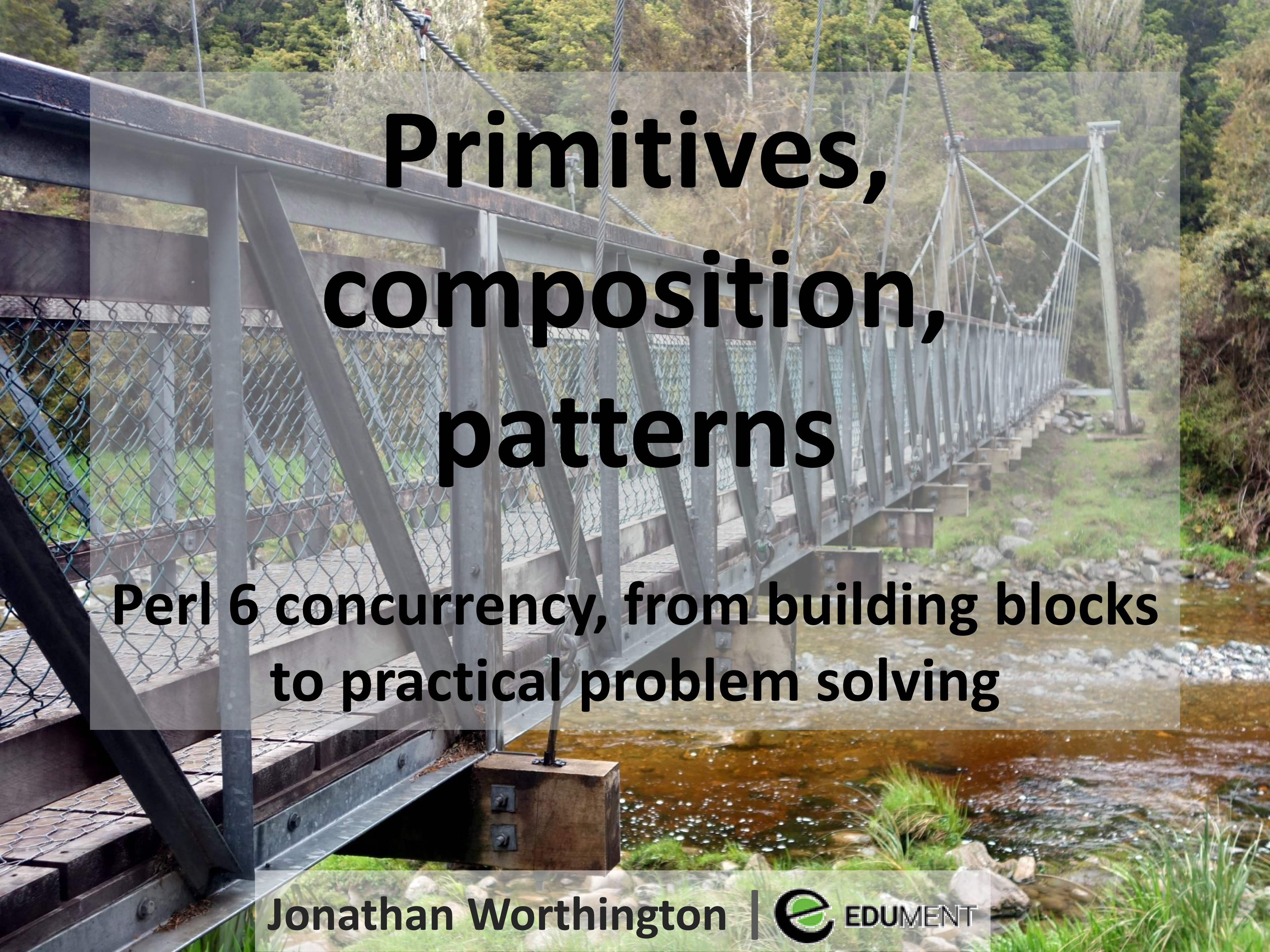


Primitives, composition, patterns

Perl 6 concurrency, from building blocks to practical problem solving

Jonathan Worthington | EDUMENT

A quick tour of the key Perl 6 concurrency primitives, ways of composing concurrent work, and a look at how we might tackle some practical concurrent problems

Perl 6 Concurrency Primitives

How do we represent...

	One value	Sequence of values
Synchronous		
Asynchronous		

A single synchronous value?

Well, that's just the value itself!

```
> 42.WHAT
(Int)

> "příběh".WHAT
(Str)

> class ShoppingList { has @.products }
> ShoppingList.new(products => <chicken ginger garlic>).WHAT
(ShoppingList)
```

How do we represent...

