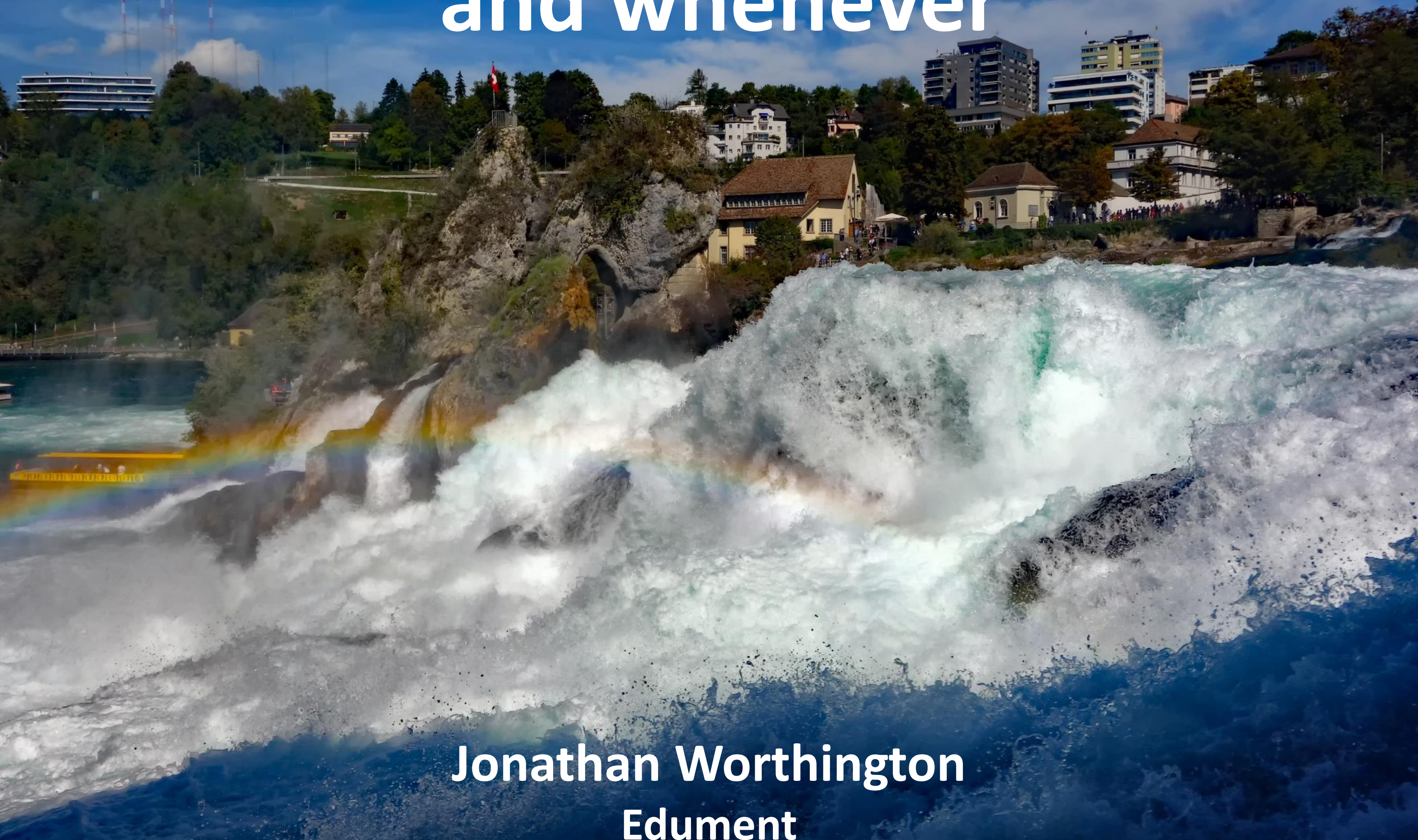
Understanding react, supply, and whenever



