior that never exits or takes a really long time to exit.

```
be bad_citizen() =>
  while true do
    _env.out.print("Never gonna give you up. Really gonna make you cry")
  end
```

scheduling.pony

That is some seriously bad citizen code that will hog a scheduler thread forever. Call that behavior a few times and your program will grind to a halt. If you find yourself writing code with loops that will run for a long time, stop and rethink your design. Take a look at the Timer class from the standard library. Combine that together with a counter in your class and you can execute the same behavior repeatedly while yielding your scheduler thread to other actors.

Function Call Side Effects

Consider the following code:

```
class Foo
  fun fn(x: U64) => None

actor Main
  new create(env: Env) =>
    var x: U64 = 0
    try foo()?.fn(x = 42) end
    env.out.print(x.string())

  fun foo(): Foo ? => error
```

function-call-side-effects.pony

What do you think it will print? Probably `0` right? Or maybe you realized this code is in the gotchas section so it must be `42`. If you went with `42`, you'd be right. Why?

Expressions for arguments in function calls are evaluated before the expression for the function receiver. The use of assignment expressions like `x = 42` is quite rare so we don't think many folks will be bitten by this. However, it's definitely something you want to be aware of. Also remember that if `fn` were to be called, it would be called with `0` (the result of the assignment expression).

Recursion

Recursive functions in Pony can cause many problems. Every function call in a program adds a frame on the system call stack, which is bounded. If the stack grows too big it will overflow, usually crashing the program. This is an out-of-memory type of error and it cannot be prevented by the guarantees offered by Pony.

If you have a heavy recursive algorithm, you must take some precautions in your code to avoid stack overflows. Most recursive functions can be easily transformed into tail-recursive function which are less problematic. A tail-recursive function is a function in which the recursive call is the last instruction of the function. Here is an example with a factorial function:

```
fun recursive_factorial(x: U32): U32 =>
  if x == 0 then
    1
  else
    x * recursive_factorial(x - 1)
  end

fun tail_recursive_factorial(x: U32, y: U32): U32 =>
  if x == 0 then
    y
  else
    tail_recursive_factorial(x - 1, x * y)
  end
```

recursion.pony:6:18

The compiler can optimise a tail-recursive function to a loop, completely avoiding call stack growth. Note that this is an *optimisation* which is only performed in release builds (i.e. builds without the `-d` flag passed to ponyc.) If you need to avoid stack growth in debug builds as well then you have to write your function as a loop manually.

If the tail-recursive version of your algorithm isn't practical to implement, there are other ways to control stack growth depending on your algorithm. For example, you can implement your algorithm using an explicit stack data structure instead of implicitly using the call stack to store data.

Note that light recursion usually doesn't cause problems. Unless your amount of recursive calls is in the hundreds, you're unlikely to encounter this problem.

Where Next?

Hey, congratulations! You've made it to the end of the Pony tutorial. So, what do you do next? Well, there's actually a bit more here. Check out the appendices as they have some useful information that doesn't fit else. After that, here are a few resources that you can look into.

"Learn" section the Pony website

If you haven't already visited it, the learn section of the Pony website has a lot of good content to help you get started with Pony. In particular, materials to help you grapple with reference capabilities.

"Learn" on the Pony website

Planet Pony

We don't have an automatic blog aggregator but wish we did. In the meantime, we have a hand-curated list of videos, blog posts etc that would be of interest to members of the Pony community. Just beware when you are checking out older posts, it's quite possible that their examples no longer compile as Pony development is currently moving very quickly.

Planet Pony

Pony Patterns

One of the hardest parts of learning a new language is figuring out how to do various different "everyday" tasks. We have a cookbook style book called "Pony Patterns" that enumerates a variety of problems you might encounter and idiomatic ways to solve those problems. The amount of content is still somewhat small but growing all the time.

Pony Patterns

Standard Library Documentation

You are going to need to learn the standard library. Some of us prefer to open the source code and explore. If you prefer an online experience, we maintain a version of the standard library documentation online. And you don't have to worry about it going out of date as it is updated on every commit to the ponyc main branch.

Standard Library Documentation

Pony Zulip

Pony Zulip

Pony Virtual Users' Group

The Pony Virtual Users' Group has occasional presentations that you can attend "in person" or catch later via the recorded video. Join our Zulip community to stay up to date on upcoming meetings. All the previous videos are available via the Main Ponylang Vimeo.

A final word

We're immensely happy that you have taken the time to start learning Pony. It's still a new and immature language with plenty of sharp pointy edges. You are going to get frustrated at times. Don't worry, it has happened to all of us. Drop by one of our support channels and someone will try to lend a hand. We want you to succeed. The more people who succeed with Pony, the more the community grows and the better it is for all of us.

Welcome to the community! Have fun!

Appendices

Welcome to the appendices; the land of misshapen and forgotten documentation. Ok, not really forgotten just… 'lesser' sounds wrong. Some of this material could get some loving and be promoted to a full chapter, some are always going to be an appendix, some might be worthy of a short book unto itself. Right now though it lives here, have a look through. You'll find a lexicon of standard Pony terminology, a symbol lookup cheat sheet that can help you locate documentation on all our funny symbols like ^, ! and much more.

`PONYPATH`

When searching for Pony packages, `ponyc` checks both the installation directory (where the standard libraries reside) and any directories listed in the optional environment variable `PONYPATH`.

Adding to `PONYPATH`

Assuming you just placed new Pony code under a directory called `pony` in your home directory here is how to inform `ponyc` that the directory contains Pony code via adding it to `PONYPATH`.

Unix/Mac

Edit/add the `rc` file corresponding to your chosen shell (`echo $SHELL` will tell you what shell you are running). For example, if using bash, add the following to your `~/.bashrc`:

```
export PONYPATH=$PONYPATH:$HOME/pony
```

(Then run `source ~/.bashrc` to add this variable to a running session. New terminal session will automatically source `~/.bashrc`.)

