PONY CHEAT SHEET

COMMUNITY

website: https://ponylang.org
irc: freenode #ponylang
twitter: @ponylang
mailing list: https://pony.groups.io/g/user
tutorial: https://tutorial.ponylang.org
stdlib: https://stdlib.ponylang.org
github: https://github.com/ponylang
play: https://playground.ponylang.org/

HELLO WORLD

```
actor Main
  new create(env: Env) =>
    env.out.print("hello")
```

CONTROL

```
if … then …
elseif …
else …
end

try … else … end

match expr
| let x: T1 => …
| let x: T2 if expr => …
else …
end

for expr in iter do … end

while expr do … end

repeat expr do … end
```

LITERALS

```
"hello" // string
[1; 2; 3] // array
```

ACTOR

```
actor MyActor
  let _x: Type // private
  let x: Type // public
  new create() =>
    // initialization
  be my_behavior() =>
    // async behavior
  fun my_fun(): Type =>
    // synchronous function
```

CLASS

```
class MyClass
  let _x: Type // private
  let x: Type // public
  new create() =>
    // initialization
  fun my_fun(): Type =>
    // synchronous function
```

PRIMITIVE

```
primitive MyPrimitive
  // only has functions
  // no members
  fun my_fun(): Type =>
    // synchronous function
```

TRAIT (nominal subtyping)

subtyping is explicit using `is`

```
trait MyTrait
  fun my_fun() // opt impl
class MyClass is MyTrait
  fun my_fun() =>
    // do something
```

INTERFACE (structural subtyping)

any class that implements the interface's methods is a subtype of the interface

```
interface MyInterface
  fun my_fun() // opt impl
class MyClass
  fun my_fun() =>
    // do something
```

LAMBDA

```
{(arg, …): Type => … }
```

OPERATORS

math

```
+
-
*
/
%
```

bit shift

```
<<
>>
```

bitwise & logical

```
and
or
xor
not
```

compare

```
==
!=
<
>
<=
>=
is
isnt
```

negative

```
-
```

method call

```
.
```

method call, return receiver

```
.>
```

REF CAPS (REFERENCE CAPABILITIES)

`iso` - (isolated) alias is R/W, no other alias can R or W
`trn` -(transitional) alias is R/W, other aliases are R-only
`ref` -(reference) alias is R/W, other aliases can be R/W
`val` -(value) alias is R-only, other aliases are R-only
`box` -(box) alias is R-only, other aliases can be R-only or R/W
`tag` -(tag) alias cannot R or W, other aliases can R-only or R/W
Any alias can be used to send a message to an actor

REF CAP RULES

- if an object can be written to then only one actor can have a readable alias to it
- if an object can be read by multiple actors then no actor can have a writable alias to it

REF CAP USAGE

default refcap for type

```
actor refcap MyActor
class refcap MyClass
trait refcap MyTrait
interface refcap MyInterface
```

refcap of alias

```
let x: Type refcap
fun my_fun(x: Type refcap)
```

refcap of recovered object

```
recover refcap … end
```

refcap of new object

```
new refcap create()
```

refcap of method receiver

```
fun refcap my_fun()
```

refcap of return value

```
fun my_fun(): Type refcap
```

CONSUME

get rid of an alias

```
let x: Type iso = …
let y: Type val = consume x
```

RECOVER

"lift" the reference capability of the object created inside the recover block

- iso, trn, or ref objects can become anything
- val or box objects can become val or tag

```
let x = recover refcap
  // create something
end
```

ALIAS TYPE (!)

means "a type (including refcap) that can be assigned to this type (including refcap)"

- useful in generics

```
refcap!
```

EPHEMERAL TYPE (^)

type for an object that has no alias

- object returned by constructor
- object from consumed alias

```
refcap^
```

REF CAP SUBTYPING

if you give up an alias of X then you can assign (-->) the aliased object to a new alias of Y

$$\begin{tikzcd}
 & & \boxed{\texttt{val}} \arrow[dr] & & \\
\boxed{\texttt{iso}} \arrow[r] & \boxed{\texttt{trn}} \arrow[ur] \arrow[dr] & & \boxed{\texttt{box}} \arrow[r] & \boxed{\texttt{tag}} \\
 & & \boxed{\texttt{ref}} \arrow[ur] & &
\end{tikzcd}$$

version 0.0.2