Objects ∩ Concurrency

Jonathan Worthington

Hi. I'm Jonathan.

Perl 6 concurrency

The work so far is mostly on *functional* constructs

Focus on computations that produce results "in the future", and avoid having state

Promises

Things that produce a single result in the future (some code, a one-shot timer, a process exit code…)

```
my $proc = Proc::Async.new('tracert', 'jnthn.net');
my $promise = $proc.start;
my $exit = await $promise;
```

Promise combinators

Combine promises in various useful ways; here we mix an async process and time

```
my $proc = Proc::Async.new('tracert', 'jnthn.net');
my $tracert-done = $proc.start;
await Promise.anyof($tracert-done, Promise.in(10));
$proc.kill unless $tracert-done;
```

Supplies

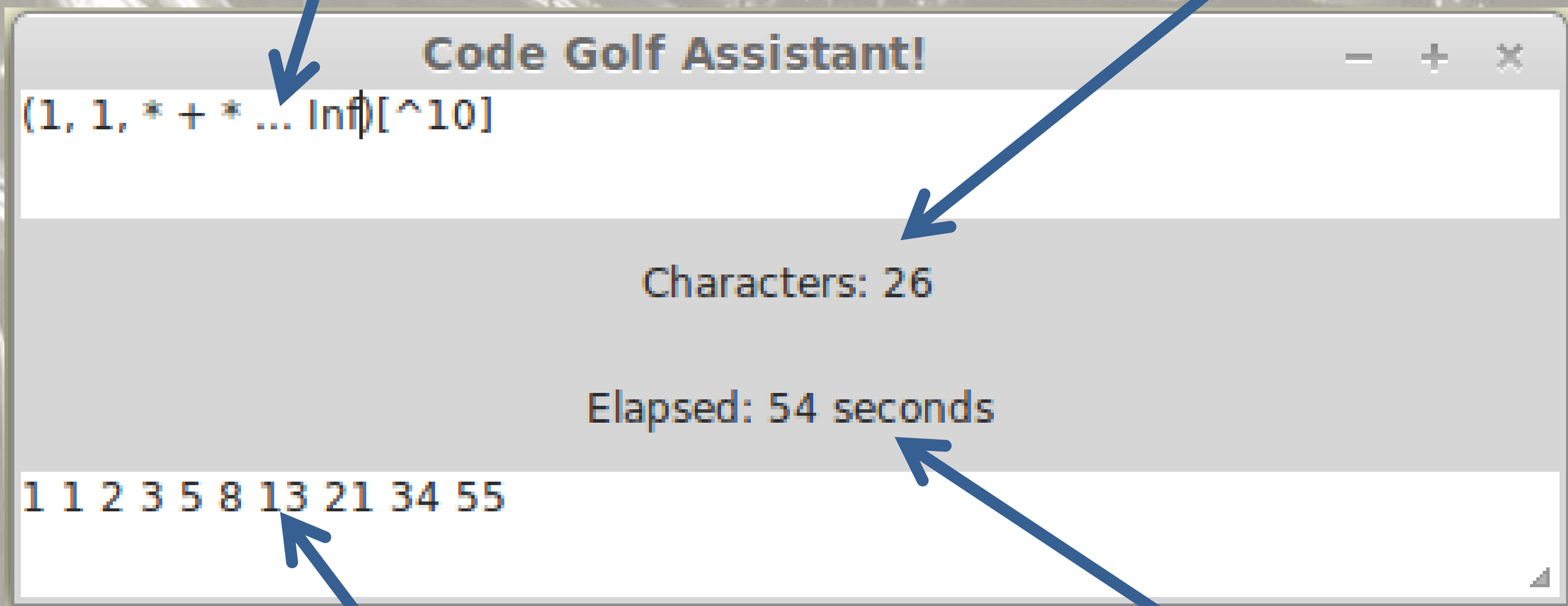
Represents things that may produce many values over time, asynchronously, and maybe from many threads

```
my $secs = Supply.interval(1);
my $tt = $secs.map({ $_ %% 2 ?? 'Tick' !! 'Tock' });
$tt.tap(&say);
sleep 10;
```