	One value	Sequence of values
Synchronous	Int, Str, ShoppingList	
Asynchronous		

A sequence of synchronous values

Represented by a Seq (for "Sequence")

Can produce values on demand (so may be lazy and infinite)

Doesn't remember the values

Example: lines from a file

```
> my $fh = open "README.md"
IO::Handle<"README.md".IO>(opened)

> $fh.lines.WHAT
(Seq)

> $fh.lines.head(2).perl
("# Rakudo Perl 6", "").Seq

> $fh.lines.head(1).perl
("This is Rakudo Perl, a Perl 6 compiler for the MoarVM",).Seq

> $fh.lines.grep(/Perl/).map(*.chars)
(72 66 68 61 61 73 71 68 64 61 65 62 63)

> $fh.eof
True
```

Concurrency with Seq

```
sub guesses($name) {
    gather loop {
        take prompt "$name, make a guess? ";
    }
}
```

Concurrency with Seq

```
sub guesses($name) {
    gather loop {
        take prompt "$name, make a guess? ";
    }
}
```

```
sub alternate(Iterable $a, Iterable $b) {
    my $iter-a = a.iterator;
    my $iter-b = b.iterator;
    gather loop {
        take $iter-a.pull-one;
        take $iter-b.pull-one;
    }
}
```

Concurrency with Seq

```
my $number = (1..100).pick;
say "I've thought of a number between 1 and 100. Guess it!";

for alternate guesses('Player A'), guesses('Player B') {
    when $number {
        say "You win!";
        exit;
    }
    when * < $number {
        say "Too low"
    }
    when * > $number {
        say "Too high"
    }
}
```

Concurrency with Seq

```
my $number = (1..100).pick;
say "I've thought of a number between 1 and 100. Guess it!";

for alternate guesses('Player A'), guesses('Player B') {
    when $number {
        say "You win!";
        exit;
    }
    when * < $number {
        say "Too low"
    }
    when * > $number {
        say "Too high"
    }
}
```

Concurrency with Seq

```
my $number = (1..100).pick;
say "I've thought of a number between 1 and 100. Guess it!";

for alternate guesses('Player A'), guesses('Player B') {
    }
}
```

```
sub alternate(Iterable $a, Iterable $b) {
    my $iter-a = a.iterator;
    my $iter-b = b.iterator;
    gather loop {
        take $iter-a.pull-one;
        take $iter-b.pull-one;
    }
}
```

Concurrency with Seq

```
my $number = (1..100).pick;
say "I've thought of a number between 1 and 100. Guess it!";

for alternate guesses('Player A'), guesses('Player B') {
}
```

```
sub alternate(Iterable $a, Iterable $b) {
    my $iter-a = a.iterator;
    my $iter-b = b.iterator;
    gather loop {
        take $iter-a.pull-one;
```

```
sub guesses($name) { # $name is Player A
    gather loop {
        take prompt "$name, make a guess? ";
    }
}
```

Concurrency with Seq

```
my $number = (1..100).pick;
say "I've thought of a number between 1 and 100. Guess it!";

for alternate guesses('Player A'), guesses('Player B') {
    ...
    }
}
```

```
sub alternate(Iterable $a, Iterable $b) {
    my $iter-a = a.iterator;
    my $iter-b = b.iterator;
    gather loop {
        take $iter-a.pull-one;
        take $iter-b.pull-one;
    }
}
```

Concurrency with Seq

```
my $number = (1..100).pick;
say "I've thought of a number between 1 and 100. Guess it!";

for alternate guesses('Player A'), guesses('Player B') {
    when $number {
        say "You win!";
        exit;
    }
    when * < $number {
        say "Too low"
    }
    when * > $number {
        say "Too high"
    }
}
```

Concurrency with Seq

```
my $number = (1..100).pick;
say "I've thought of a number between 1 and 100. Guess it!";

for alternate guesses('Player A'), guesses('Player B') {
    }
}
```

```
sub alternate(Iterable $a, Iterable $b) {
    my $iter-a = a.iterator;
    my $iter-b = b.iterator;
    gather loop {
        take $iter-a.pull-one;
        take $iter-b.pull-one;
    }
}
```

Concurrency with Seq

```
my $number = (1..100).pick;
say "I've thought of a number between 1 and 100. Guess it!";

for alternate guesses('Player A'), guesses('Player B') {
}
```

```
sub alternate(Iterable $a, Iterable $b) {
    my $iter-a = a.iterator;
    my $iter-b = b.iterator;
    gather loop {
        take $iter-a.pull-one;
        take $iter-b.pull-one;
```

```
sub guesses($name) { # $name is Player B
    gather loop {
        take prompt "$name, make a guess? ";
    }
}
```

Concurrency with Seq

Cooperative (control explicitly given up)

Asking for the next value blocks until it is available (either on computation or IO)

Quietly useful; often so quietly that people don't realize it's concurrency! ☺

How do we represent

	One value	Sequence of values
Synchronous	Int, Str, ShoppingList	Seq
Asynchronous		

A single asynchronous value?