Jonathan Worthington

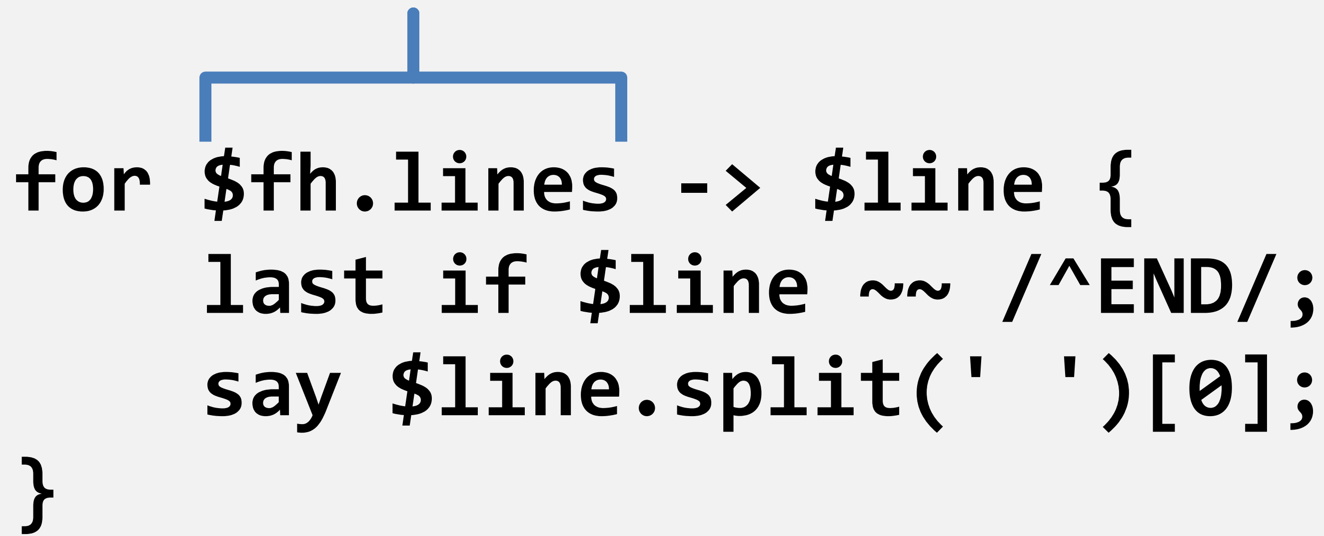
Edument

Well, they're just...

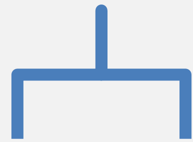
A mechanism for consuming zero or more asynchronous streams (often called "reactive streams"), providing concurrency control though one-message-at-a-time processing, automatically handling error and completion propagation, and managing subscriptions, with the option of producing a new stream of values as a result

Any questions?

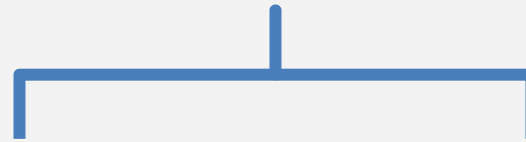
Let's talk about `for` loops.


Iterable
data source
for $fh.lines -> $line {
    last if $line ~~ /^END/;
    say $line.split(' ')[0];
}

Pull one value

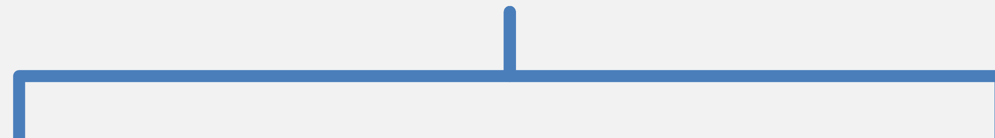


Run the loop body



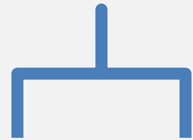
```
for $fh.lines -> $line {
    last if $line ~~ /^END/;
    say $line.split(' ')[0];
}
```