Example: code golf assistant

Type code here

Char count updates automatically



Run code in background thread and show result

Show how much time I've wasted

Example: code golf assistant

UI setup code

```
my $app = GTK::Simple::App.new(
    title => 'Code Golf Assistant!');

$app.set_content(GTK::Simple::VBox.new(
    my $source  = GTK::Simple::TextView.new(),
    my $chars   = GTK::Simple::Label.new(
        text => 'Characters: 0'),
    my $elapsed = GTK::Simple::Label.new(),
    my $results = GTK::Simple::TextView.new(),
));
```

Example: code golf assistant

UI events can be seen as an asynchronous sequence of values, so supplies fit well!

```
$source.changed.tap({
    $chars.text =
        "Characters: $source.text.chars()";
});
```

Example: code golf assistant

Ticking seconds are just an interval - but we must update the UI on the correct thread!

```
Supply.interval(1).schedule_on(
    GTK::Simple::Scheduler
).tap(-> $secs {
    $elapsed.text = "Elapsed: $secs seconds";
});
```

Example: code golf assistant

When code is unchanged for a second, eval it on a thread…

```
$source.changed.stable(1).start({
    (try EVAL .text) // $!.message
})
…
```

Example: code golf assistant

…and show (latest!) result on the UI - using the UI thread

```
$source.changed.stable(1).start({
    (try EVAL .text) // $!.message
}).migrate().schedule_on(
    GTK::Simple::Scheduler
).tap(
    { $results.text = $_ }
);
```

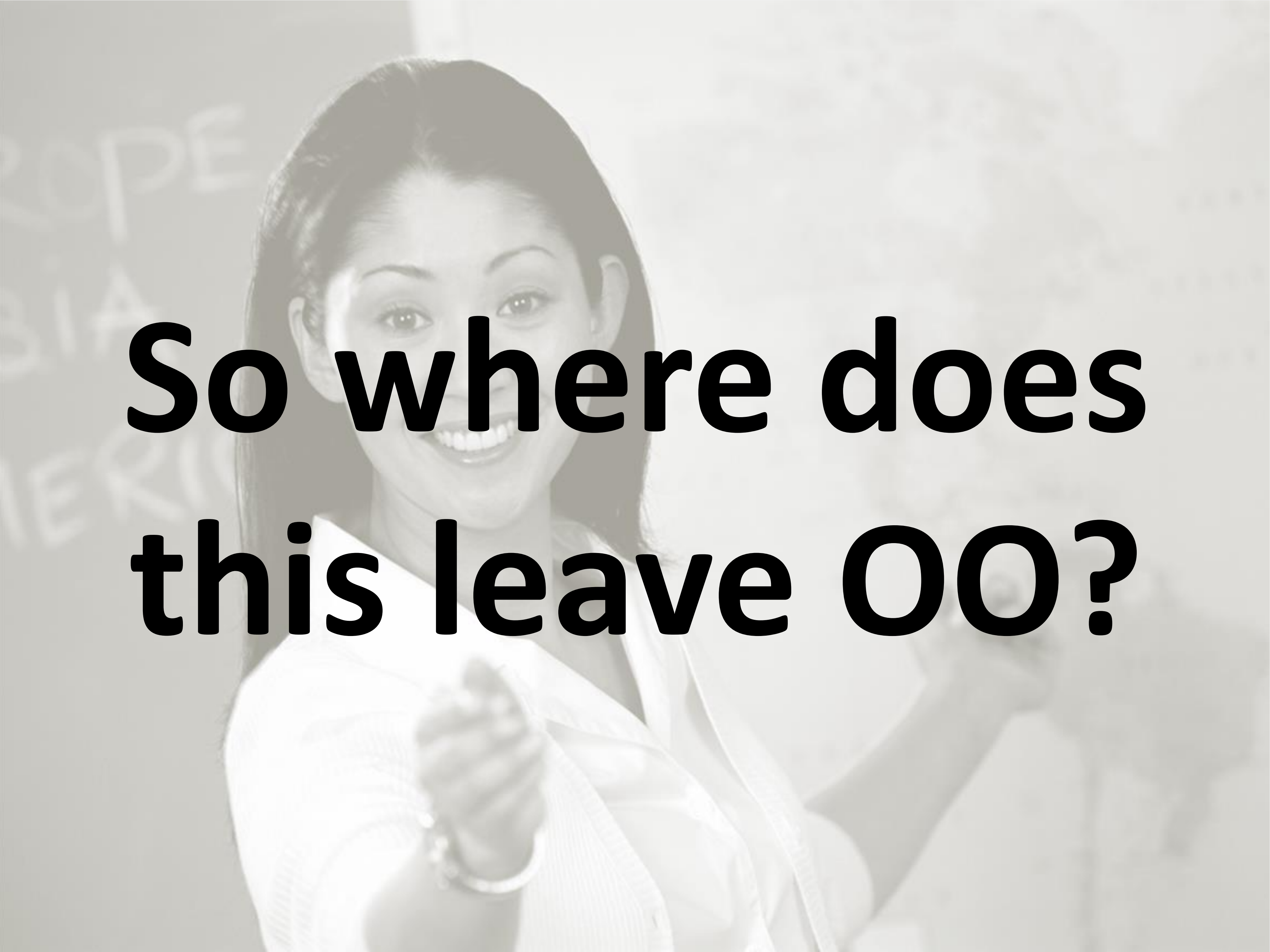
Threads and mutable shared state is a source of bugs

