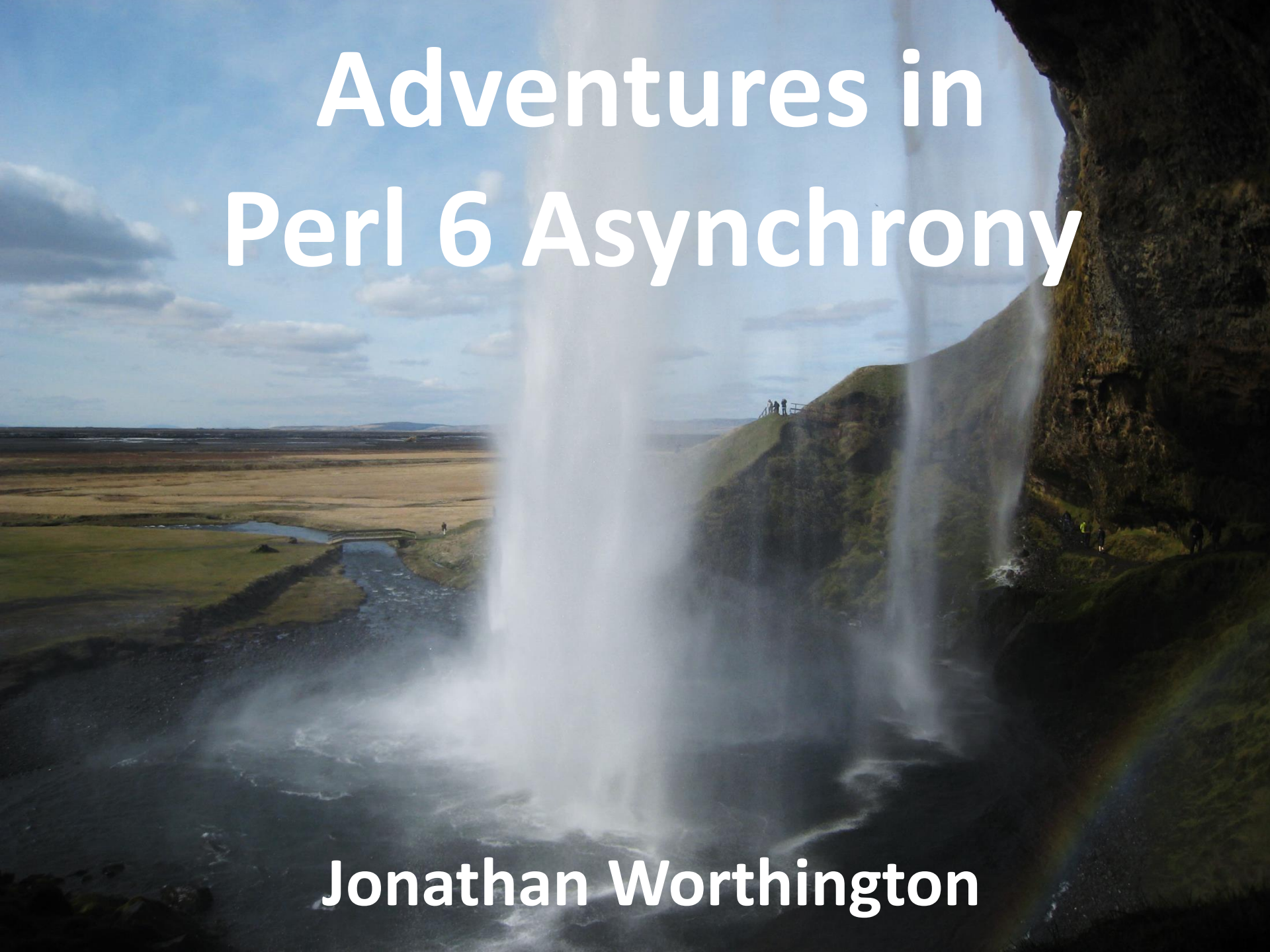


# Adventures in Perl 6 Asynchrony

Jonathan Worthington