Windows

1. Create folder at `C:\Users\<yourusername>\pony`.
2. Right click on "Start" and click on "Control Panel". Select "System and Security", then click on "System".
3. From the menu on the left, select the "Advanced systems settings".
4. Click the "Environment Variables" button at the bottom.
5. Click "New" from the "User variables" section.
6. Type `PONYPATH` into the "Variable name" field.
7. Type `%PONYPATH%;%USERPROFILE%\pony` into the "Variable value" field.
8. Click OK.

You can also add to `PONYPATH` from the command prompt via:

```
setx PONYPATH %PONYPATH%;%USERPROFILE%\pony
```

Lexicon

Words are hard. We can all be saying the same thing but do we *mean* the same thing? It's tough to know. Hopefully, this lexicon helps a little.

Terminology

Braces: { }. Synonymous with curly brackets.

Brackets: This term is ambiguous. In the UK it usually means ( ) in the US is usually means [ ]. It should, therefore, be avoided for use for either of these. Can be used as a general term for any unspecified grouping punctuation, including { }.

Compatible type: Two types are compatible if there can be any single object which is an instance of both types. Note that a suitable type for the single object does not have to have been defined, as long as it could be. For example, any two traits are compatible because a class could be defined that provides both of them, even if such a class has not been defined. Conversely, no two classes can ever be compatible because no object can be an instance of both.

Compound type: A type combining multiple other types, i.e. union, intersection, and tuple. Opposite of a single type.

Concrete type: An actor, class or primitive.

Curly brackets: { }. Synonymous with braces.

Declaration and **definition**: synonyms for each other, we do not draw the C distinction between forward declarations and full definitions.

Default method body: Method body defined in a trait and optionally used by concrete types.

Entity: Top level definition within a file, i.e. alias, trait, actor, class, primitive.

Explicit type: An actor, class or primitive.

Member: Method or field.

Method: Something callable on a concrete type/object. Function, behaviour or constructor.

Override: When a concrete type has its own body for a method with a default body provided by a trait.

Parentheses: ( ). Synonymous with round brackets.

Provide: An entity's usage of traits and the methods they contain. Equivalent to implements or inherits from.

Round brackets: ( ). Synonymous with parentheses.

Single type: Any type which is not defined as a collection of other types. Actors, classes, primitives, traits and structural types are all single types. Opposite of a compound type.

Square brackets: [ ]

Trait clash: A trait clashes with another type if it contains a method with the same name, but incompatible signature as a method in the other type. A clashing trait is incompatible with the other type. Traits can clash with actors, classes, primitives, intersections, structural types and other traits.

Symbol Lookup Cheat Sheet

Pony, like just about any other programming language, has plenty of odd symbols that make up its syntax. If you don't remember what one means, it can be hard to search for them. Below you'll find a table with various Pony symbols and what you should search the tutorial for in order to learn more about the symbol.

Symbol	Search Keywords
`!`	Alias
`->`	Arrow type, viewpoint
`.>`	Chaining
`^`	Ephemeral
`@`	FFI
`&`	Intersection
`=>`	Match arrow
`~`	Partial application
`?`	Partial function
`'`	Prime

Symbol	Search Keywords
`<:`	Subtype

Here is a more elaborate explanation of Pony's use of special characters: (a line with (2) or (3) means an alternate usage of the symbol of the previous line)

Symbol	Usage
`,`	to separate parameters in a function signature, or the items of a tuple
`.`	(1) to call a field or a function on a variable (field access or method call)
	(2) to qualify a type/method with its package name
`.>`	to call a method on an object and return the receiver (chaining)
`'`	used as alternative name in parameters (prime)
`"`	to delineate a literal string
`"""`	to delineate a documentation string
`(`	(1) start of line: start of a tuple
	(2) middle of line: method call
`()`	(1) parentheses, for function or behavior parameters
	(2) making a tuple (values separated by `,`)
	(3) making an enumeration (values separated by |)
`[`	(1) start of line: start of an array literal
	(2) middle of line: generic formal parameters
`[]`	(1) to indicate a generic type, for example `Range[U64]`
	(2) to indicate the return type of an FFI function call
`{}`	a function type
`:`	(1) after a variable: is followed by the type name
	(2) to indicate a function return type
	(3) a type constraint
`;`	only used to separate expressions on the same line
`=`	(1) (destructive) assignment
	(2) in: use alias = package name
	(3) supply default argument for method
	(4) supply default type for generics
`!`	(1) boolean negation
	(2) a type that is an alias of another type
`?`	(1) partial functions
	(2) a call to a C function that could raise an error
`-`	(1) start of line: unary negation
	(2) middle of line: subtraction
`_`	(1) to indicate a private variable, constructor, function, behavior
	(2) to ignore a tuple item in a pattern match
`~`	partial application
`^`	an ephemeral type
|	(1) separates the types in an enumeration (the value can be any of these types)
	(2) starts a branch in a match
`&`	(1) separates the types in a complex type (the value is of all of these types)
	(2) intersection
`@`	FFI call
`//`	comments
`/* */`	multi-line or block comments

Symbol	Usage
`=>`	(1) start of a function body (2) starts the code of a matching branch
`->`	(1) arrow type (2) viewpoint
`._i`	where `i = 1,2,…` means the item at position i in the tuple
`<:`	“is a subtype of” or “can be substituted for”

Keywords

This listing explains the usage of every Pony keyword.