→

Factor synchronization and shared state out of user code

→

WIN!



So where does this leave OO?

If state tends to make concurrency hard…

…and objects are stateful…

…are objects and concurrency a bad mix?


NEIN!

What are objects *really* about?

Hiding state inside of an encapsulated boundary

Defining invariants on that state, and ensuring mutating methods always uphold it

Good objects bound state

State protected inside the object, and interacted with through calling methods

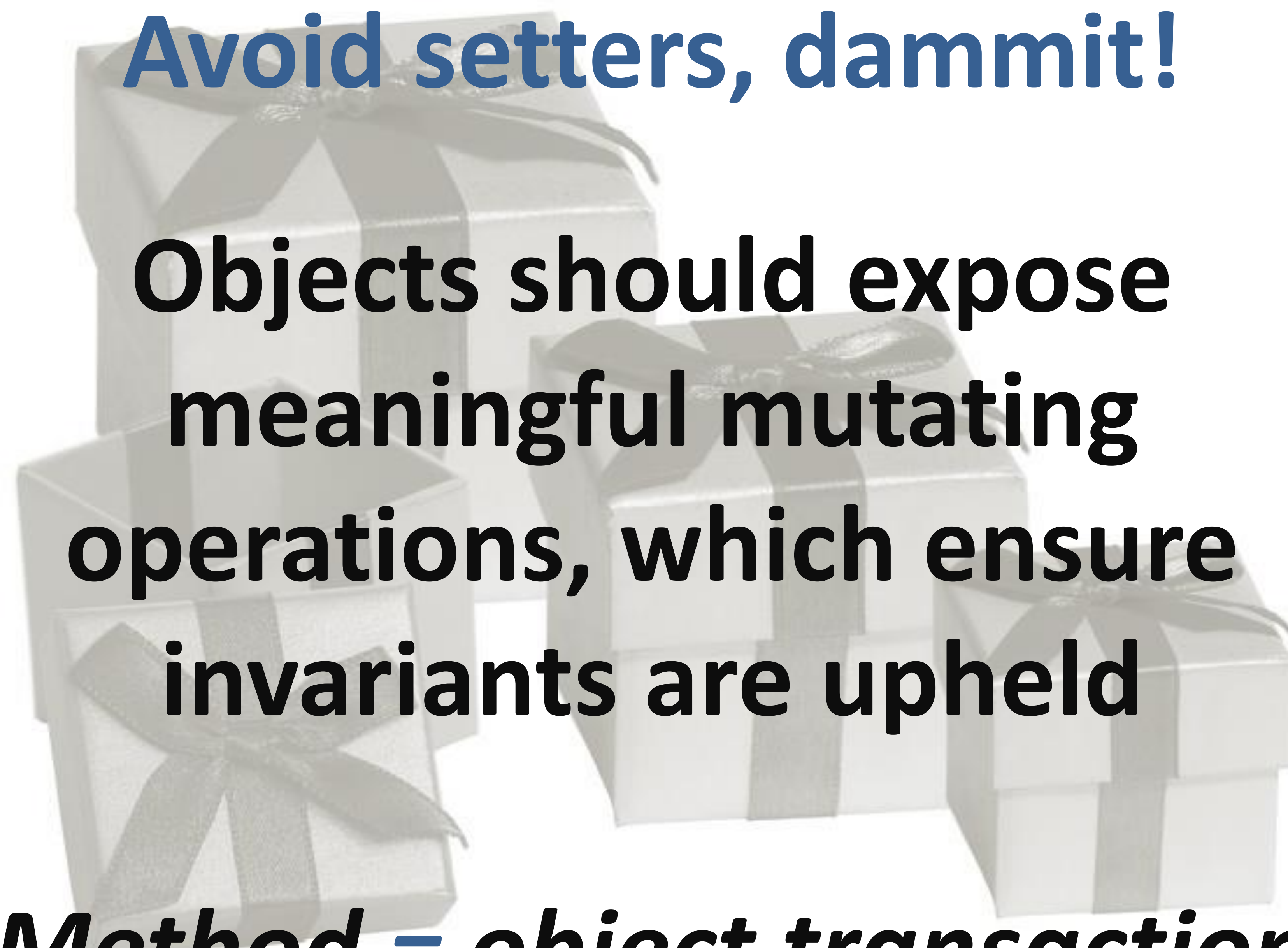
→

Method call is a natural point of concurrency control

Avoid getters, dammit!

Getters are outright dangerous on mutable attributes

Even on immutable ones, risk logic leaks. Remember: *tell* objects things, <u>don't ask</u>!



Avoid setters, dammit!

Objects should expose meaningful mutating operations, which ensure invariants are upheld

Method = object transaction

3 approaches

There's more than one way to put objects to work in a concurrent situation.

We'll examine three of them, with different use cases.

Monitors

Just like classes, they have attributes and methods

But only one thread may be inside the monitor's methods at a time (so recursion is OK)

Concurrent calls block

If a thread is running one of the monitor's methods, other callers must queue up

Example: IP filter

Use the Monitors module, which adds a `monitor` package declarator

```
use OO::Monitors;

monitor IPFilter {
    ...
}
```

Example: IP filter

Declare state, knowing only one thread can use it at a time

```
monitor IPFilter {
    has %!blacklist;
    has %!active;
    has $.limit = 10;
    has $.blocked = 0;

    ...
}
```

Example: IP filter

Write methods that work with that state

```
method add-to-blacklist($ip) {
    %!blacklist{$ip} = True;
}

method remove-from-blacklist($ip) {