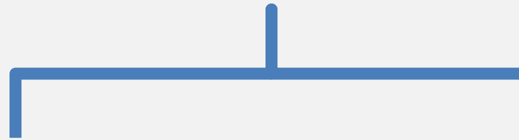
These happen in lockstep



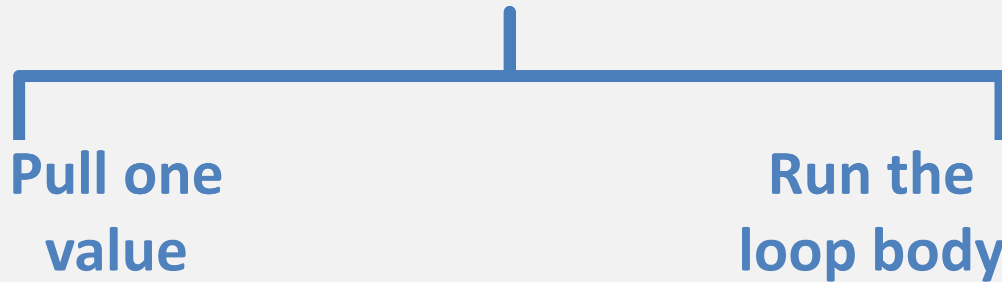
Pull one value

Run the loop body



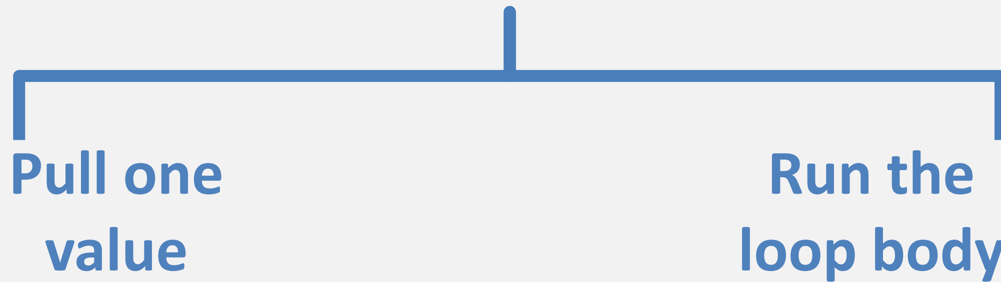


```
for $fh.lines -> $line {
    last if $line ~~ /^END/;
    say $line.split(' ')[0];
}
```




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

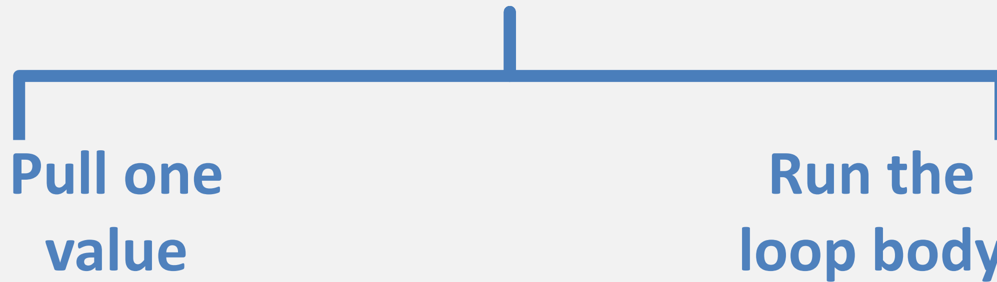
Lost interest? Just stop. GC eats iterator.




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

Lost interest? Just stop. GC eats iterator.

Next statement runs after completion of the loop




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

Lost interest? Just stop. GC eats iterator.

Next statement runs after completion of the loop

Synchronous programming

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?