Keyword	Usage
`actor`	defines an actor
`as`	conversion of a value to another Type (can raise an error)
`be`	behavior, executed asynchronously
`box`	default reference capability – object is readable, but not writable
`break`	to step out of a loop statement
`class`	defines a class
`compile_error`	will provoke a compile error
`compile_intrinsic`	implementation is written in C and not available as Pony code
`continue`	continues a loop with the next iteration
`consume`	move a value to a new variable, leaving the original variable empty
`digestof`	create a `USize` value that summarizes the Pony object, similar to a Java object’s `hashCode()` value.
`do`	loop statement, or after a with statement
`else`	conditional statement in if, for, while, repeat, try (as a catch block), match
`elseif`	conditional statement, also used with `ifdef`
`embed`	embed a class as a field of another class
`end`	ending of: `if then`, `ifdef`, `while do`, `for in`, `repeat until`, `try`, `object`, `recover`, `match`
`error`	raises an error
`for`	loop statement
`fun`	define a function, executed synchronously
`if`	(1) conditional statement (2) to define a guard in a pattern match
`ifdef`	when defining a build flag at compile time: ponyc –D “foo”
`iftype`	type conditional statement `iftype A <: B` checks if `A` is a subtype of `B`
`in`	used in a for in - loop statement
`interface`	used in structural subtyping
`is`	(1) used in nominal subtyping (2) in type aliasing (3) identity comparison
`isnt`	negative identity comparison
`iso`	reference capability – read and write uniqueness
`let`	declaration of immutable variable: you can’t rebind this name to a new value
`match`	pattern matching
`new`	constructor
`not`	logical negation

Keyword	Usage
`object`	to make an object literal
`primitive`	declares a primitive type
`recover`	removes the reference capability of a variable
`ref`	reference capability – object (on which function is called) is mutable
`repeat`	loop statement
`return`	to return early from a function
`tag`	reference capability – neither readable nor writeable, only object identity
`then`	(1) in if conditional statement
	(2) as a (finally) block in try
`this`	the current object
`trait`	used in nominal subtyping: `class Foo is TraitName`
`trn`	reference capability – write uniqueness, no other actor can write to the object
`try`	error handling
`type`	to declare a type alias
`until`	loop statement
`use`	(1) using a package
	(2) using an external library foo: use "lib:foo"
	(3) declaration of an FFI signature
	(4) add a search path for external libraries: `use "path:/usr/local/lib"`
`var`	declaration of mutable variable: you can rebind this name to a new value
`val`	reference capability – globally immutable object
`where`	when specifying named arguments
`while`	loop statement
`with`	ensure disposal of an object

Examples

Small *how do I* examples for Pony. These will eventually find another home. Until then, they live here.

Enumeration with values

```
primitive Black fun apply(): U32 => 0xFF000000
primitive Red   fun apply(): U32 => 0xFFFF0000
```

appendices-examples-enumeration-with-values.pony:4:5

Enumeration with values with namespace

```
primitive Colours
  fun black(): U32 => 0xFF000000
  fun red(): U32 => 0xFFFF0000
```

appendices-examples-enumeration-with-values-with-namespace.pony:4:6

Enumeration which can be iterated

```
primitive Black
primitive Blue
primitive Red
primitive Yellow
```

```
type Colour is (Black | Blue | Red | Yellow)

primitive ColourList
  fun tag apply(): Array[Colour] =>
    [Black; Blue; Red; Yellow]

for colour in ColourList().values() do
end
```

appendices-examples-iterable-enumerations.pony

Pass an Array of values to FFI

```
use @eglChooseConfig[U32](disp: Pointer[_EGLDisplayHandle], attrs: Pointer[U16] tag,
  config: Pointer[_EGLConfigHandle], config_size: U32, num_config: Pointer[U32])

primitive _EGLConfigHandle
let a = Array[U16](8)
a.push(0x3040)
a.push(0x4)
a.push(0x3033)
a.push(0x4)
a.push(0x3022)
a.push(0x8)
a.push(0x3023)
a.push(0x8)
a.push(0x3024)
let config = Pointer[_EGLConfigHandle]
if @eglChooseConfig(e_dpy, a.cpointer(), config, U32(1), Pointer[U32]) == 0 then
    env.out.print("eglChooseConfig failed")
end
```

appendices-examples-pass-array-of-values-to-ffi.pony

How to access command line arguments

```
actor Main
  new create(env: Env) =>
    // The no of arguments
    env.out.print(env.args.size().string())
    for value in env.args.values() do
      env.out.print(value)
    end
    // Access the arguments the first one will always be the application name
    try env.out.print(env.args(0)?) end
```

appendices-examples-access-command-line-arguments.pony

How to use the `cli` package to parse command line arguments

```
use "cli"

actor Main
```

```
  new create(env: Env) =>
    let command_spec =
      try
        CommandSpec.leaf(
          "pony-embed",
          "sample program",
          [ OptionSpec.string("output", "output filename", 'o') ],
          [ ArgSpec.string("input", "source of input" where default' = "-") ]
        )? .> add_help()?
      else
        env.exitcode(1)
        return
      end
    let command =
      match CommandParser(command_spec).parse(env.args, env.vars)
      | let c: Command => c
      | let ch: CommandHelp =>
        ch.print_help(env.out)
        env.exitcode(0)
        return
      | let se: SyntaxError =>
        env.err.print(se.string())
        env.exitcode(1)
        return
      end
    let input_source = command.arg("input").string()
    let output_filename = command.option("output").string()
    env.out.print("Loading data from " + input_source + ". Writing output to " + output_fil
    // ...
```

appendices-examples-use-cli-package-to-parse-command-line-arguments.pony

How to write tests

Create a test.pony file

```
use "pony_test"

actor Main is TestList
  new create(env: Env) => PonyTest(env, this)
  new make() => None

  fun tag tests(test: PonyTest) =>
    test(_TestAddition)

class iso _TestAddition is UnitTest
  """
  Adding 2 numbers
  """
  fun name(): String => "u32/add"

  fun apply(h: TestHelper) =>
    h.assert_eq[U32](2 + 2, 4)
```