    %!blacklist{$ip}:delete;
}
```

Example: IP filter

```
method should-start-request($ip) {
    if %!blacklist{$ip} ||
           (%!active{$ip} // 0) == $.limit {
        $!blocked++;
        return False;
    }
    %!active{$ip}++;
    return True;
}

method end-request($ip) {
    %!active{$ip}--;
}
```

Simulating 4 request threads

```
my $phil = IPFilter.new(limit => 5);

my @ips = '12.13.14.' <<~<< ^128;
$phil.add-to-blacklist(@ips.pick);
await do for ^4 {
    start {
        for ^100 {
            $phil.should-start-request: @ips.pick;
            $phil.end-request:          @ips.pick;
        }
    }
}

say "Blocked $phil.blocked() requests";
```

Monitors with conditions

Sometimes, a monitor can not proceed until another thread makes a (separate) change

Conditions allow us to handle such scenarios

Build a bounded queue

Adds should block if the queue is full, and removes should block if the queue is empty

Declare the conditions

Declare the monitor with two wait conditions: `not-full` and `not-empty`

```
monitor PriorityQueue
        is conditioned(< not-full not-empty >) {
    ...
}
```

Add the state

Declare queue tasks storage along with a task limit

```
monitor PriorityQueue
        is conditioned(< not-full not-empty >) {
    has @!tasks;
    has $.limit = die "Must specify a limit";
    ...
}
```

Adding a task

Wait for `not-full` if needed, add task, meet `not-empty`

```
method add-task($task) {
    while @!tasks.elems == $!limit {
        wait-condition <not-full>;
    }
    @!tasks.push($task);
    meet-condition <not-empty>;
}
```

Taking a task

Wait for `not-empty` if needed, take task, meet `not-full`

```
method take-task() {
    until @!tasks {
        wait-condition <not-empty>;
    }
    meet-condition <not-full>;
    return @!tasks.shift;
}
```

