Phasers Mark II

Philosophy

One well-known part of the Raku philosophy is "Make the easy things easy, and the hard things possible". I'd like to propose a small addendum "...and the orthogonal things orthogonal".

Overview

I love phasers. I love the idea of them, anyway. Once you actually start using them, reality shows up. My experience with phasers is that:

a) The ones that I want to use often don't exist anyway, because the orthogonal things haven't been kept orthogonal.

b) There are a lot of them, and I have to look them up if I want to use them. "Set phasers to the max" sounds fun ... until you have to remember all their names.

This proposal will make phasers much more flexible. But only if it actually gets implemented. On the down side, it will also make them more wordy. But on the up side again, instead of remembering 20 words, it will be a combination of about 10 words, and will present many more possibilities than the current phasers.

The things added will be:

- An optional topicaliser (to Phasers); this means that every phaser has a built-in, implicit `given {}`.
- Some additional methods on Block
- More (optional) prefixes (to Phasers)
- A modification of 'once', and a control structure 'assert'.
- Another magic variable

The things removed will be 18 of the 20 current phasers. While still having more flexibility than before.

Throughout this document, I will refer to the Current Style (ie. current Raku) and the New Style (ie. my suggestion). While I'm not proposing that we eliminate the existing phasers (they can remain as aliases), I'm not against it either. I won't hold you back if people really want to ... fire the phasers!

Given that my knowledge of Raku internals (or even asynchronous phasers) ranges from none to minimal, this proposal should be considered merely a beginning for discussion. But it may spark some ideas about orthogonality.

Demonstration

It will be useful to start by listing the Current Style Phasers, and suggesting a replacement for each in the New Style. Note that this does not demonstrate any of the newly-possible functionality. The

default topic for the `LEAVE` phaser is the `&?BLOCK` variable (but see below under *class Block (Enhanced)*), to the extent that if the topicaliser starts with a '.' (eg. `.block-done`), then that's considered to be `&?BLOCK.block-done()`. However, the topicaliser could also be another block label, or a variable containing a block, or something like that. There's nothing ... blocking you from doing that.

```
BEGIN {}   COMPILE ENTER {}
CHECK {}   COMPILE LEAVE {}
INIT {}    RUNTIME ENTER {}
END {}     RUNTIME LEAVE {}
DOC *      DOC *       (No change)
ENTER {}   ENTER {}    (No change)
LEAVE {}   LEAVE {}    (No change in syntax)
KEEP {}    LEAVE { .success and do {} }
UNDO {}    LEAVE { .success or do {} }
FIRST {}   ENTER { once {} }
NEXT {}    LEAVE .iter-done { when False {} }
LAST {}    LEAVE .iter-fine { when True {} }
PRE {}     ENTER { assert {} }
POST {}    LEAVE { assert {} }
CONTROL {} LEAVE .block-done { when X::Control {} }
CATCH {}   LEAVE .block-fine { when False {} }
QUIT {}    LEAVE .iter-fine  { when False {} }
LAST {}    LEAVE .iter-fine(CX::Done) { when True {} }
CLOSE {}   LEAVE .iter-done { when CX::Done {} }
COMPOSE {} COMPOSE ENTER {}
```

A lot of the above is relatively self-evident in its meaning. The parts that are the least evident are the new methods on Block. So I feel like I'll clarify things the most by starting there.

class Block (Enhanced)

These methods are being added to Block to support the functionality above.

method block-done

```
method block-done(--> Bool|Exception)
```

How was the block exited?

Possible return values and meanings are:

- `Bool False`: Not complete yet, or no entry-success (see entry-success, below)
- `Bool True`: No Exception (ie. normal exit)
- `Exception`: How the block exited. Note that this is not throwing an exception, but returning one.

method entry-success

```
method entry-success(--> Bool)
```

Was the the block successfully entered?

Returns True. Will be overridden by:

- `Iterating`: see below
- `Routine`: Currently, `LEAVE` blocks in a Routine currently run even if the parameter binding fails. To avoid this problem, use `LEAVE .entry-success { when True {} }`

method block-fine

```
method block-fine(Exception @exceptions --> Bool)
```

Did the block exit fine (if you'll pardon the English/Italian pun)?

The funtion is trivial, but convenient. Pseudo-code is:

```
method block-fine(Exception @exceptions) {
     @exceptions or @exceptions = (X::Control);
     given self.block-done {
           when Bool { return $_; }
           when any(@exceptions) { return True; }
           default { return False; }
     }
}
```

method success

```
method success(--> Bool)
```

What was the success value of the block?

This takes the Defintion of Success[1] and makes it no longer implementation-defined. It adds the idea that, if `entry-success` is `False`, then it's `False`. The short (pseudo-code) version is:

```
.block-done ~~ all(Bool, True) ?? $return-value !! False
```

class Iterating

Apologies if this concept already exists, but I didn't see anything in the documentation.

```
class Iterating is Block
```

The Iterating is a descendant of Block, and an ancestor to loop, `supply`, and `react` blocks. Because I intend it to be an ancestor for all loops (not just ones with an iterator) as well as `supply` and `react` blocks, possibly it should have a different name.

method entry-success

```
method entry-success(--> Bool)
```

Was the the block successfully entered?

Overrides the method on the parent Block. This is `True` when the Iterating has had at least one item provided to it, but is `False` if the Iterating has no items provided to it.

method iter-done

```
method iter-done(--> Bool|Exception)
```

How was the Iterating exited?