# My original idea

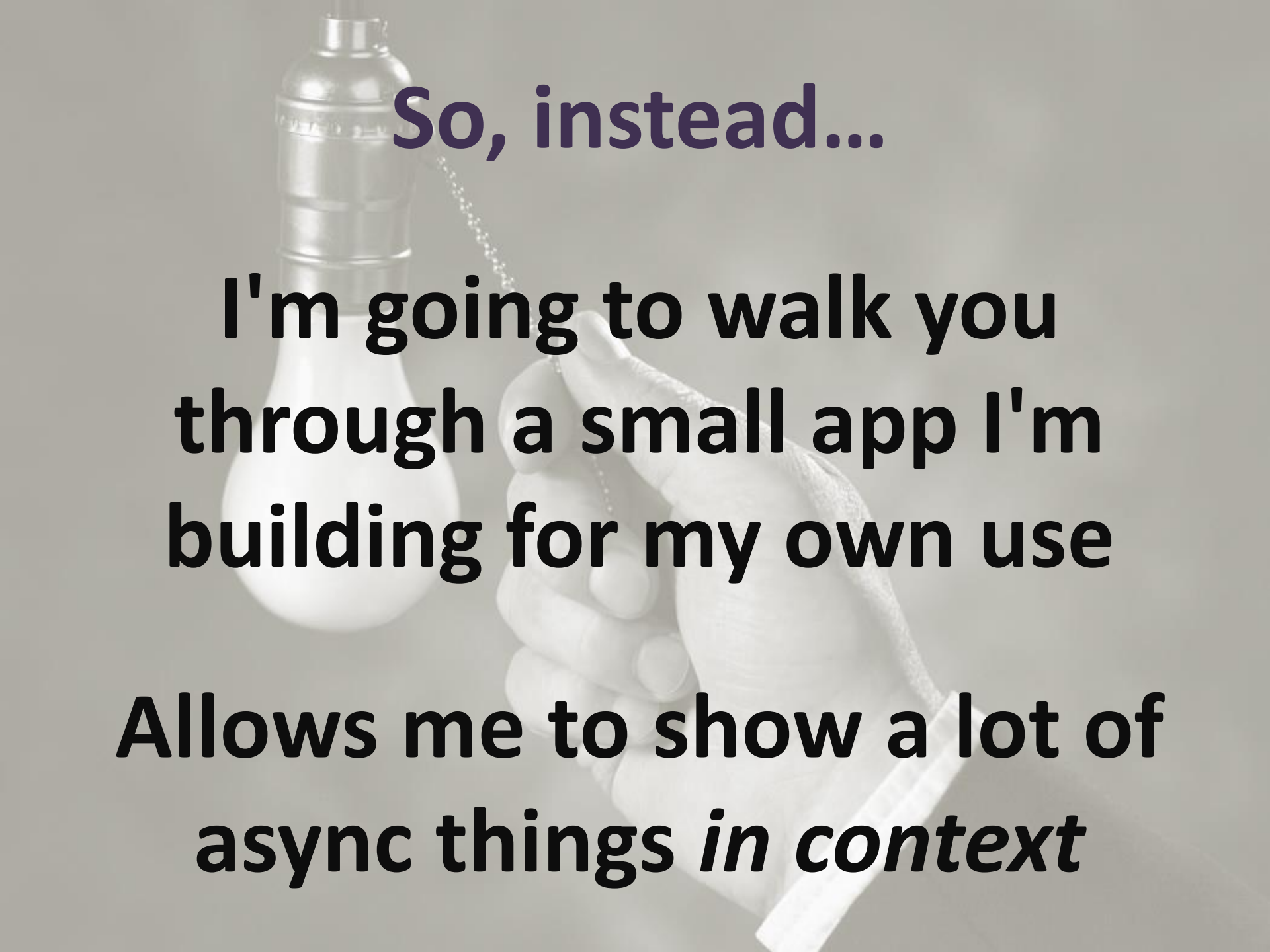
**Extoll the beautiful duality of iterators and observers**