appendices-examples-write-tests.pony

Some assertions you can make with `TestHelper` are

```
fun tag log(msg: String, verbose: Bool = false)
be fail() =>
be assert_failed(msg: String) =>
fun tag assert_true(actual: Bool, msg: String = "") ?
fun tag expect_true(actual: Bool, msg: String = ""): Bool
fun tag assert_false(actual: Bool, msg: String = "") ?
fun tag expect_false(actual: Bool, msg: String = ""): Bool
fun tag assert_error(test: ITest, msg: String = "") ?
fun tag expect_error(test: ITest box, msg: String = ""): Bool
fun tag assert_is (expect: Any, actual: Any, msg: String = "") ?
fun tag expect_is (expect: Any, actual: Any, msg: String = ""): Bool
fun tag assert_eq[A: (Equatable[A] #read & Stringable)]
  (expect: A, actual: A, msg: String = "") ?
fun tag expect_eq[A: (Equatable[A] #read & Stringable)]
  (expect: A, actual: A, msg: String = ""): Bool
```

appendices-examples-test-helper.pony

Operator overloading (easy for copy and paste)

```
fun add(other: A): A
fun sub(other: A): A
fun mul(other: A): A
fun div(other: A): A
fun rem(other: A): A
fun mod(other: A): A
fun eq(other: A): Bool
fun ne(other: A): Bool
fun lt(other: A): Bool
fun le(other: A): Bool
fun ge(other: A): Bool
fun gt(other: A): Bool
fun shl(other: A): A
fun shr(other: A): A
fun op_and(other:A): A
fun op_or(other: A): A
fun op_xor(othr: A): A
```

appendices-examples-operator-overloading.pony

Create empty functions in a class

```
class Test
  fun alpha() =>
    """
    """
  fun beta() =>
    """
    """
```

appendices-examples-empty-class-functions.pony

How to create Arrays with values

Single values can be separated by semicolon or newline.

```
let dice: Array[U32] = [1; 2; 3
  4
  5
  6
]
```

appendices-examples-create-arrays-with-values.pony:5:9

How to modify a lexically captured variable in a closure

```
actor Main
  fun foo(n:U32): {ref(U32): U32} =>
    var s: Array[U32] = Array[U32].init(n, 1)
    {ref(i:U32)(s): U32 =>
      try
        s(0)? = s(0)? + i
        s(0)?
      else
        0
      end
    }

  new create(env:Env) =>
    var f = foo(5)
    env.out.print(f(10).string())
    env.out.print(f(20).string())
```

appendices-examples-modify-a-lexically-captured-variable-in-a-closure.pony

Whitespace

Whitespace (e.g. spaces, tabs, newlines, etc.) in Pony isn't significant.

Well, it mostly isn't significant.

Mostly insignificant whitespace

Pony reads a bit like Python, which is a *whitespace significant* language. That is, the amount of indentation on a line means something in Python. In Pony, the amount of indentation is meaningless.

That means Pony programmers can format their code in whatever way suits them.

There are three exceptions:

1. A `-` at the beginning of a line starts a new expression (unary negation), whereas a `-` in the middle of an expression is a binary operator (subtraction).
2. A `(` at the beginning of a line starts a new expression (a tuple), whereas a `(` in the middle of an expression is a method call.

3. A `[` at the beginning of a line starts a new expression (an array literal), whereas a `[` in the middle of an expression is generic formal parameters.

That stuff may seem a little esoteric right now, but we'll explain it all later. The `-` part should make sense though.

```
a - b
```

appendices-whitespace-subtract-b-from-a.pony:14:14

That means "subtract b from a".

```
a
-b
```

appendices-whitespace-do-a-then-do-a-unary-negation-of-b.pony:14:15

That means "first do a, then, in a new expression, do a unary negation of b".

Semicolons

In Pony, you don't end an expression with a `;`, unlike C, C++, Java, C#, etc. In fact, you don't need to end it at all! The compiler knows when an expression has finished, like Python or Ruby.

However, sometimes it's convenient to put more than one expression on the same line. When you want to do that, you **must** separate them with a `;`.

Why? Can't the compiler tell an expression has finished? Yes, it can. The compiler doesn't really need the `;`. However, it turns out the programmer does! By requiring a `;` between expressions on the same line, the compiler can catch some pretty common syntax errors for you.

Docstrings

Including documentation in your code makes you awesome. If you do it, everyone will love you.

Pony makes it easy by allowing you to put a **docstring** on every type, field, or method. Just put a string literal right after declaring the type or field, or right after the `=>` of a method, before writing the body. The compiler will know what to do with them.

For traits and interfaces that have methods without bodies, you can put the docstring after the method declaration, even though there is no `=>`.

By convention, a docstring should be a triple-quoted string, and it should use Markdown for any formatting.

```
actor Main
  """
  This is documentation for my Main actor
  """

  var count: USize = 0
    """
    This is documentation for my count field
    """

  new create(env: Env) =>
    """
    This is documentation for my create method
    """
    None

trait Readable
  fun val read()
    """
    This is documentation for my unimplemented read method
    """
```

appendices-whitespace-docstrings.pony

Comments

Use **docstrings** first! But if you need to put some comments in the implementation of your methods, perhaps to explain what's happening on various lines, you can use C++ style comments. In Pony, block comments can be nested.

```
// This is a line comment.
/* This is a block comment. */
/* This block comment /* has another block comment */ inside of it. */
```

appendices-whitespace-comments.pony

Compiler Arguments

`ponyc`, the compiler, is usually called in the project directory, where it finds the `.pony` files and its dependencies automatically. There it will create the binary based on the directory name. You can override this and tune the compilation with several options as described via `ponyc --help` and you can pass a separate source directory as an argument.

```
ponyc [OPTIONS] <package directory>
```

The most useful options are `--debug`, `--path` or just `-p`, `--output` or just `-o` and `--docs` or `-g`.

`--debug` will skip the LLVM optimizations passes. This should not be mixed up with `make config=debug`, the default make configuration target. `config=debug` will create DWARF symbols, and add slower assertions to ponyc, but not to the generated binaries. For those, you can omit DWARF symbols with the `--strip` or `-s` option.