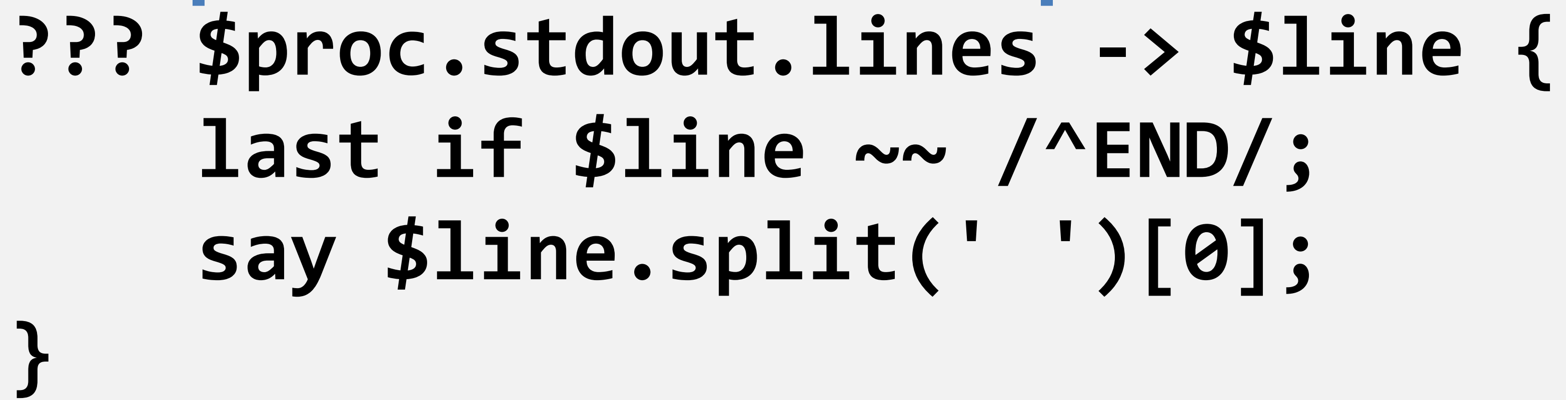
The idea of asynchronous streams, like iterators, captures so many things...

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

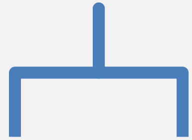
Thought experiment:

What if we were to have a loop-like construct for asynchronous streams?

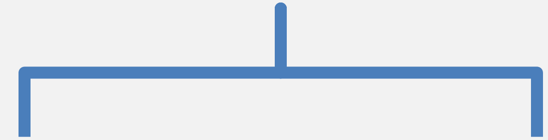
Observable data source



Subscribe to events



Run code on an event



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

Events occur any time, even on any thread, after subscription

Subscribe to events

Run code on an event

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

Events occur any time, even on any thread, after subscription

Subscribe to events

Run code on an event

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

Unsubscribe

Events occur any time, even on any thread, after subscription

Subscribe to events

Run code on an event

```
??? $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

Run code on an event

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

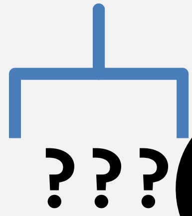
Unsubscribe

Next statement...uhh...hmmm

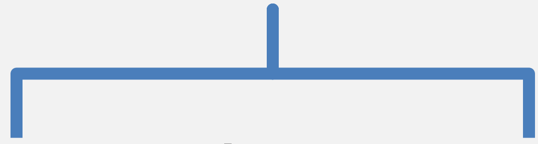
Asynchronous programming

Events occur any time, even on any thread, after subscription

Subscribe to events



Run code on an event



```
???            ines -> $line {
               ~~ /^END/;
    s          lit(' ')[0];
}
```

Unsubscribe

We'll need a means of concurrency control

Next statement...uhh...hmmm

Asynchronous programming

Events occur any time, even on any thread, after subscription

Subscribe to events

Could await the stream...

Run code on an event

```
??? $proc            -> $line {
    last i           /^END/;
    say $  ne.sp  t(' ')[0];
}
```

Unsubscribe

Next statement...uhh...hmmm

Asynchronous programming

Events occur any time, even on any thread, after subscription

Subscribe to events

Run code on an event

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