The possible return values are the same as for `Block.block-done()`, except that they apply to the completeness of the Iterating (loop/`supply`/`react`), and not the completeness of the block.

method iter-fine

```
method fine(Exception @exceptions --> Bool)
```

1 See https://design.raku.org/S04.html#Definition_of_Success

Did the Iterating exit fine?

This is the same as `block-fine`, but based around `.iter-done` instead of `.block-done`.

Magic Variable

I'd like to propose a new Magic Variable, `&?ITERATING`, which is like `&?BLOCK` and `&?ROUTINE`, but for the nearest containing Iterating block.

Control Flow Structures

The following changes would be useful.

once

This is a modified version of `once`. It's like the current `once`, but takes a block label (or Block) as a parameter, and happens only once within that block. It will reset (to run again) after the specified block is completed (or when the next one starts).

The default value is `&?ITERATING`. To get something like the current behaviour, we probably want to pass in something like `&?PACKAGE`. Unless there's a larger scope for the whole program that we could use instead.

First example: how the new structure would work without a block label:

```
for [1, 2, 3] -> $outeritem {
     for <a b c> -> $inneritem {
           print "$outeritem"
           once { print "--" }
           print "$inneritem, ";
     }
}
say;

# 1--a, 1b, 1c, 2--a, 2b, 2c, 3--a, 3b, 3c
```

The exact same code, but we pass a block label

```
OUTER: for [1, 2, 3] -> $outeritem {
     for <a b c> -> $inneritem {
           print "$outeritem"
           once OUTER { print "--" }
           print "$inneritem, ";
     }
}
say;

# 1--a, 1b, 1c, 2a, 2b, 2c, 3a, 3b, 3c
```

assert

```
assert { <expression> }
```

This is shorthand for the following:

```
if (<expression>) {raise Exception...}
```

The reason for this is to simplify the Current Style `PRE`/`POST` phasers.

New Phaser Syntax

The following is pseudocode, but should get the idea across.

```
rule phaser { <prefix>* <phaser-name> <topic>? <block> }
rule phaser-name { 'ENTER' | 'LEAVE' }
rule prefix { 'DOC' | 'RUNTIME' | 'COMPILE' | 'COMPOSE' }
```

From this, it will be observed that remaining phasers are:

- `ENTER`: When the block (or prefix-defined item) is entered.
- `LEAVE`: When the block (or refix-defined item) is left. However, just as putting a `LEAVE` inside a Routine that doesn't successfully bind parameters, putting a `LEAVE` in an Iterating should also activate if there are no iterations of the Iterating.

Regarding execution order *within a queue*, it's the same as declaration order. This actually gives more flexibility (especially as far as ordering goes), but is not backwards compatible.

Current Style

```
for 1..3 -> $item {
     say "Item is $item";
     ENTER { say "new loop"; }
     FIRST { say "first"; }
     PRE { $item < 3; }
}
```

New Style

```
for 1..3 -> $item {
     say "Item is $item";
     ENTER {
           assert { $item < 3; }
           once { say "first"; }
           say "new loop";
     }
}
```

The two code samples above are equivalent. In the Current Style, the order is fixed by the phaser ordering. In the New Style, while the order is the same as the other code sample, the option is available to change the ordering of the 3 lines in the `ENTER` block. More flexibility.

Prefixes

The prefixes are:

- `DOC`: Almost exactly the same as the one already in the raku documentation. Implies `COMPILE` (unless declared `RUNTIME`)
- `RUNTIME`: Makes the phaser happen at runtime, as early/late as possible
- `COMPILE`: Makes the phaser happen at compile time, as early/late as possible
- `COMPOSE`: Runs when a role is composed into a class

Obviously, `RUNTIME` and `COMPILE` override each other.

New Features Available

The New Style provides a number of advantages. Some have already been shown, but a couple more examples might be useful.

Only run the `LEAVE` code if the exit was a fallthrough, rather than a Control Exception.

```
for [1, 2, 3, 4, 5] -> $item {
```

```
    if $item == 6 {
        last;
    }
    LEAVE .block-done {
        when all(Bool, True) { say "No items match the special six"; }
    }
}
```

That feature alone should practically justify the new system. However, there are many others.

For example:

```
@array = ();
for @array -> $item {
    say "Item is $item";
    LEAVE .entry-success {
        when False { say "I wannan Item!  Gimmie Item!  .... no Item :("; }
    }
}
```

The code inside the `when False {}` only runs if there are no items in `@array`.

Alternative Ideas

Things that might need changing are:

- If we also need an `ASYNC` prefix to declare a Phaser asynchronous, that's an option too.
- If, through discussion, we establish that an Iterating should automatically nest another block inside it (and returns from the inner block are from the current iteration, whereas returns from the outer are returns from the whole thing), then we could replace `Block.block-(done|fine)` with `Block.(done|fine)` and `Iterating.iter-(done|fine)` with `Iterating.(done|fine)`. This might have advantages, but would probably also necessitate a restructure of the code, so I've avoided it.

Becasuse of the collapsing of most Phasers into `ENTER`/`LEAVE`, Phaser queues have become somewhat irrelevant.

Conclusion

Hopefully this will provide a starting point for a discussion about phasers, and how they might be made easier to work with.