`--path` or `-p` take a `:` separated path list as the argument and adds those to the compile-time library paths for the linker to find source packages and the native libraries, static or dynamic, being linked at compile-time or via the FFI at run-time. The system adds several paths already, e.g. on windows it queries the registry to find the compiler run-time paths, you can also use `use "lib:path"` statements in the source code and as a final possibility, you can add `-p` paths. But if you want the generated binary to accept such a path to find a dynamic library on your client system, you need to handle that in your source code by yourself. See the `options` package for this.

`--output` or `-o` takes a directory name where the final binary is created.

`--docs` or `-g` creates a directory of the package with documentation in Read the Docs format, i.e. markdown with nice navigation.

Let's study the documentation of the builtin standard library:

```
pip install mkdocs
ponyc packages/stdlib --docs && cd stdlib-docs && mkdocs serve
```

And point your web browser to 127.0.0.1:8000 serving a live-reloading local version of the docs.

Note that there is *no built-in debugger* to interactively step through your program and interpret the results. But ponyc creates proper DWARF symbols and you can step through your programs with a conventional debugger, such as GDB or LLDB.

Runtime options for Pony programs

Besides using the `cli` package, there are also several built-in options for the generated binary (*not for use with ponyc*) starting with `--pony*`, see `ponyc --help`, to tweak runtime performance. You can override the number of initial threads, tune cycle detection (*CD*), the garbage collector and even turn off yield, which is not really recommended.

Memory Allocation at Runtime

Pony is a null-free, type-safe language, with no dangling pointers, no buffer overruns, but with a **very fast garbage collector**, so you don't have to worry about explicit memory allocation, if on the heap or stack, if in a threaded actor, or not.

Fast, Safe and Cheap

- An actor has ~240 bytes of memory overhead.
- No locks. No context switches. All mutation is local.
- An idle actor consumes no resources (other than memory).
- You can have millions of actors at the same time.

But Caveat Emptor

But Pony can use external C libraries via the **FFI** which does not have this luxury.

So you **can** use any external C library out there, but the question is if you **need to** and if you **should**.

The biggest problem is external heap memory, created by an external FFI call, or created to support an external call. But external stack space might also need some thoughts, esp. when being created from actors.

Pony does have **finalisers** (callbacks which are called by the garbage collector which may be used to free resources allocated by an FFI call); However, the garbage collector is *not timely* (as with pure reference counting), it is not triggered immediately when some object goes out of scope.

A blocked actor will keep its memory allocated, only a dead actor will release it eventually.

And, long-running actors

Might cause unexpected out of memory errors, since the GC is not yet triggered on an out-of-memory segfault or stack exhaustion.

...

Garbage Collection with Pony-ORCA

Pony-ORCA is a fully concurrent protocol for garbage collection in the actor paradigm. It allows cheap and small actors to perform garbage collection concurrently with any number of other actors, and this number can go into the millions since one actor needs only 256 bytes on 64bit systems. It does not require any form of synchronization across actors except those introduced through the actor paradigm, i.e. **message send** and **message receive**.

Pony-ORCA, yes *the killer whale*, is based on ideas from ownership and deferred, distributed, weighted **reference counting**. It adapts messaging systems of actors to keep the reference count consistent. The main challenges in concurrent garbage collection are the detection of cycles of sleeping actors in the actor's graph, in the presence of the concurrent mutation of this graph. With message passing, you get deferred direct reference counting, a dedicated actor for the detection of (cyclic) garbage, and a confirmation protocol (to deal with the mutation of the actor graph).

1. *Soundness*: the technique collects only dead actors.
2. *Completeness*: the technique collects all dead actors eventually.
3. *Concurrency*: the technique does not require a stop-the-world step, clocks, time stamps, versioning, thread coordination, actor introspection, shared memory, read/write barriers or cache coherency.

The type system ensures at compile time that your program can **never have data races**. It's **deadlock free**... Because Pony has **no locks**!

When an actor has completed local execution and has no pending messages on its queue, it is *blocked*. An actor is *dead*, if it is blocked and all actors that have a reference to it are blocked, transitively. A collection of dead actors depends on being able to collect closed cycles of blocked actors.

The Pony type system guarantees race and deadlock free concurrency and soundness by adhering to the following principles:

Pony-ORCA characteristics

1. An actor may perform garbage collection concurrently with other actors while they are executing any kind of behaviour.
2. An actor may decide whether to garbage collect an object solely based on its own local state, without consultation with or inspecting the state of any other actor.
3. No synchronization between actors is required during garbage collection, other than potential message sends.
4. An actor may garbage collect between its normal behaviours, i.e. it need not wait until its message queue is empty.
5. Pony-ORCA can be applied to several other programming languages, provided that they satisfy the following two requirements:
    - **Actor behaviours are atomic**.
    - **Message delivery is causal**. Causal: messages arrive before any messages they may have caused if they have the same destination. So there needs to be some kind of causal ordering guarantee, but fewer requirements than with comparable concurrent, fast garbage collectors.

Platform-dependent Code

The Pony libraries, of course, want to abstract platform differences. Sometimes you may want a `use` command that only works under certain circumstances, most commonly only on a particular OS or only for debug builds. You can do this by specifying a condition for a `use` command:

```
use "foo" if linux
use "bar" if (windows and debug)
```

appendices-platform-dependent-code.pony

Use conditions can use any of the methods defined in `builtin/Platform` as conditions. There are currently the following booleans defined: `freebsd`, `linux`, `osx`, `posix` => `(freebsd or linux or osx)`, `windows`, `x86`, `arm`, `lp64`, `llp64`, `ilp32`, `native128`, `debug`

They can also use the operators `and`, `or`, `xor` and `not`. As with other expressions in Pony, parentheses **must** be used to indicate precedence if more than one of `and`, `or` and `xor` is used.

Any use command whose condition evaluates to false is ignored.

A Short Guide to Pony Error Messages

You've been through the tutorial, you've watched some videos, and now you're ready to write some Pony code. You fire up your editor, shovel coal into the compiler, and…you find yourself looking at a string of gibberish.

