Understanding react, supply, and whenever

Jonathan Worthington
Edument
Well, they're just...
A mechanism for consuming zero or more asynchronous streams (often called "reactive streams"), providing concurrency control through one-message-at-a-time processing, automatically handling error and completion propagation, and managing subscriptions, with the option of producing a new stream of values as a result.
Any questions?
Let's talk about for loops.
for $fh.lines -> $line {
    last if $line ~~ /^END/;
    say $line.split(' '())[0];
}
for $fh.lines -> $line {
    last if $line ~~ /^END/;
    say $line.split(' ')[0];
}
for $fh.lines -> $line {
    last if $line ~~ /^END/;
    say $line.split(' ')[0];
}
These happen in lockstep

- Pull one value
- Run the loop body

```perl
for $fh->lines -> $line {
    last if $line eq 'END';
    say $line.split(' ')[0];
}
```

Lost interest? Just stop. GC eats iterator.
for $fh.lines -> $line {
  last if $line eq 'END';
  say $line.split(' ')[0];
}

These happen in lockstep

Pull one value

Run the loop body

Lost interest? Just stop.
GC eats iterator.

Next statement runs after completion of the loop
These happen in lockstep

Pull one value

Run the loop body

for $fh.lines -> $line {
  last if $line eq 'END';
  say $line.split(' ')[0];
}

Next statement runs after completion of the loop

Synchronous programming

Lost interest? Just stop. GC eats iterator.
Once we spot the iterator pattern, we can see its use in many situations...

- Moving through a collection
- Reading lines (lazily) from a file
- Reading rows from a database
- Walking anything via a generator
So what about asynchronous programming?
The idea of asynchronous streams, like iterators, captures so many things...

Output from sub-process handles
Packets arriving over a socket
Ticks of a timer
User interface events
Message queue messages
Business/domain events
Thought experiment:

What if we were to have a loop-like construct for asynchronous streams?
```perl
$proc.stdout.lines -> $line {
    last if $line ~~ /^END/;
    say $line.split(' ')[0];
}
```

Observable data source
??? $proc.stdout.lines -> $line {
    last if $line ~~ /^END/;
    say $line.split(' ')[0];
}

Subscribe to events

Run code on an event
Events occur any time, even on any thread, after subscription

Subscribe to events

```perl
$proc.stdout.lines -> $line { 
    last if $line ~~ /^END/;
    say $line.split(' ')[0];
}
```

Run code on an event
Events occur any time, even on any thread, after subscription

Subscribe to events

```perl
$proc.stdout.lines -> $line { last if $line ~~ /^END/;
say $line.split(' ')[0];
}
```

Run code on an event

Unsubscribe
Subscribe to events

Run code on an event

```perl
$proc.stdout.lines -> $line { 
  last if $line ~~ /^END/;
  say $line.split(' ')[0];
}
```

Unsubscribe

Next statement... uhh... hmmm
Events occur any time, even on any thread, after subscription.

Subscribe to events:

```perl
$proc.stdout.lines -> $line {  
  last if $line ~~ /^END/;  
  say $line.split(' ')[0];  
}
```

Run code on an event:

Unsubscribe

Next statement... uhh... hmmm

Asynchronous programming
Events occur any time, even on any thread, after subscription.

Asynchronous programming

We'll need a means of concurrency control.

Subscribe to events

Run code on an event

Unsubscribe

Next statement... uhh... hmmm

```
proc.stdout.lines -> $line { 
~~ /^END/; 
  say $line.split(' ')[0];
}
```
Events occur any time, even on any thread, after subscription.

Subscribe to events

Run code on an event

Unsubscribe

Asynchronous programming

Could await the end of the stream...

```perl
proc.stdout.lines -> $line {
  last if $line ~~ /^END/;
  say $line.split(' ')[0];
}
```

Next statement... uhh... hmmm
Events occur any time, even on any thread, after subscription

Subscribe to events

...but what if we want to process many streams?

```perl
$proc.stdout.lines |-> $line {
  last if $line ~~ /^END/;
  say $line.split(' ')[0];
}
```

Run code on an event

Unsubscribe

Next statement... uhh... hmmm

Asynchronous programming
In Perl 6, **whenever** is an asynchronous loop construct

```perl
whenever $proc.stdout.lines -> $line {
    last if $line ~~ /^END/;
    say $line.split(' ')
         [0];
}
```