**Give lots of little examples, showing off various features in relative isolation**

# But…

**Practice beats theory**

**Last year, the only thing I *could* show were isolated examples. Now we can do more interesting things…**



# So, instead…

**I'm going to walk you through a small app I'm building for my own use**

**Allows me to show a lot of async things *in context***

# I like to travel…

# …and collect photos…

# …and collect photos…

# …and collect photos…

# I wanted a small tool to…

**Categorize images by trip and places I went**

**Produce various sizes**

**Stick them on my server**

# Make use of the hardware

**Even my laptop is multi-core and hyper-threaded**

**Should be able to perform the image resizing in parallel, using multiple cores**

# Make use of the network

**Work on uploading a few images at a time**

**Will just `scp` them, so really this means juggling a few different processes**

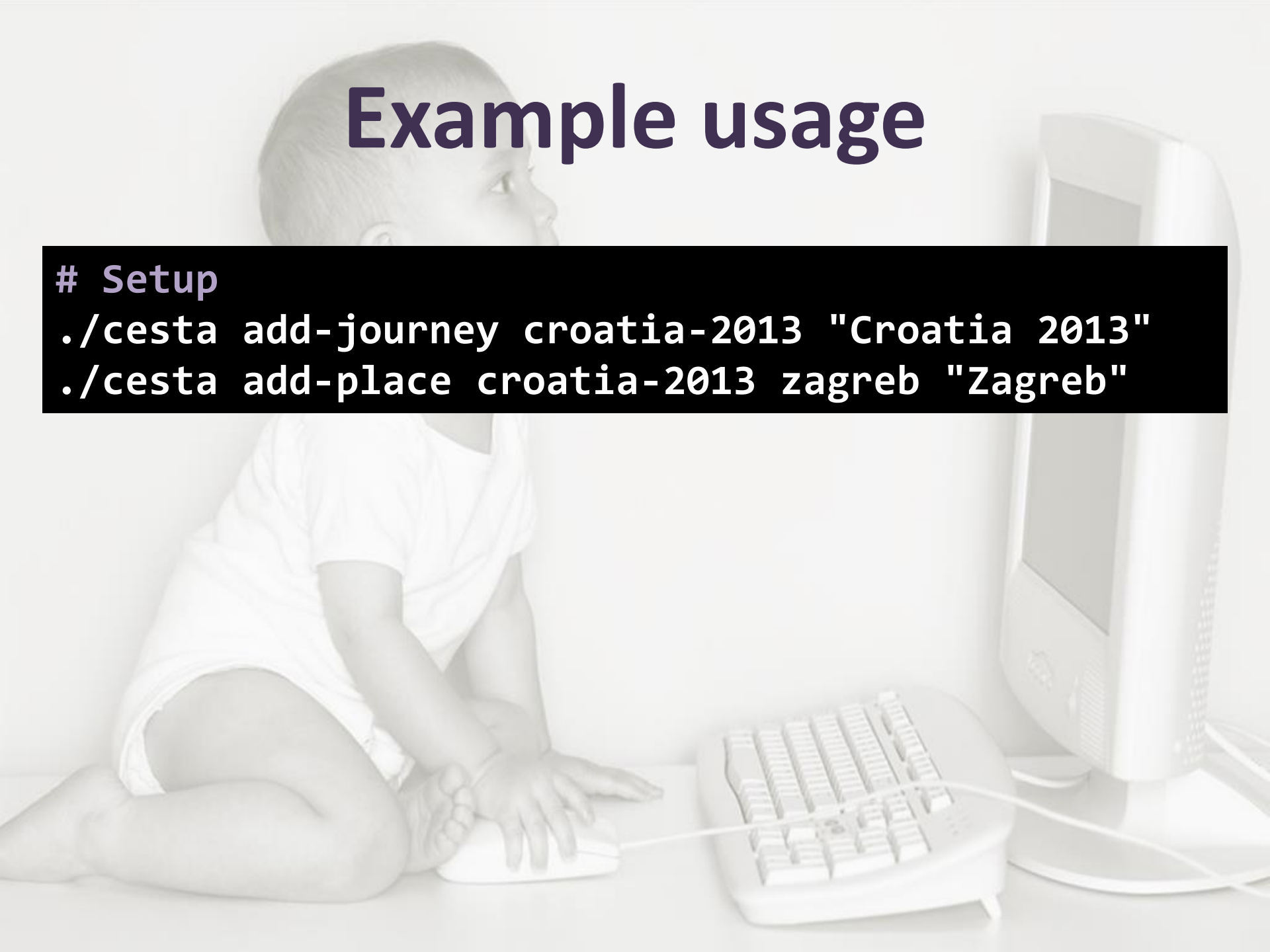
# Ctrl + C, and resume later

**Don't always have time to wait for all the uploading**

**Want to be able to suspend it at any point, and have it able to resume later**

# Example usage

```
# Setup
./cesta add-journey croatia-2013 "Croatia 2013"
./cesta add-place croatia-2013 zagreb "Zagreb"
```