A `Promise` represents a value that will be produced asynchronously

```
> my $p = Promise.new
> $p.status
Planned

> $p.keep(42)
Nil

> $p.status
Kept
> $p.result
42
```

Or inability to produce a value

A `Promise` can convey an exception

```
> my $p = Promise.new
> $p.break("I just couldn't do it man!")
Nil

> $p.status
Broken

> $p.result
Tried to get the result of a broken Promise
  in block <unit> at <unknown file> line 1
Original exception:
    I just couldn't do it man!
      in block <unit> at <unknown file> line 1
```

How is this useful?

A `Promise` will typically be kept by an operation that runs concurrently

That may be by code running on another thread, or some kind of asynchronous I/O (running a process, a network connection, etc.)

Kept by computation

The `start` keyword runs code in the thread pool, and returns a `Promise` that is kept/broken with the result

```
> my $p = start (1, 1, * + * ... Inf)[100000]
> $p.status
Planned

> $p.status
Kept

> $p.result.chars
20899
```

Kept by running a process

Built-in asynchronous operations uses `Promise` to convey results also

```
> my $proc = Proc::Async.new('/bin/sh', '-c', 'sleep 4')
Proc::Async.new(...)

> my $exit = $proc.start
> $exit.status
Planned

> $exit.status
Kept
> $exit.result.exitcode
0
```

How do we represent...

	One value	Sequence of values
Synchronous	Int, Str, ShoppingList	Seq
Asynchronous	Promise	

A sequence of asynchronous values

Represented by a `Supply`

As with Seq, can chain operations

But values are *pushed* through the pipeline of operations (it's reactive)

Basic publish/subscribe

```
> my $source = Supplier.new
> my $supply = $source.Supply;

> my $t1 = $supply.tap: { say "Got $_" }
> $source.emit("chili")
Got chili

> my $t2 = $supply.map(*.uc).tap: { say "OH WOW $_" }
> $source.emit("beef")
Got beef
OH WOW BEEF

> $t1.close
> $source.emit("noodles")
OH WOW NOODLES
```

Live vs. on-demand

A `Supplier` produces a live Supply

We tap into the stream of values at its current point, and won't see the past

Many - in fact, most - Supplies are on-demand; they start producing values at the point that they are tapped

The interval `Supply` factory

When the `Supply` returned by `interval` is tapped, it emits a value at the specified time interval

```
> my $ticks = Supply.interval(0.5)

> my $tap = $ticks.tap: { say now }; sleep 3; $tap.close;
Instant:1498686115.539947
Instant:1498686116.040888
Instant:1498686116.541719
Instant:1498686117.042902
Instant:1498686117.543302
Instant:1498686118.044487
```

Proc::Async again

Output arriving from stdout and stderr is exposed as a `Supply` also

```
> my $proc = Proc::Async.new('ps')
> my $collected = '';
> $proc.stdout.tap: { $collected ~= $_ }

> $proc.start.result.exitcode
0

> $collected
  PID TTY          TIME CMD
 6002 pts/18   00:00:00 bash
21472 pts/18   00:00:06 moar
29685 pts/18   00:00:00 ps
```

How do we represent...

	One value	Sequence of values
Synchronous	Int, Str, ShoppingList	Seq
Asynchronous	Promise	Supply


Composing
Asynchronous
Operations

Real programs will often involve dozens of asynchronous operations

We need good ways to compose them (that is, use them together)

Good compositions offer safety, correctness, error propagation, and resource management

`await`

The `await` function is the best way to prevent progress until a single value becomes available

```
> my $p = start (1, 1, * + * ... Inf)[100000]
> say await($p).chars
20899
```

Returns the `Promise` result if kept, or throws its exception if broken

Semantics of `await`

In Perl 6.c, it blocks the thread running the code until the result is available

In Perl 6.d, an `await` performed on a thread in the thread pool will take a continuation. When the results is available, the continuation is scheduled.

await many things

When many `Promise` objects are passed to `await`, it will wait for all of them and then return a list of the results

```
> my $parse-foo = start from-json slurp 'foo.json'
> my $parse-bar = start from-json slurp 'bar.json'

> my ($foo, $bar) = await $parse-foo, $parse-bar

> say $foo
{foo => 42}
> say $bar
[1 2 3]
```