Monitors: sometimes good

Relatively simple mechanism and programming model

Easy to go from a (well designed) class to a monitor

Monitors: sometimes bad

Under contention, monitors cause threads to block

Vulnerable to deadlock, though much less so than unstructured application of locks

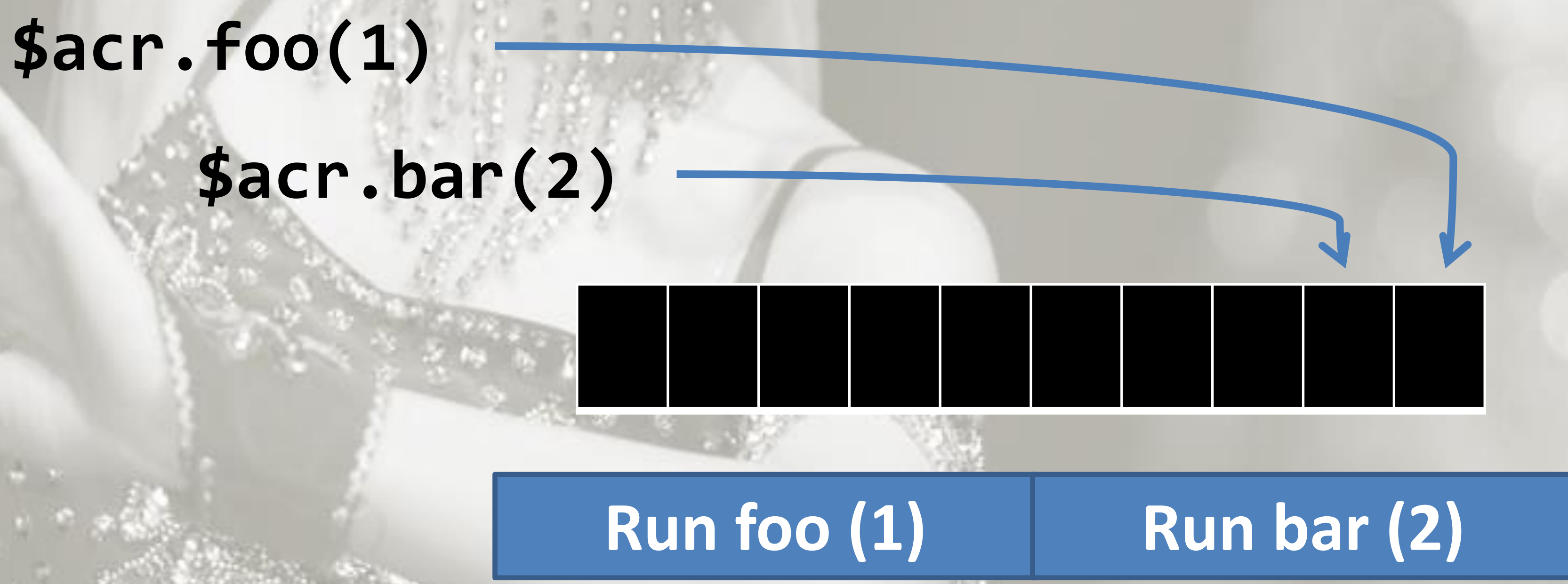
Actors

As with monitors, only one thread can be in a given method at a time

However, the method calls are asynchronous/non-blocking

How Actors (basically) work

Calls are put in a "queue", and a (pool) thread processes them



Example: logging

Want to log events at a range of severity levels

```
enum Severity <Fatal Error Warning Notice>;
```

Many threads can log, and don't want to block execution

Stubbing the actor

Use the Actors module, declare the actor, and give it state using attributes

```
use OO::Actors;

actor EventLog {
    has %!events-by-level{Severity};
    ...
}
```

Methods

```
method log(Severity $level, Str $message) {
    push %!events-by-level{$level}, $message;
}

method latest-entries(Severity $level-limit) {
    my @found;
    for %!events-by-level.kv -> $level, @messages {
        next if $level > $level-limit;
        push @found, @messages;
    }
    return @found;
}
```

Using the actor

Can have many threads calling methods on it. Note they are executed asynchronously!

```
my $el = EventLog.new;
await do for ^4 {
    start {
        $el.log(Severity.pick, 'OMG') for ^100;
    }
}
```