# Example usage

```
# Setup
./cesta add-journey croatia-2013 "Croatia 2013"
./cesta add-place croatia-2013 zagreb "Zagreb"
```

```
# Add today's photos.
./cesta add-photos croatia-2013 zagreb ../today
```

# Example usage

```
# Setup
./cesta add-journey croatia-2013 "Croatia 2013"
./cesta add-place croatia-2013 zagreb "Zagreb"
```

```
# Add today's photos.
./cesta add-photos croatia-2013 zagreb ../today
```

```
# Maybe review the work, and then set it off...
./cesta worklist
./cesta process
```

# The worklist

**JSON file containing the list of resizes and uploads to do**

**Each photo added gets an entry for full size, a large version, and a thumbnail**

# Example worklist

```
[
  {
    "file" : "../today/DSC02864.JPG",
    "output" : "full/croatia-2014-dubrovnik-5.jpg"
  },
  {
    "file" : "../today/DSC02864.JPG",
    "output" : "large/croatia-2014-dubrovnik-5.jpg",
    "max-width" : 700,
    "max-height" : 450
  },
  ...
 ]
```

# The code: plan of attack

Parallel image resizing

Parallel image uploading

Link the two together

Ctrl + C handling, logging…

# Resizing

Farm off the real work to ImageMagick - which is, happily, threadsafe

Wrote Image::Magick::Resize - using Perl 6's NativeCall

# Basic resizing

## Here's how to use the module to resize an image (it handles proportional bits):

```
my $ir = Resize.new(image => 'large.jpg');
$ir.resize('thumb.jpg',
     max-height => 100,
     max-width => 150);
```

# Resize (or just copy) per worklist item

```
sub resize-one($todo) {
    if $todo<max-width> && $todo<max-height> {
        my $ir = Resize.new(image => $todo<file>);
        $ir.resize($todo<output>,
            max-height => $todo<max-height>,
            max-width => $todo<max-width>);
    }
    else {
        copy($todo<file>, $todo<output>);
    }
}
```

# Sequential resizing

## Just loop over the worklist and resize each of the things

```
sub resize-worker(@worklist) {
    for @worklist -> $todo {
        resize-one($todo);
    }
}
```

# An easy way to parallelize

## Here's all we need to change to use multiple cores

```
sub resize-worker(@worklist) {
    await do for @worklist -> $todo {
        start { resize-one($todo); }
    }
}
```

# Taking stock

**We've just used Perl 6 code to call a C library over multiple threads**

**Not a single explicit thread or lock in sight!**

# What is `start`?

`start` schedules code on the thread pool, and returns a `Promise` to represent it

A `Promise` represents some asynchronous piece of work

# What is `await`?

**`await` takes one or more `Promise` objects, and waits for all of them to complete**

```
// Note: do for works like a map
await do for @worklist -> $todo {
    start { resize-one($todo); }
}
```

# Good enough?

## Not quite yet

## Would like to control how many threads work on it

```
my constant PARALLEL_RESIZE = 4;
```

# Building simple throttling

## Keep an "active work" list

```
my @working;
```

# Building simple throttling

## Loop over the worklist

```
my @working;
for @worklist -> $todo {
    ...
}
```

# Building simple throttling

## Push resize Promises…

```
my @working;
for @worklist -> $todo {
    @working.push(start { resize-one($todo) });
    ...
}
```

# Building simple throttling

# …until we hit the limit.

```
my @working;
for @worklist -> $todo {
    @working.push(start { resize-one($todo) });
    next if @working < PARALLEL_RESIZE;
    ...
}
```

# Building simple throttling

## Wait for any to complete…

```
my @working;
for @worklist -> $todo {
    @working.push(start { resize-one($todo) });
    next if @working < PARALLEL_RESIZE;
    await Promise.anyof(@working);
    ...
}
```