Sequencing

Sometimes, we want to wait until one of, or all of, a set of `Promise` objects are either kept or broken - without caring for the results (or getting the errors)

This is done by `Promise.anyof(...)` and `Promise.allof(...)`

Kill a process after a timeout

A fairly common use of anyof is to wait for something to happen, or for a timeout, whichever comes first

```
> my $proc = Proc::Async.new('/bin/sh', '-c', 'sleep 100');
> my $exited = $proc.start

> await Promise.anyof($exited, Promise.in(5))
True

> unless $exited { $proc.kill }
1
```

Supplies: more challenging

Operations receiving data from multiple supplies present some challenges:

Data may arrive concurrently

Must keep track of when we're done

Must remember to "unsubscribe"

Train delay notifications

We have a stream of events about delays to train services

```
class TrainDelay {
    has Str $.train-code;
    has Int $.minutes-delayed;
}
```

An app uses a web socket to receive notifications of delays

Train delay notifications

The app sends a list of train codes that the user wishes to get notifications on

We want to batch up delay information arriving within 15 seconds, so as to reduce network traffic

A means to notify

We will create a `Supplier` in order to emit notifications on

```
sub user-notifications(@relevant-codes --> Supply) {
    my $notifications = Supplier.new;
    # ...
    return $notifications.Supply;
}
```

We return the `Supply` obtained from it

Subscribe for each train

There is a `Supply` of delay information for each train, which we can tap

```
sub user-notifications(@relevant-codes --> Supply) {
    my $notifications = Supplier.new;
    for @relevant-codes -> $code {
        delays-for($code).tap: -> $delay {
            # ...
        }
    }
    return $notifications.Supply;
}
```

Collect latest delay info

Unpack the object field we want, form a message, stash it away

```
sub user-notifications(@relevant-codes --> Supply) {
    my $notifications = Supplier.new;
    my @latest;
    for @relevant-codes -> $code {
        delays-for($code).tap: -> (:$minutes-delayed, *%) {
            push @latest, "$code delay: $minutes-delayed mins";
        }
    }
    return $notifications.Supply;
}
```

Notify every 15 seconds

```
sub user-notifications(@relevant-codes --> Supply) {
    my $notifications = Supplier.new;
    my @latest;
    for @relevant-codes -> $code {
        delays-for($code).tap: -> (:$minutes-delayed, *%) {
            push @latest, "$code delay: $minutes-delayed mins";
        }
    }
    Supply.interval(15).tap: {
        if @latest {
            $notifications.emit: @latest.join("\n");
            @latest = ();
        }
    }
    return $notifications.Supply;
}
```

So easy, right?

So easy, right?

Well, not so fast

So easy, right?

Well, not so fast

This code leaks resources

And it has data races

And it silently eats any errors

Leaks

```
sub user-notifications(@relevant-codes --> Supply) {
    my $notifications = Supplier.new;
    my @latest;
    for @relevant-codes -> $code {
        delays-for($code).tap: -> (:$minutes-delayed, *%) {
            push @latest, "$code delay: $minutes-delayed mins";
        }
    }
    Supply.interval(15).tap: {
        if @latest {
            $notifications.emit: @latest.join("\n");
            @latest = ();
        }
    }
    return $notifications.Supply;
}
```

Tracking the taps

```
sub user-notifications(@relevant-codes --> Supply) {
    my $notifications = Supplier.new;
    my @taps;
    my @latest;
    for @relevant-codes -> $code {
        push @taps, delays-for($code).tap: -> (:$minutes-delayed, *%) {
            push @latest, "$code delay: $minutes-delayed mins";
        }
    }
    push @taps, Supply.interval(15).tap: {
        if @latest {
            $notifications.emit: @latest.join("\n");
            @latest = ();
        }
    }
    return $notifications.Supply.on-close({ @taps>>.close });
}
```

Data races

```
sub user-notifications(@relevant-codes --> Supply) {
    my $notifications = Supplier.new;
    my @taps;
    my @latest;
    for @relevant-codes -> $code {
        push @taps, delays-for($code).tap: -> (:$minutes-delayed, *%) {
            push @latest, "$code delay: $minutes-delayed mins";
        }
    }
    push @taps, Supply.interval(15).tap: {
        if @latest {
            $notifications.emit: @latest.join("\n");
            @latest = ();
        }
    }
    return $notifications.Supply.on-close({ @taps>>.close });
}
```