Querying the actor

Since execution is async, the method call can't return the result! Instead, it returns a `Promise` that will be kept with the result in the future.

```
say await $el.latest-entries(Fatal);
```

Actors go much further

This is only a very basic implementation. Actors also have supervision, which is how they manage to work robustly and recover from failures. But that's for a future talk… ☺

Actors: great but different

Solve the blocking issues associated with monitors

However, need their callers to be designed expecting asynchronous execution also

Considering mutating methods

Mutating methods typically consist of validation (to ensure we won't break invariants) followed by mutation

```
die "Seat $seat taken" if %!seat-taken{$seat};
%!seat-taken{$seat} = True;
```

Introducing events

We could instead have methods validate, and then produce an event describing the decision reached

```
die "Seat $seat taken" if %!seat-taken{$seat};
return SeatSelected.new(:$.id, :$seat);
```

Event application

We could then write a separate event application method, which grabs data from the event and mutates the object

```
multi method apply(SeatSelected $e) {
    %!seat-status{$e.seat} = True;
}
```

Persistence through events

Given a stream of events, we can replay them to build up an object with the current state

We can in turn use it to validate the next operation

Optimistic concurrency

Since we always *work against a fresh copy* of the object, if we lose the race to produce the next event, we can simply produce a fresh object and *try the operation over again*!

A quick example: plane seats

Let's consider a simple plane seat selection object

Events

```
class FlightOpened {
    has $.id;
    has $.flight-number;
    has @.available-seats;
}

class SeatSelected {
    has $.id;
    has $.seat;
    has $.passenger-name;
}
```

Exceptions

```
class X::PlaneSeatingPlan::BadSeat is Exception {
    has $.seat;
    method message() {
        "No such seat $!seat"
    }
}

class X::PlaneSeatingPlan::SeatTaken is Exception {
    has $.seat;
    method message() {
        "Seat $!seat is already taken"
    }
}
```

The aggregate

We inherit from a class Aggregate, which provides event application logic

```
use Evject;

class PlaneSeatingPlan is Aggregate {
    has %!seat-status;
    ...
}
```

Opening a flight

This method hasn't much to validate, and so simply produces an event

```
method open-flight($flight-number,
                   @available-seats) {
    return FlightOpened.new(:$.id, :$flight-number,
                            :@available-seats);
}
```

Picking a seat

Validates the seat is valid and free, then produces an event

```
method choose-seat($seat, $passenger-name) {
    X::PlaneSeatingPlan::BadSeat.new(:$seat).throw
        unless %!seat-status{$seat}:exists;
    X::PlaneSeatingPlan::SeatTaken.new(:$seat).throw
        if defined %!seat-status{$seat};
    return SeatSelected.new(:$.id, :$seat,
                            :$passenger-name);
}
```

Event appliers

Update state based on events

```
multi method apply(FlightOpened $e) {
    for $e.available-seats -> $seat {
        %!seat-status{$seat} = Nil;
    }
}

multi method apply(SeatSelected $e) {
    %!seat-status{$e.seat} = $e.passenger-name;
}
```

Infrastructure

We need some way to store events, and something that loads objects, runs methods, and tries to save new events.

```
use InMemoryEventStore;
my $dom = Domain.new(
    event-store => InMemoryEventStore.new);
```

And finally…

```
my @seats  = 1..10 X~ <A C D F>;
$dom.process:
    PlaneSeatingPlan, 1,
    *.open-flight('SK123', @seats);

# Works fine
$dom.process:
    PlaneSeatingPlan, 1,
    *.choose-seat('2A', 'jnthn');

# Exception, seat taken
$dom.process:
    PlaneSeatingPlan, 1,
    *.choose-seat('2A', 'jnthn');
```

Events are awesome

Here, we used the concept of events to deal with both persistence and provide optimistic, non-blocking, concurrency control. Plus we can distribute the events!

Re-thinking "calling"

Some languages name method calls "message sends"

There's more than one way to send and process messages - some good for concurrency

In summary…

	Concurrency Model	Nature of call
Classes	No concurrency control	Synchronous, calls immediately
Monitors	Mutual exclusion	Synchronous, call may block
Actors	Mutual exclusion	Asynchronous (so non-blocking)
Event-Sourced Aggregates	Optimistic concurrency control	Synchronous, may fail and retry

Questions?