Subscribes to the stream
Runs the body on each event
Unsubscribes on last
Whenever must be used in combination with either react or supply.
react whenever Supply.interval(1) -> $i { 
say $i %% 2 ?? "Tick" !! "Tock";
last if $i > 10;
}

A whenever subscribes, then execution continues
Waits until whenever exits or the stream ends

react whenever Supply.interval(1) -> $i {  
say $i % 2 ?? "Tick" !! "Tock";  
last if $i > 10;
}
As with other statement prefixes, `react` takes a block or a statement

```plaintext
react {
    whenever Supply.interval(1) -> $i {
        say $i % 2 ?? "Tick" !! "Tock";
        last if $i > 10;
    }
}
```
As with other statement prefixes, react takes a block or a statement

```javascript
react {
    whenever Supply.interval(1) -> $i {
        say $i % 2 ?? "Tick" !! "Tock";
        last if $i > 10;
    }
    # So we can add another whenever!
}
```
whenever sets up a subscription

react manages a set of subscriptions
Run a process, color STDOUT green and STDERR yellow, pass on exit code

use Terminal::ANSIColor;
unit sub MAIN(Str $program, *@args);
react {
    my $proc = Proc::Async.new($program, @args);
    whenever $proc.stdout.lines {
        say colored($_, 'green');
    }
    whenever $proc.stderr.lines {
        note colored($_, 'yellow');
    }
    whenever $proc.start {
        exit .exitcode;
    }
}
A react terminates when there are no more subscriptions

Often, this allows us to eliminate a bunch of completion tracking logic
# Run 4 test files at a time
my $degree = 4;

# Find the test files to run
my @tests = dir('t').grep(/\.t$/);

# And here comes the fun stuff...
react {
    ...
}

Run test files in parallel, envelope output

# Run 4 test files at a time
my $degree = 4;

# Find the test files to run
my @tests = dir('t').grep(/\.t$/);

# And here comes the fun stuff...
react {
  # Set off $degree tests at first
  run-one for 1..$degree;

  sub run-one {
    # Run one test, call run-one again when it ends,
    # thus maintaining $degree active tests at a time
    ...
  }
}
sub run-one {
    # Run one test, call run-one again when it ends,
    # thus maintaining $degree active tests at a time
    ...
}
Run test files in parallel, envelope output

```perl
sub run-one {
    # If there's no more tests, just return
    my $test = @tests.shift // return;
    # Otherwise, run the test and collect the output
    ...
}
```
Run test files in parallel, envelope output

sub run-one {
    # If there's no more tests, just return
    my $test = @tests.shift // return;
    # Otherwise, run test, collect, and send output
    my $proc = Proc::Async.new('perl6', '-Ilib', $test);
    my @output = "FILE: $test";
    whenever $proc.stdout.lines {
        push @output, "OUT: \$_";
    }
    whenever $proc.stderr.lines {
        push @output, "ERR: \$_";
    }
    whenever $proc.start {
        push @output, "EXIT: {.exitcode}";
        say @output.join("\n");
        ...
    }
}