Fix it with a lock

```
sub user-notifications(@relevant-codes --> Supply) {
    my $notifications = Supplier.new;
    my @taps;
    my $lock = Lock.new;
    my @latest;
    for @relevant-codes -> $code {
        push @taps, delays-for($code).tap: -> (:$minutes-delayed, *%) {
            $lock.protect: {
                push @latest, "$code delay: $minutes-delayed mins";
            }
        }
    }
    push @taps, Supply.interval(15).tap: {
        $lock.protect: {
            if @latest {
                $notifications.emit: @latest.join("\n");
                @latest = ();
            }
        }
    }
    return $notifications.Supply.on-close({ @taps>>.close });
}
```

All this tricky boilerplate ☹

```
sub user-notifications(@relevant-codes --> Supply) {
    my $notifications = Supplier.new;
    my @taps;
    my $lock = Lock.new;
    my @latest;
    for @relevant-codes -> $code {
        push @taps, delays-for($code).tap: -> (:$minutes-delayed, *%) {
            $lock.protect: {
                push @latest, "$code delay: $minutes-delayed mins";
            }
        }
    }
    push @taps, Supply.interval(15).tap: {
        $lock.protect: {
            if @latest {
                $notifications.emit: @latest.join("\n");
                @latest = ();
            }
        }
    }
    return $notifications.Supply.on-close({ @taps>>.close });
}
```

`supply` and `whenever`

A `supply` block evaluates to a `Supply`

The body runs each time it is tapped

The `whenever` construct taps a `Supply`

Automatic tap management and concurrency control

Start with a supply block

It is returned implicitly, though we could write `return` before it if we wished

```
sub user-notifications(@relevant-codes --> Supply) {
    supply {

    }
}
```

Tap with whenever

This automatically captures the taps, and will automatically close them for us

```
sub user-notifications(@relevant-codes --> Supply) {
    supply {
        for @relevant-codes -> $code {
            whenever delays-for($code) {
            }
        }
        whenever Supply.interval(15) {
        }
    }
}
```

Just emit values

```
sub user-notifications(@relevant-codes --> Supply) {
    supply {
        my @latest;
        for @relevant-codes -> $code {
            whenever delays-for($code) {
                push @latest,
                    "$code delay: {.minutes-delayed} mins";
            }
        }
        whenever Supply.interval(15) {
            if @latest {
                emit @latest.join("\n");
                @latest = ();
            }
        }
    }
}
```

And the concurrency control?

Only one thread is allowed to be inside of the code in a `Supply` block at a time

No two `whenever` blocks can be running at the same time

No `whenever` block can start until the `supply` block's setup work is done

If you know what actors are...

You can think of a `supply` block as being a little bit like one (it's not quite, but...)

Each tapping instantiates a new actor (the state is just lexicals, not attributes)

One message is processed at a time

Practical Examples

A retry mechanism

Various ways to build these

The sequence operator is a cute way to specify the back-off strategy

We'll build it a couple of different ways to see some of the possibilities

Retry synchronous operation

The first way assumes we are passed a code object that runs synchronously

We'll return a `Promise` that will be kept when the operation succeeds (maybe after some retries), or is broken when all of the retries are used up

Solution

Loop over back-off intervals prepended with zero, break out of the loop if we succeed, throw if we never succeed

```
sub retry(&operation, @intervals --> Promise) {
    start {
        for flat 0, @intervals -> $backoff {
            await Promise.in($backoff);
            try operation();
            last without $!;
            LAST .rethrow with $!;
        }
    }
}
```

Solution

Loop over back-off intervals prepended with zero, break out of the loop if we succeed, throw if we never succeed

```
sub retry(&operation, @intervals --> Promise) {
    start {
        for flat 0, @intervals -> $backoff {
            await Promise.in($backoff);
            try operation();
            last without $!;
            LAST .rethrow with $!;
        }
    }
}
```

In Perl 6.d, this will free up the thread to work on another operation

Example usage: immediate success

```
await retry { say "Worked!"; 42 }, (1, 2 ... 5);
```

```
Worked!
```

Example usage: success after some retries

```
await retry
    {
        state $i++;
        say "Attempt $i at {now}";
        die "oops" if $i < 3;
        say "Worked!"
    },
    (1, 2 ... 5);
```

```
Attempt 1 at Instant:1498776121.241535
Attempt 2 at Instant:1498776122.252485
Attempt 3 at Instant:1498776124.258838
Worked!
```

Example usage: broken

```
await retry
    {
        state $i++;
        say "Attempt $i at {now}";
        die "totally busted"
    },
    (1, 2 ... 5);
```

```
Attempt 1 at Instant:1498776281.514535
Attempt 2 at Instant:1498776282.518944
Attempt 3 at Instant:1498776284.524859
Attempt 4 at Instant:1498776287.532635
Attempt 5 at Instant:1498776291.541726
Attempt 6 at Instant:1498776296.552463
Tried to get the result of a broken Promise
  in block <unit> at retry-and-backoff.p6 line 23
Original exception:
    totally busted
    ...
```

For asynchronous work...