```
??? $proc                  -> $line {
    last i              /^END/;
    say $ ine.sp  t(' ')[0];
}
```

Unsubscribe

Next statement...uhh...hmmm

Asynchronous programming

In Perl 6, `whenever` is an asynchronous loop construct

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

Subscribes to the stream

Runs the body on each event

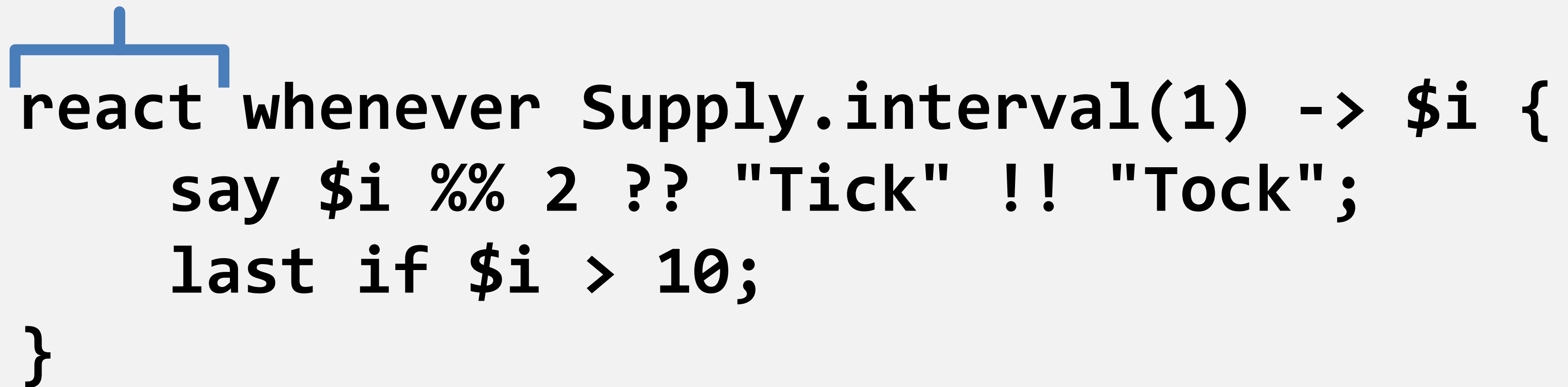
Unsubscribes on `last`

`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



As with other statement prefixes, `react` takes a block *or* a statement

```
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

```
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 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 {
    ...
}
```

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
        ...
    }
}
```

Run test files in parallel, envelope output

```
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

```
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");
        ...
    }
}
```

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");
        # Since this test is done, trigger one more
        run-one();
    }
}
```

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

```
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;
}
```

Each whenever registers its subscription with the enclosing react...

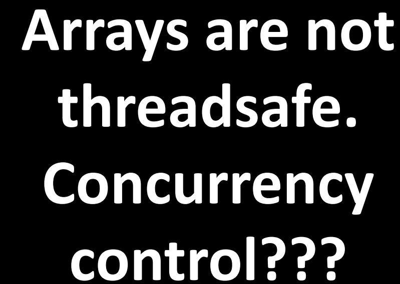
Run test files in parallel, envelope output

```
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;
}
```

...so it can terminate when there are no more active subscriptions

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: $t[illegible]t";
        whenever $proc.stdout.lines {
            push @output, "OUT: $_
        }
        whenever $proc.stderr.lines {
            push @output, "ERR: $_";
        }
        whenever $proc.start {
            push @output, "EXIT: {.e
            say @output.join("\n");
            run-one();
        }
    }
    run-one for 1..$degree;
}
```



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

Can we factor out the enveloping part?

```
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

```
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:

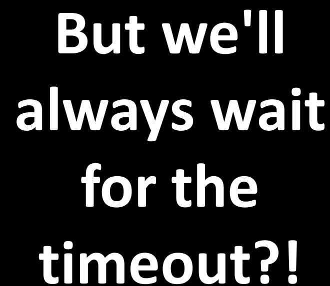
How can we implement a timeout mechanism for hanging processes?

```
# 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 --> 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;
        }
    }
}
```



```
# 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?