Don't panic! Pony's error messages try to be as helpful as possible and the ultimate goal is to improve them further. But, in the meantime, they can be a little intimidating.

This section tries to provide a short bestiary of Pony's error messages, along with a guide to understanding them.

Let's start with a simple one.

left side must be something that can be assigned to

Suppose you wrote:

```
actor Main
  let x: I64 = 0
  new create(env: Env) =>
    x = 12
```

error-messages-left-side-must-be-something-that-can-be-assigned-to.pony

The error message would be:

```
Error:
main.pony:4:5: can't assign to a let or embed definition more than once
    x = 12
    ^
Error:
main.pony:4:7: left side must be something that can be assigned to
    x = 12
      ^
```

What happened is that you declared `x` as a constant, by writing `let x`, and then tried to assign a new value to it, 12. To fix the error, replace `let` with `var` or reconsider what value you want `x` to have.

That one error resulted in two error messages. The first, pointing to the `x`, describes the specific problem, that `x` was defined with `let`. The second, pointing to the `=` describes a more general error, that whatever is on the left side of the assignment is not something that can be assigned to. You would get that same error message if you attempted to assign a value to a literal, like `3`.

left side is immutable

Suppose you create a class with a mutable field and added a method to change the field:

```
class Wombat
  var color: String = "brown"
  fun dye(new_color: String) =>
    color = new_color
```

error-messages-left-side-is-immutable.pony

The error message would be:

```
Error:
main.pony:4:11: left side is immutable
    color = new_color
          ^
```

To understand this error message, you have to have some background. The field `color` is mutable since it is declared with `var`, but the method `dye` does not have an explicit receiver reference capability. The default receiver reference capability is `box`, which allows `dye` to be called on any mutable or immutable `Wombat`; the `box` reference capability says that the method may read from but not write to the receiver. As a result, it is illegal to attempt to modify the receiver in the method.

To fix the error, you would need to give the `dye` method a mutable reference capability, such as `ref`: `fun ref dye(new_color: String) => ....`

receiver type is not a subtype of target type

Suppose you made a related, but slightly different error:

```
class Rainbow
  let colors: Array[String] = Array[String]
  fun add_stripe(color: String) =>
    colors.push(color)
```

error-messages-receiver-type-is-not-a-subtype-of-target-type.pony

In this example, rather than trying to change the value of a field, the code calls a method which attempts to modify the object referred to by the field.

The problem is very similar to that of the last section, but the error message is significantly more complicated:

```
Error:
main.pony:4:16: receiver type is not a subtype of target type
    colors.push(color)
```

```
            ^
  Info:
  main.pony:4:5: receiver type: this->Array[String val] ref (which becomes 'Array[String v
      colors.push(color)
      ^
  /root/.local/share/ponyup/ponyc-release-0.58.0-x86_64-linux-musl/packages/builtin/array
    fun ref push(value: A) =>
    ^
  main.pony:2:15: Array[String val] box is not a subtype of Array[String val] ref^: box is
    let colors: Array[String] = Array[String]
                ^
  main.pony:3:3: you are trying to change state in a box function; this would be possible
    fun add_stripe(color: String) =>
    ^
```

Once again, Pony is trying to be helpful. The first few lines describe the error, in general terms that only a programming language maven would like: an incompatibility between the receiver type and the target type. However, Pony provides more information: the lines immediately after "Info:" tell you what it believes the receiver type to be and the next few lines describe what it believes the target type to be. Finally, the last few lines describe in detail what the problem is.

Unfortunately, this message does not locate the error as clearly as the previous examples.

Breaking it down, the issue seems to be with the call to `push`, with the receiver `colors`. The receiver type is `this->Array[String val] ref`; in other words, the view that this method has of a field whose type is `Array[String val] ref`. In the class `Rainbow`, the field `colors` is indeed declared with the type `Array[String]`, and the default reference capability for `String`s is `val` while the default reference capability for `Array` is `ref`.

The "target type" in this example is the type declaration for the method `push` of the class `Array`, with its type variable `A` replaced by `String` (again, with a default reference capability of `val`). The reference capability for the overall array, as required by the receiver reference capability of `push`, is `ref`. It seems that the receiver type and the target type should be pretty close.

But take another look at the final lines: what Pony thinks is the actual receiver type, `Array[String val] box`, is significantly different from what it thinks is the actual target type, `Array[String val] ref`. And a type with a reference capability of `box`, which is immutable, is indeed not a subtype of a type with a reference capability of `ref`, which *is* mutable.

The issue must lie with the one difference between the receiver type and the target type, which is the prefix "this->". The type `this->Array[String val] ref` is a viewpoint adapted type, or arrow type, that describes the `Array[String val] ref` "as seen by the receiver". The receiver, in this case, has the receiver reference capability of the method `add_stripe`, which is the default `box`. *That* is why the final type is `Array[String val] box`.

The fundamental error in this example is the same as the last: the default receiver reference capability for a method is `box`, which is immutable. This method, however, is attempting to modify the receiver, by adding another color stripe. That is not legal at all.

As an aside, while trying to figure out what is happening, you may have been misled by the declaration of the `colors` field, `let colors....` That declaration makes the `colors` binding constant. As a result, you cannot assign a new array to the field. On the other hand, the array itself can be mutable or immutable. In this example, it is mutable, allowing `push` to be called on the `colors` field in the `add_stripe` method.

A note on compiler versions

The error messages shown in this section are from ponyc `0.58.0-a161b7c` release, the current "release" version at the time this is written. The messages from other versions of the compiler may be different, to a greater or lesser degree.

Program Annotations

In Pony, we provide a special syntax for implementation-specific annotations to various elements of a program. The basic syntax is a comma-separated list of identifiers surrounded by backslashes:

```
\annotation1, annotation2\
```

appendices-annotations-syntax.pony

Here, `annotation1` and `annotation2` can be any valid Pony identifier, i.e. a sequence of alphanumeric characters starting with a letter or an underscore.