# Building simple throttling

# …and filter the completed.

```
my @working;
for @worklist -> $todo {
    @working.push(start { resize-one($todo) });
    next if @working < PARALLEL_RESIZE;
    await Promise.anyof(@working);
    @working .= grep({ !$_ });
}
```

# Building simple throttling

## Or more cutely:

```
my @working;
for @worklist -> $todo {
    @working.push(start { resize-one($todo) });
    next if @working < PARALLEL_RESIZE;
    await Promise.anyof(@working);
    @working .= grep(!*);
}
```

# Building simple throttling

## Wait for last ones to be done.

```
my @working;
for @worklist -> $todo {
    @working.push(start { resize-one($todo) });
    next if @working < PARALLEL_RESIZE;
    await Promise.anyof(@working);
    @working .= grep(!*);
}
await Promise.allof(@working);
```

# Promise combinators

anyof returns a Promise that is kept once one or more of the specified Promises are kept

For allof, all of the specified Promises must be kept

# Promise = 1 async value

**Any time we want to communicate *a single asynchronously produced value or event* safely, we can use `Promises`.**

# A termination Promise

**The need to stop resizing images can be communicated easily using a `Promise`.**

**We can simply poll it now and then to see if it was kept…**

# A termination Promise

```
sub resize-worker(@worklist, $kill) {
    my @working;
    for @worklist -> $todo {
        @working.push(start {
            resize-one($todo, $output);
        });
        next if @working < PARALLEL_RESIZE;
        await Promise.anyof(@working, $kill);
        @working .= grep(!*);
        last if $kill;
    }
    await Promise.allof(@working);
}
```

# The uploading

So far, we've seen `Promises` represent computation, cancellation, and combination.

Turns out we can also use them for asynchronous processes.

# Simple async processes

**Here's the simplest possible thing: spawn a process and await its exit.**

```
my $proc = Proc::Async.new:
    path => 'pscp',
    args => [$file, "$server-path/$file"];
await $proc.start;
```

# Keep process and Promise

## So we can kill it if needed, we'll keep the process and exit Promise together

```
sub start-upload($file) {
    my $proc = Proc::Async.new:
        path => 'pscp',
        args => [$file, "$server-path/$file"];
    return { :$proc, :$file, done => $proc.start };
}
```

# Upload worker

## The upload worker will return the things it successfully uploaded

```
sub upload-worker(@files) {
    my @working;
    my @done;
    ...
    return @done;
}
```

# Upload worker

## We'll go over the files to do…

```
sub upload-worker(@files) {
    my @working;
    my @done;
    for @files -> $file {
        ...
    }
    ...
    return @done;
}
```

# Upload worker

## Inside the loop, we do much as we did with the resize worker

```
for @files -> $file {
    @working.push(start-upload($file));
    next if @working < PARALLEL_UPLOAD;
    await Promise.anyof(@working.map(*.<done>));
    process-completed-uploads(@working, @done);
}
```

# Upload worker

## After the loop, wait for all the uploads to get done

```
sub upload-worker($input) {
    my @working;
    my @done;
    ...
    await Promise.allof(@working.map(*.<done>));
    process-completed-uploads(@working, @done);
    return @done;
}
```

# Upload worker in full

```
sub upload-worker($input) {
    my @working;
    my @done;
    for @files -> $file {
        @working.push(start-upload($file));
        next if @working < PARALLEL_UPLOAD;
        await Promise.anyof(@working.map(*.<done>));
        process-completed-uploads(@working, @done);
    }
    await Promise.allof(@working.map(*.<done>));
    process-completed-uploads(@working, @done);
    return @done;
}
```

# Processing completed uploads

```
sub process-completed-uploads(@working, @done) {
    @working .= grep({
        if .<done> {
            my $file = .<file>;
            if .<done>.status == Kept &&
               .<done>.result.exit == 0 {
                @done.push($file);
            }
            False
        }
        else {
            True
        }
    });
}
```

# Uploads and $kill

# Changes in the loop:

```
for @files -> $file {
    last if $kill;
    @working.push(start-upload($file));
    next if @working < PARALLEL_UPLOAD;
    await Promise.anyof(@working.map(*.<done>),
        $kill);
    process-completed-uploads(@working, @done);
}
```