```
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

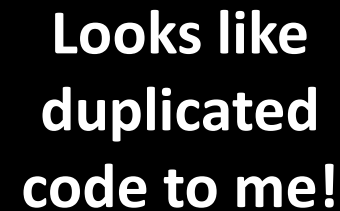
The exception is rethrown in the code that triggered the `react` block

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;
}
```

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;
}
```



Tidy up, and we're done

```
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….

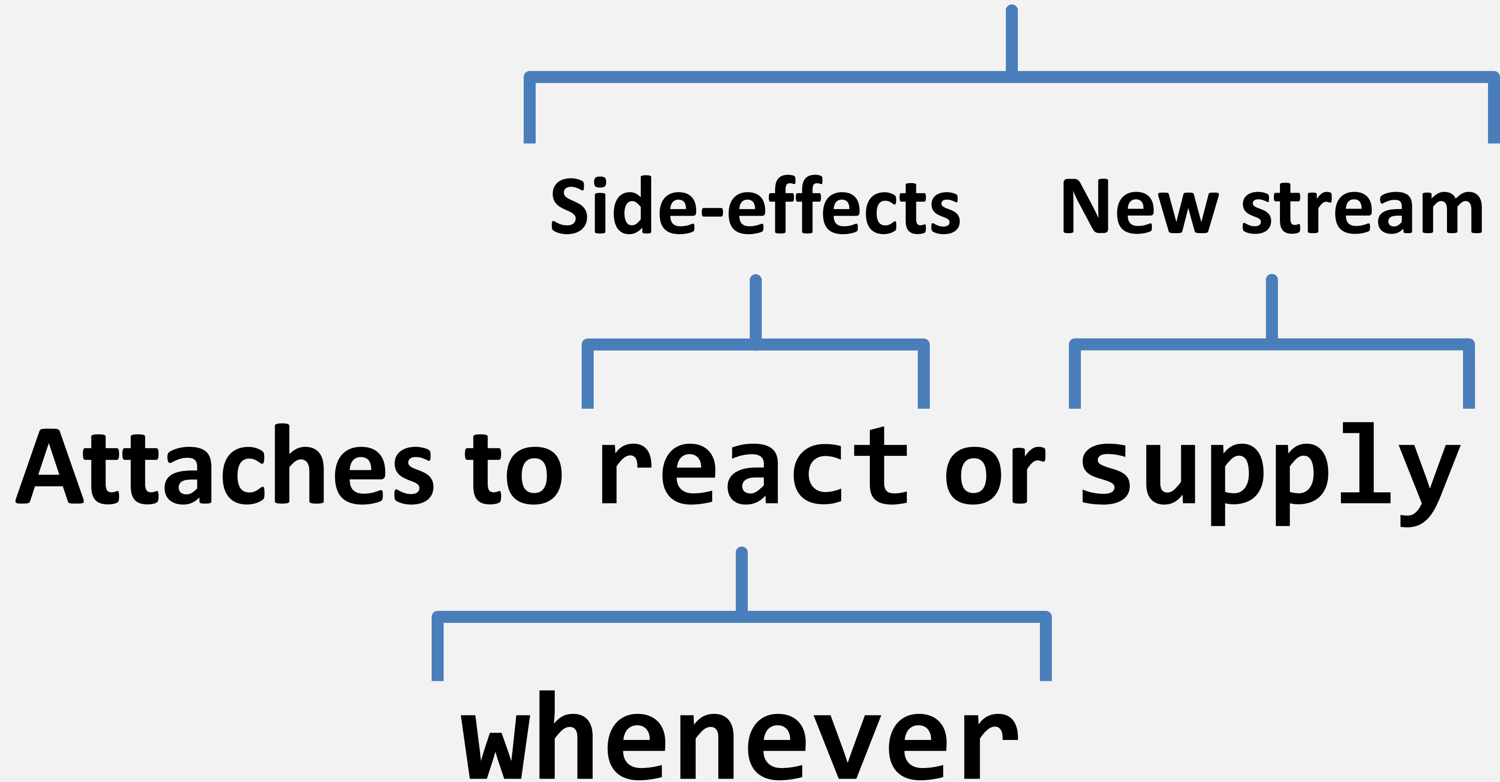
Attaches to `react` or `supply`

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

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?