What can be annotated

Annotations are allowed after any scoping keyword or symbol. The full list is:

- `actor`
- `class`
- `struct`
- `primitive`
- `trait`
- `interface`
- `new`
- `fun`
- `be`
- `if` (only as a condition, not as a guard)
- `ifdef`
- `elseif`
- `else`
- `while`
- `repeat`
- `until`
- `for`
- `match`
- `|` (only as a case in a `match` expression)
- `recover`
- `object`
- `{` (only as a lambda)
- `with`
- `try`
- `then` (only when part of a `try` block)

The effect of annotations

Annotations are entirely implementation-specific. In other words, the Pony compiler (or any other tool that processes Pony programs) is free to take any action for any annotation that it encounters, including not doing anything at all. Annotations starting with `ponyint` are reserved by the compiler for internal use and shouldn't be used by external tools.

Annotations in the Pony compiler

The following annotations are recognised by the Pony compiler. Note that the Pony compiler will ignore annotations that it doesn't recognise, as well as the annotations described here if they're encountered in an unexpected place.

`packed` Recognised on a `struct` declaration. Removes padding in the associated `struct`, making it ABI-compatible with a packed C structure with compatible members (declared with the `__attribute__((packed))` extension or the `#pragma pack` preprocessor directive in many C compilers).

```
struct \packed\ MyPackedStruct
  var x: U8
  var y: U32
```

appendices-annotations-packed-annotation.pony:1:3

`likely` and `unlikely` Recognised on a conditional expression (`if`, `while`, `until` and `|` (as a pattern matching case)). Gives optimisation hints to the compiler on the likelihood of a given conditional expression.

`nodoc`

Recognised on objects and methods (`actor`, `class`, `struct`, `primitive`, `trait`, `interface`, `new`, `be`, `fun`). Indicates to the documentation system that the item and any of its children shouldn't be included in generated output.

```
class \nodoc\Foo
   """
   We don't want this class and its methods to appear in generated documentation
   """
```

appendices-annotations-nodoc-annotation.pony

`nosupertype`

Recognised on objects(`actor`, `class`, `primitive`, `struct`). A type annotated with `nosupertype` will not be a subtype of any other type (except _), even if the type structurally provides an interface. If a `nosupertype` type has a provides list, a compiler error is reported. As a result, a `nosupertype` type is excluded from both nominal and structural subtyping.

Here's an example of how `nosupertype` can be important:

```
class Empty

class Foo
  fun foo[A: Any](a: (A | Empty val)) =>
    match consume a
    | let a': A => None
    end
```

appendices-annotations-empty-without-nosupertype-annotation.pony

The above code won't compile because you could supply `Empty ref`. Doing so results in a compiler error about an unsafe match because we would need to distinguish between `Empty val` and `Empty ref` at runtime.

By adding `nosupertype` to the definition of `Empty`, we declare that `Empty` is not a subtype of `Any` and thereby allow the code to compile as there is no longer an unsafe match.

```
class \nosupertype\ Empty

class Foo
  fun foo[A: Any](a: (A | Empty val)) =>
    match consume a
    | let a': A => None
    end
```

appendices-annotations-empty-with-nosupertype-annotation.pony:1:7

`nosupertype` is particularly valuable when constructing generic classes like collections that need a marker class to describe "lack of an item".

Serialisation

Pony provides a built-in mechanism for serialising and deserialising objects so that they can be passed between Pony processes. Serialisation takes an object and turns it into an array of bytes that can be used to send the object to another process by, for example, writing it to a TCP stream. Deserialisation takes an array of bytes and turns them into a Pony object.

Pony uses an intermediate object type called `Serialised` to represent a serialised object. A `Serialised` object can be created in one of two ways:

- calling the `create(...)` constructor with the `SerialiseAuth` authority and the object to serialize
- calling the `input(...)` constructor with the `DeserialiseAuth` authority and an `Array[U8]` that represents the object to deserialise. This intermediate object can then be used to either:
  - generate an `Array[U8]` that represents the object by calling the `output(...)` method with the `OutputSerialisedAuth` authority, or
  - generate a deserialised object by calling the `apply(...)` method with the `InputSerialisedAuth` authority

This program serialises and deserialise an object, and checks that the fields of the original object are the same as the fields of the deserialised object.

```
use "serialise"

class Foo is Equatable[Foo box]
  let _s: String
  let _u: U32

  new create(s: String, u: U32) =>
    _s = s
    _u = u

  fun eq(foo: Foo box): Bool =>
```

```
    (_s == foo._s) and (_u == foo._u)

actor Main
  new create(env: Env) =>
    try
      // get serialization authorities
      let serialise = SerialiseAuth(env.root)
      let output = OutputSerialisedAuth(env.root)
      let deserialise = DeserialiseAuth(env.root)
      let input = InputSerialisedAuth(env.root)

      let foo1 = Foo("abc", 123)

      // serialisation
      let sfoo = Serialised(serialise, foo1)?
      let bytes_foo: Array[U8] val = sfoo.output(output)

      env.out.print("serialised representation is " +
        bytes_foo.size().string() +
        " bytes long")

      // deserialisation
      let dfoo = Serialised.input(input, bytes_foo)
      let foo2 = dfoo(deserialise)? as Foo

      env.out.print("(foo1 == foo2) is " + (foo1 == foo2).string())
    else
      env.err.print("there was an error")
    end
```

appendices-serialization-compare-original-object-with-deserialized-object.pony

Caveats

There are several things to keep in mind when using Pony's serialisation system:

- Serialised objects will currently only work when passed between two running instances of the same Pony executable. You cannot pass objects between different Pony programs, nor can you pass them between different versions of the same program. Using the `Serialise.signature` function can help you determine if your two Pony programs are the same.
- Objects with `embed` fields will not be properly serialised.
- Objects with `Pointer` fields must use the custom serialisation mechanism or else the `Pointer` fields will be null when the object is deserialised. For information on how to handle these kinds of fields, please see the discussion of custom serialisation and deserialisation below.