Just await what the operation returns

```
sub retry(&operation, @intervals --> Promise) {
    start {
        for flat 0, @intervals -> $backoff {
            await Promise.in($backoff);
            try await operation();
            last without $!;
            LAST .rethrow with $!;
        }
    }
}
```

Again, will scale better in v6.d

Back-off strategies

Arithmetic progression, as already:

```
retry &the-work, (5, 10 ... 25)
```

Geometric progression:

```
retry &the-work, (2, 4, 8 ... 64)
```

Fibonacci sequence:

```
retry &the-work, (1, 1, * + * ... 34)
```

Thinking less: say how many

Especially with Fibonacci, it becomes less obvious how many retries we'll actually get. So, just write the infinite sequence and use head.

```
retry &the-work, (5, 10 ... *).head(5)
```

```
retry &the-work, (2, 4, 8 ... *).head(5)
```

```
retry &the-work, (1, 1, * + * ... *).head(5)
```

A Supply retry

```
sub retry(Supply $s, @intervals --> Supply) {
    supply {
        my @remaining = @intervals;
        sub attempt() {
            whenever $s -> $result {
                emit $result;
                QUIT {
                    when @remaining != 0 {
                        whenever Promise.in(@remaining.shift) {
                            attempt();
                        }
                    }
                }
            }
        }
        attempt();
    }
}
```

A Supply retry

```
sub retry(Supply $s, @intervals --> Supply) {
    supply {
        my @remaining = @intervals;
        sub attempt() {
            whenever $s -> $result {
                emit $result;
                QUIT {
                    when @remaining != 0 {
                        whenever Promise.in(@remaining.shift) {
                            attempt();
                        }
                    }
                }
            }
        }
        attempt();
    }
}
```

Get our own copy of the interval array, so we can shift from it

A Supply retry

```
sub retry(Supply $s, @intervals --> Supply) {
    supply {
        my @remaining = @intervals;
        sub attempt() {
            whenever $s -> $result {
                emit $result;
                QUIT {
                    when @remaining != 0 {
                        whenever Promise.in(@remaining.shift) {
                            attempt();
                        }
                    }
                }
            }
        }
        attempt();
    }
}
```

QUIT is for handling asynchronous exceptions

A Supply retry

```
sub retry(Supply $s, @intervals --> Supply) {
    supply {
        my @remaining = @intervals;
        sub attempt() {
            whenever $s -> $result {
                emit $result;
                QUIT {
                    when @remaining != 0 {
                        whenever Promise.in(@remaining.shift) {
                            attempt();
                        }
                    }
                }
            }
        }
        attempt();
    }
}
```

If no when clauses match in a QUIT, exception re-thrown (just like in CATCH)

A Supply retry

```
sub retry(Supply $s, @intervals --> Supply) {
    supply {
        my @remaining = @intervals;
        sub attempt() {
            whenever $s -> $result {
                emit $result;
                QUIT {
                    when @remaining != 0 {
                        whenever Promise.in(@remaining.shift) {
                            attempt();
                        }
                    }
                }
            }
        }
        attempt();
    }
}
```

Notice how whenever can work with a Promise too; it's just like a 1-value Supply!

A Supply retry

```
sub retry(Supply $s, @intervals --> Supply) {
    supply {
        my @remaining = @intervals;
        sub attempt() {
            whenever $s -> $result {
                emit $result;
                QUIT {
                    when @remaining != 0 {
                        whenever Promise.in(@remaining.shift) {
                            attempt();
                        }
                    }
                }
            }
        }
        attempt();
    }
}
```