sub run-one {
    # If there's no more tests, just return
    my $test = @tests.shift // return;
    # Otherwise, run test, collect, and send output
    my $proc = Proc::Async.new('perl6', '-Ilib', $test);
    my @output = "FILE: $test";
    whenever $proc.stdout.lines {
        push @output, "OUT: \$_";
    }
    whenever $proc.stderr.lines {
        push @output, "ERR: \$_";
    }
    whenever $proc.start {
        push @output, "EXIT: {.exitcode}";
        say @output.join("\n");
        # Since this test is done, trigger one more
        run-one();
    }
}
Run test files in parallel, envelope output

```perl
react {
    sub run-one {
        my $test = @tests.shift // return;
        my $proc = Proc::Async.new('perl6', '-Ilib', $test);
        my @output = "FILE: $test";
        whenever $proc.stdout.lines {
            push @output, "OUT: $_";
        }
        whenever $proc.stderr.lines {
            push @output, "ERR: $_";
        }
        whenever $proc.start {
            push @output, "EXIT: {.exitcode}";
            say @output.join("\n");
            run-one();
        }
    }
    run-one for 1..$degree;
}
```
Run test files in parallel, envelope output

```
react {
    sub run-one {
        my $test = @tests.shift // return;
        my $proc = Proc::Async.new('perl6', '-Ilib', $test);
        my @output = "FILE: $test";
        whenever $proc.stdout.lines {
            push @output, "OUT: ":$_; 
        }
        whenever $proc.stderr.lines {
            push @output, "ERR: ":$_; 
        }
        whenever $proc.start {
            push @output, "EXIT: { .exitcode }";
            say @output.join("\n");
            run-one();
        }
    }
    run-one for 1..$degree;
}
```
Run test files in parallel, envelope output

```perl
react {
  sub run-one {
    my $test = @tests.shift // return;
    my $proc = Proc::Async.new('perl6', '-Ilib', $test);
    my @output = "FILE: $test";
    whenever $proc.stdout.lines {
      push @output, "OUT: " . ";"
    }
    whenever $proc.stderr.lines {
      push @output, "ERR: " . ";"
    }
    whenever $proc.start {
      push @output, "EXIT: {.exitcode}";
      say @output.join("\n");
      run-one();
    }
  }
  run-one for 1..$degree;
}
```

...so it can terminate when there are no more active subscriptions
Run test files in parallel, envelope output

```perl
react {
    sub run-one {
        my $test = @tests.shift // return;
        my $proc = Proc::Async.new('perl6', '-Ilib', $test);
        my @output = "FILE: $test";
        whenever $proc.stdout.lines {
            push @output, "OUT: $_";
        }
        whenever $proc.stderr.lines {
            push @output, "ERR: $_";
        }
        whenever $proc.start {
            push @output, "EXIT: {.exitcode}";
            say @output.join("\n");
        run-one();
        }
    }
    run-one for 1..$degree;
}
```

Arrays are not threadsafe. Concurrency control???
A react processes one event at a time

So all state inside the react block is covered by this concurrency control

The setup phase - running the react block to establish initial subscriptions - is considered as an initial event
react {
    sub run-one {
        my $test = @tests.shift // return;
        my $proc = Proc::Async.new('perl6', '-Ilib', $test);
        my @output = "FILE: $test";
        whenever $proc.stdout.lines {
            push @output, "OUT: $_";
        }
        whenever $proc.stderr.lines {
            push @output, "ERR: $_";
        }
        whenever $proc.start {
            push @output, "EXIT: { .exitcode }";
            say @output.join("\n");
            run-one();
        }
    }
    run-one for 1..$degree;
}
One more thought experiment:

If `react` is like for, in so far as we process values and do side-effects, then what is like `map`, where we produce a result sequence?
In Perl 6, we call that a supply block
sub envelope-process(Proc::Async $proc --> Supply) {
  supply {
    whenever $proc.stdout.lines {
      emit "OUT: \$_";
    }
    whenever $proc.stderr.lines {
      emit "ERR: \$_";
    }
    whenever $proc.start {
      emit "EXIT: {.exitcode}";
    }
  }
}
Returns a Supply (the Perl 6 type for an asynchronous stream)

Runs the supply body each time that Supply is tapped (subscribed to)

Think of it like subscribing making a new "instance"
Call the sub, subscribe to what it returns

```perl
react {
    sub run-one {
        my $test = @tests.shift // return;
        my $proc = Proc::Async.new('perl6', '-Ilib', $test);
        my @output = "FILE: $test";
        whenever envelope-process($proc) {
            push @output, $_;
            LAST {
                say @output.join("\n");
                run-one();
            }
        }
    }
    run-one for 1..$degree;
}
```
LAST runs when the stream ends

```
react {
    sub run-one {
        my $test = @tests.shift // return;
        my $proc = Proc::Async.new('perl6', '-Ilib', $test);
        my @output = "FILE: $test";
        whenever envelope-process($proc) {
            push @output, $_;
            LAST {
                say @output.join("\n");
                run-one();
            }
        }
    }
    run-one for 1..$degree;
}
```
Final challenge:

How can we implement a timeout mechanism for hanging processes?
# Exception type to throw if we time out.
class X::Timeout is Exception {}

sub timeout(Supply $source, Real $seconds --> Supply) {
  supply {
    # Pass on values. If $seconds have elapsed, # throw exception.
    ...
  }
}
# Exception type to throw if we time out.
class X::Timeout is Exception {}

sub timeout(Supply $source, Real $seconds --&gt; Supply) {
    supply {
        # Pass on values.
        whenever $source {
            emit $_;
        }
        # If $seconds have elapsed, throw exception.
        ...
    }
}
# Exception type to throw if we time out.
class X::Timeout is Exception {}

sub timeout(Supply $source, Real $seconds --> Supply) {
supply {
    # Pass on values.
    whenever $source {
        emit $_;
    }
    # If $seconds have elapsed, throw exception.
    whenever Promise.in($seconds) {
        die X::Timeout.new;
    }
}
}
# Exception type to throw if we time out.
class X::Timeout is Exception {}

sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        # Pass on values.
        whenever $source {
            emit $_;
        }
        # If $seconds have elapsed, throw exception.
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}

But we'll always wait for the timeout?!
# Exception type to throw if we time out.
class X::Timeout is Exception {}

sub timeout(Supply $source, Real $seconds --> Supply) {
    supply {
        # Pass on values.
        whenever $source {
            emit $_;
        # Use done to terminate the supply block
            LAST done;
        }
        # If $seconds have elapsed, throw exception.
        whenever Promise.in($seconds) {
            die X::Timeout.new;
        }
    }
}
Using timeout is easy...but where does the exception go?

```perl
react {
    sub run-one {
        my $test = @tests.shift // return;
        my $proc = Proc::Async.new('perl6', '-Ilib', $test);
        my @output = "FILE: $test";
        whenever timeout(envelope-process($proc), 2) {
            push @output, $_;
            LAST {
                say @output.join("\n");
                run-one();
            }
        }
    }
    run-one for 1..$degree;
}
```
Unhandled exceptions cause all subscriptions to be closed

The exception is rethrown in the code that triggered the react block
QUIT catches asynchronous exceptions

react {
    sub run-one {
        my $test = @tests.shift // return;
        my $proc = Proc::Async.new('perl6', '-Ilib', $test);
        my @output = "FILE: $test";
        whenever timeout(envelope-process($proc), 2) {
            push @output, $._;
            LAST {
                say @output.join("\n");
                run-one();
            }
        }
        QUIT {
            when X::Timeout {
                $proc.kill;
                push @output, "TIMEOUT";
                say @output.join("\n");
                run-one();
            }
        }
    }
    run-one for 1..$degree;
}
react {
  sub run-one {
    my $test = @tests.shift // return;
    my $proc = Proc::Async.new('perl6', '-Ilib', $test);
    my @output = "FILE: $test";
    whenever timeout(envelope-process($proc), 2) {
      push @output, $_;
      LAST {
        say @output.join("\n");
        run-one();
      }
    }
    QUIT {
      when X::Timeout {
        $proc.kill;
        push @output, "TIMEOUT";
        say @output.join("\n");
        run-one();
      }
    }
  }
  run-one for 1..$degree;
}

Looks like duplicated code to me!
react {
    sub run-one {
        my $test = @tests.shift // return;
        my $proc = Proc::Async.new('perl6', '-Ilib', $test);
        my @output = "FILE: $test";
        whenever timeout(envelope-process($proc), 2) {
            push @output, $_;
            LAST handle-termination;
            QUIT {
                when X::Timeout {
                    $proc.kill;
                    push @output, "TIMEOUT";
                    handle-termination;
                }
            }
        }
    }

    sub handle-termination() {
        say @output.join("\n");
        run-one();
    }
}

run-one for 1..$degree;
In summary....
Attaches to react or supply whenever
Attaches to react or supply whenever
Concurrency control
Completion and error propagation

Side-effects    New stream

Attaches to react or supply

whenever
Concurrency control
Completion and error propagation

Side-effects

New stream

Attaches to react or supply

whenever

Works like a loop (last, next, LAST)
End all subscriptions with done
Or, in other words...
A mechanism for consuming zero or more asynchronous streams (often called "reactive streams"), providing concurrency control though one-message-at-a-time processing, automatically handling error and completion propagation, and managing subscriptions, with the option of producing a new stream of values as a result.
Thank you!

Questions?