Custom Serialisation and Deserialisation

Pony objects can have `Pointer` fields that store pointers to memory that contains things that are opaque to Pony but that may be useful to code that is called via FFI. Because the objects that `Pointer` fields point to are opaque, Pony cannot serialise and deserialise them by itself. However, Pony's serialisation system provides a way for the programmer to specify how the objects pointed to by these fields should be serialised and deserialised. This system is called custom serialisation.

Since `Pointer` fields are opaque to Pony, it is assumed that the serialisation and deserialisation code will be written in another language that knows how to read the object referenced by the pointers.

Custom Serialisation

In order to serialise an object from a pointer field, Pony needs to know how much space to set aside for that object and how to write a representation of that object into the reserved space. The programmer must provide two methods on the object:

- `fun _serialise_space(): USize` – This method returns the number of bytes that must be reserved for the object.
- `fun _serialise(bytes: Pointer[U8] tag)` – This method receives a pointer to the memory that has been set aside for serialising the object. The programmer must not write more bytes than were returned by the `_serialise_space` method.

Custom Deserialisation

Custom deserialisation is handled by a `fun ref _deserialise(bytes: Pointer[U8] tag)` method. This method receives a pointer to the character array that stores the serialised representation of the object (or objects) that the `Pointer` fields should point to. The programmer must copy out any bytes that will be used by the deserialised object.

The custom deserialisation method is expected to modify the values of the objects `Pointer` fields, so the fields must be declared `var` so that they can be modified.

Considerations

Fixed Versus Variable Object Sizes The programmer must write their custom serialisation and deserialisation code in such a way that it is aware of how many bytes are available in the byte arrays that are passed to the methods. If the objects are always of a fixed size then the functions can read and write that many bytes to the buffer. However, if the objects are of varying sizes (for example, if the object was a string), then the serialized representation must include information that the deserialisation code can use to ensure that it does not read beyond the end of the memory occupied by the object. The custom serialisation system does not provide a mechanism for doing this, so it is up to the program to choose a mechanism and implement it. In the case of a string, the serialisation format could consist of a 4-byte header that encodes the length of the string, followed by a string of the specified length. These additional four bytes must be included in the value returned by `_serialise_space()`. The deserialisation function would then start by reading the first four bytes of the array to obtain the size of the string and then read only that many bytes from the array.

Classes With Multiple `Pointer` Fields If a class has more than one `Pointer` field then all of those fields must be handled by the custom serialisation and deserialisation methods for that class; there are not methods for each field. For example, if a class has three `Pointer` fields then the `_serialise_space()` method must return the total number of bytes required to serialise the objects from all three fields.

Example

Assume we have a Pony class with a field that is a pointer to a C string. We would like to be able to serialise and deserialise this object. In order to do that, the Pony class implements the methods `_serialise_space(...)`, `_serialise(...)`, and `_deserialise(...)`. These methods, in turn, call C functions that calculate the number of bytes needed to serialise the string and serialise and deserialise it. In this example the serialised string is represented by a four-byte big-endian number that represents the length of the string, followed by the string itself without the terminating null. So if the C string is `hello world\0` then the serialised string is `\0x00\0x00\0x00\0x0Bhello world` (where the first four bytes of the serialised string are a big-endian representation of the number `0x0000000B`, which is `11`).

```
use "serialise"

use "lib:custser"

use @get_string[Pointer[U8]]()
use @serialise_space[USize](s: Pointer[U8] tag)
use @serialise[None](bytes: Pointer[U8] tag, str: Pointer[U8] tag)
use @deserialise[Pointer[U8] tag](bytes: Pointer[U8] tag)
use @printf[I32](fmt: Pointer[U8] tag, ...)

class CStringWrapper
  var _cstr: Pointer[U8] tag

  new create(cstr: Pointer[U8] tag) =>
    _cstr = cstr

  fun _serialise_space(): USize =>
    @serialise_space(_cstr)

  fun _serialise(bytes: Pointer[U8] tag) =>
    @serialise(bytes, _cstr)

  fun ref _deserialise(bytes: Pointer[U8] tag) =>
    _cstr = @deserialise(bytes)

  fun print() =>
    @printf(_cstr)

actor Main
  new create(env: Env) =>
    let csw = CStringWrapper(@get_string())
    csw.print()
    try
      let serialise = SerialiseAuth(env.root)
      let deserialise = DeserialiseAuth(env.root)

      let sx = Serialised(serialise, csw)?
      let y = sx(deserialise)? as CStringWrapper
      y.print()
    else
      env.err.print("there was an error")
    end
```

appendices-serialization-custom-serialization.pony

```c
// custser.c

#include <stdlib.h>
#include <string.h>

extern char *get_string()
{
  return "hello world\n";
}

extern size_t serialise_space(char *s)
{
  // space for the size and the string (without the null)
  return 4 + strlen(s);
}

extern void serialise(char *buff, char *s)
{
  size_t sz = strlen(s);
  unsigned char *ubuff = (unsigned char *) buff;
  // write the size as a 32-bit big-endian integer
  ubuff[0] = (sz >> 24) & 0xFF;
  ubuff[1] = (sz >> 16) & 0xFF;
  ubuff[2] = (sz >> 8) & 0xFF;
  ubuff[3] = sz & 0xFF;

  // copy the string
  strncpy(buff + 4, s, sz);
}

extern char *deserialise(char *buff)
{
  unsigned char *ubuff = (unsigned char *) buff;
  size_t sz = (ubuff[0] << 24) + (ubuff[1] << 16) + (ubuff[2] << 8) + ubuff[3];
  char *s = malloc(sizeof(char) * sz + 1);
  memcpy(s, buff + 4, sz);
  s[sz] = '\0';
  return s;
}
```