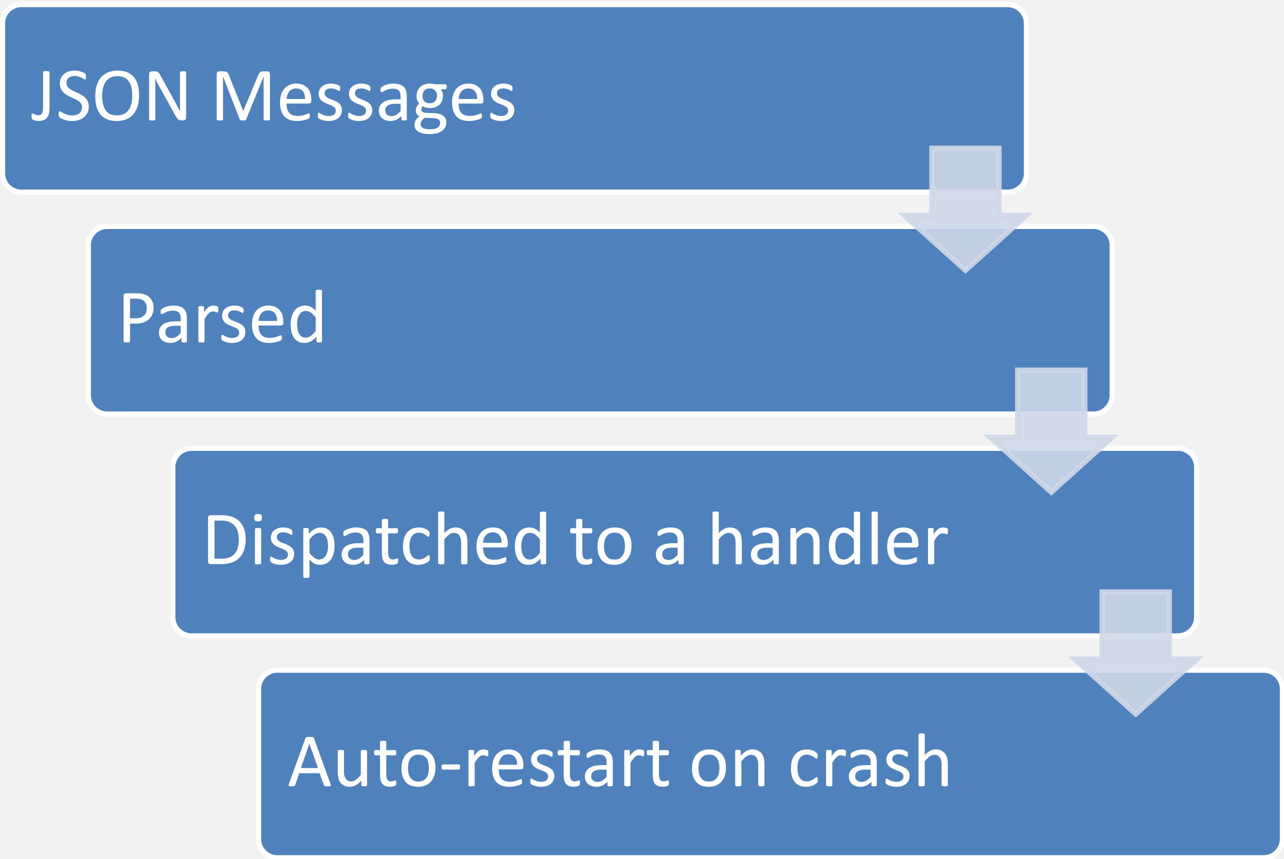
Since whenever is an asynchronous construct, this is not actually recursive!

Reactive message processing

Using `supply` blocks, it is possible to build up chains of operations that react to incoming messages

A nice fallout from this approach is that if something crashes and goes unhandled, it will tear down the chain for us; we can then restart it

An example pipeline



The parse stage

Parses the input as JSON, and emits the result of the parsing

```
sub parse(Supply $incoming --> Supply) {
    use JSON::Tiny;
    supply {
        whenever $incoming {
            emit from-json($_);
        }
    }
}
```

Basic JSON→object mapper

```
sub make-objectifier(%class-map) {
    return -> Supply $incoming {
        supply {
            whenever $incoming -> $json {
                if $json ~~ Hash and $json<type>:exists {
                    if %class-map{$json<type>}:exists {
                        emit %class-map{$json<type>}.new(|$json);
                    }
                    else {
                        die "Message type $json<type> unhandled";
                    }
                }
                else {
                    die "JSON did not parse to an object";
                }
            }
        }
    }
}
```

Call a handler on each

We could write:

```
sub make-processor(&handler) {
    return -> Supply $incoming {
        supply {
            whenever $incoming -> $object {
                handler($object)
            }
        }
    }
}
```

But that's just a long way to say:

```
sub make-processor(&handler) {
    return $incoming.map(&handler);
}
```

An auto-restarter

```
sub auto-restart(Supply $incoming) {
    supply {
        sub run() {
            whenever $incoming {
                QUIT {
                    default {
                        .note;
                        run();
                    }
                }
            }
        }
        run();
    }
}
```

Some message types

These are classes that some incoming messages will be transformed into

```
class TrainDelayed {
    has $.train-code;
    has $.minutes;
}

class TrainCancelled {
    has $.train-code;
    has $.reason;
}
```

Some message handlers

Now that we have types, we can use multiple dispatch to write handlers

```
multi handle(TrainDelayed $d) {
    say "Train $d.train-code() was delayed $d.minutes() mins";
}
multi handle(TrainCancelled $c) {
    say "Train $c.train-code() was cancelled. $c.reason()";
}
```

A composition mechanism

Finally, we need a way to put all of the pieces together into one pipeline

```
sub compose(Supply $input, *@stages) {
    my $current = $input;
    for @stages -> &build-stage {
        $current = build-stage($current);
    }
    return $current;
}
```

A composition mechanism

Which is actually just a reduce, in functional speak

```
sub compose(Supply $input, *@stages) {
     ($input, |@stages).reduce({ $^b($^a) })
}
```

Let's run it!

Compose the pipeline, and then run it (it runs forever, so `wait` never returns)

```
my $pipeline = compose
    $fake-message-source,
    &parse,
    make-objectifier({
        delay => TrainDelayed,
        cancellation => TrainCancelled
    }),
    make-processor(&handle),
    &auto-restart;

$pipeline.wait;
```

Concurrent processing

Supplies are a tool for controlling concurrency, not introducing it

However, with a little effort we can get some concurrency in place

We can also support asynchronous message handlers

Parse JSON in the thread pool

Note: the trade-off here is that we may lose message order

```
sub parse(Supply $incoming --> Supply) {
    use JSON::Tiny;
    supply {
        whenever $incoming {
            whenever start from-json($_) -> $parsed {
                emit $parsed;
            }
        }
    }
}
```

Allowing concurrent handlers

This way lets handlers choose to be concurrent (return `Promise` or `Supply`) but will cope with synchronous too

```
sub make-processor(&handler) {
    return -> Supply $incoming {
        supply {
            whenever $incoming -> $object {
                whenever handler($object) { }
            }
        }
    }
}
```

The whenever means we will not lose asynchronous errors

Allowing concurrent handlers

This way lets handlers choose to be concurrent (return `Promise` or `Supply`) but will cope with synchronous too

```
sub make-processor(&handler) {
    return -> Supply $incoming {
        supply {
            whenever $incoming -> $object {
                whenever handler($object) { }
            }
        }
    }
}
```

And synchronous handlers? Return value coerces into a 1-item Supply. "Just works."

Running handlers on threads

Alternatively, we could expect handlers to always be synchronous and then run them off in the thread pool

```
sub make-processor(&handler) {
    return -> Supply $incoming {
        supply {
            whenever $incoming -> $object {
                whenever start handler($object) { }
            }
        }
    }
}
```

A word of warning

Once we add in concurrency, we lose back-pressure

A very active source of messages could flood the system with work

For production use, it's wise to have a mechanism to cope with this

A back-pressure approach

```
sub make-processor(&handler) {
    return -> Supply $incoming {
        supply {
            my @in-progress;
            whenever $incoming -> $object {
                @in-progress .= grep(*.status == Planned);
                if @in-progress > 5 {
                    await Promise.anyof(@in-progress);
                }
                push @in-progress, my $done = Promise.new;
                whenever start handler($object) {
                    $done.keep();
                }
            }
        }
    }
}
```

A back-pressure approach

```
sub make-processor(&handler) {
    return -> Supply $incoming {
        supply {
            my @in-progress;
            whenever $incoming -> $object {
                @in-progress .= grep(*.status == Planned);
                if @in-progress > 5 {
                    await Promise.anyof(@in-progress);
                }
                push @in-progress, my $done = Promise.new;
                whenever start handler($object) {
                    $done.keep();
                }
            }
        }
    }
}
```

The await prevents processing of any new messages

Quick mention: an alternative

Having a `Supply` per message type sometimes is more suitable (and then a router that emits them to each)

This is especially true of Complex Event Processing, where we want to write logic to correlate events


In
Summary

Shared async data structures

Writing modules that worked together would be hard in a language with no common understanding of what a string, array, or hash is

By putting `Promise` and `Supply` into the core Perl 6 language, we provide a means for asynchronous composition

Use high-level constructs

Where possible, prefer to use `await`, or `supply`/`react` blocks with `whenever`

These provide for structured concurrent programming (much like `if` statements and loops are the structured equivalent to a load of `goto`)

Perl 6 can help, but...

At the end of the day, concurrent programming is still concurrent programming

Requires different thinking

Time becomes part of the programming model

The journey continues

For Perl 6, this is just the beginning

Already we know 6.d will make `await` far more scalable

Also plans for a more declarative approach to concurrent message processing and back-pressure


Questions
and
Discussion