# Uploads and $kill

# Changes after the loop:

```
await Promise.anyof(
    $kill,
    Promise.allof(@working.map(*.<done>)));
if $kill {
    .<proc>.kill() for @working;
}
process-completed-uploads(@working, @done);
return @done;
```

# Putting the pieces together

We now have a resizing stage and an uploading stage

Next, we need to wire them together in a safe way

# Use a Channel

## Make a Channel, and then pass it to each of them

```
sub process(@worklist) {
    my $kill = Promise.new;
    my $upload = Channel.new;
    start {
        resize-worker(@worklist, $upload, $kill);
    }
    upload-worker($upload, $log, $kill);
}
```

# Send the files to upload

```
sub resize-worker(@input, $output, $kill) {
    my @working;
    for @input -> $todo {
        @working.push(start {
            resize-one($todo, $output);
            $output.send($todo<output>);
        });
        ...
    }
    await Promise.allof(@working);
    $output.close();
}
```

# Receive the files to upload

## Iterate the channel like a list, until the sender closes it

```
sub upload-worker($input, $kill) {
    my @working;
    my @done;
    for $input.list -> $file {
        ...
    }
    ...
}
```

# About Channels

**At their heart, a concurrent queue data structure**

**Ideal for wiring together larger stages of a system; less good for fine-grained things**

# Reporting progress

**Want a thread-safe, loosely coupled mechanism for reporting back progress**

**Really, we have a stream of asynchronous values**

# Introducing Supply

A `Supply` is a little like a `Promise` in that you can push values or events out in an asynchronous fashion. However, many values can be pushed over time.

# Logging via. a Supply

# Create it and pass it

```
sub process(@worklist) {
    my $log = Supply.new;
    my $kill = Promise.new;
    my $upload = Channel.new;
    start {
        resize-worker(@worklist, $upload, $log,
            $kill);
    }
    upload-worker($upload, $log, $kill);
}
```

# Logging via. a Supply

# Simply say each value

```
sub process(@worklist) {
    my $log = Supply.new;
    $log.act(&say);
    my $kill = Promise.new;
    my $upload = Channel.new;
    start {
        resize-worker(@worklist, $upload, $log,
            $kill);
    }
    upload-worker($upload, $log, $kill);
}
```

# Logging via. a Supply

## Then, code that wants to log something just delivers the value using the Supply

```
$log.more("Resized $todo<output>");
```

```
$log.more("Uploaded $file");
```

# We Supply all sorts!

**Anything that provides a sequence of asynchronous values is exposed as a `Supply`**

**Let's consider how we handle `SIGINT` (from Ctrl + C)**

# Supporting termination

## All we need to do, upon SIGINT, is to keep the $kill Promise

```
my $kill = Promise.new;
signal(SIGINT).act({
    $kill.keep(True) unless $kill;
    $log.more('Terminating...');
});
```

# The command line interface

## Just need to write a MAIN!

```
multi MAIN('process') {
    my @worklist :=
        (try from-json slurp "db/worklist.json")  // [];
    whinge("Nothing to do") unless @worklist;

    my %completed-ids = process(@worklist).map(* => True);
    spurt "db/worklist.json", to-json
        @worklist.grep({ !%completed-ids{.<output>} });

    say "Completed";
}
```

# MAIN subroutines

# The rest look similar…

```
multi MAIN('add-journey', $journey-id, $title) {
    ...
}
multi MAIN('add-photos', $journey-id, $place-id,
           $photo-dir) {
    ...
}
multi MAIN('worklist') {
    ...
}
```

# MAIN subroutines

# …and Perl 6 even introspects them to generate usage!

```
$ ./cesta
Usage:
  cesta.p6 add-journey <journey-id> <title>
  cesta.p6 add-place <journey-id> <place-id> <title>
  cesta.p6 add-photos <journey-id> <place-id> <photo-dir>
  cesta.p6 journeys
  cesta.p6 worklist
  cesta.p6 process
```

# We've built something that…

**Does CPU-bound work over multiple threads, juggles multiple processes, passes data along a thread-safe pipeline, handles signals, and supports cancellation!**

# Together with the CLI…

**This entire application weighs in at 176 lines**

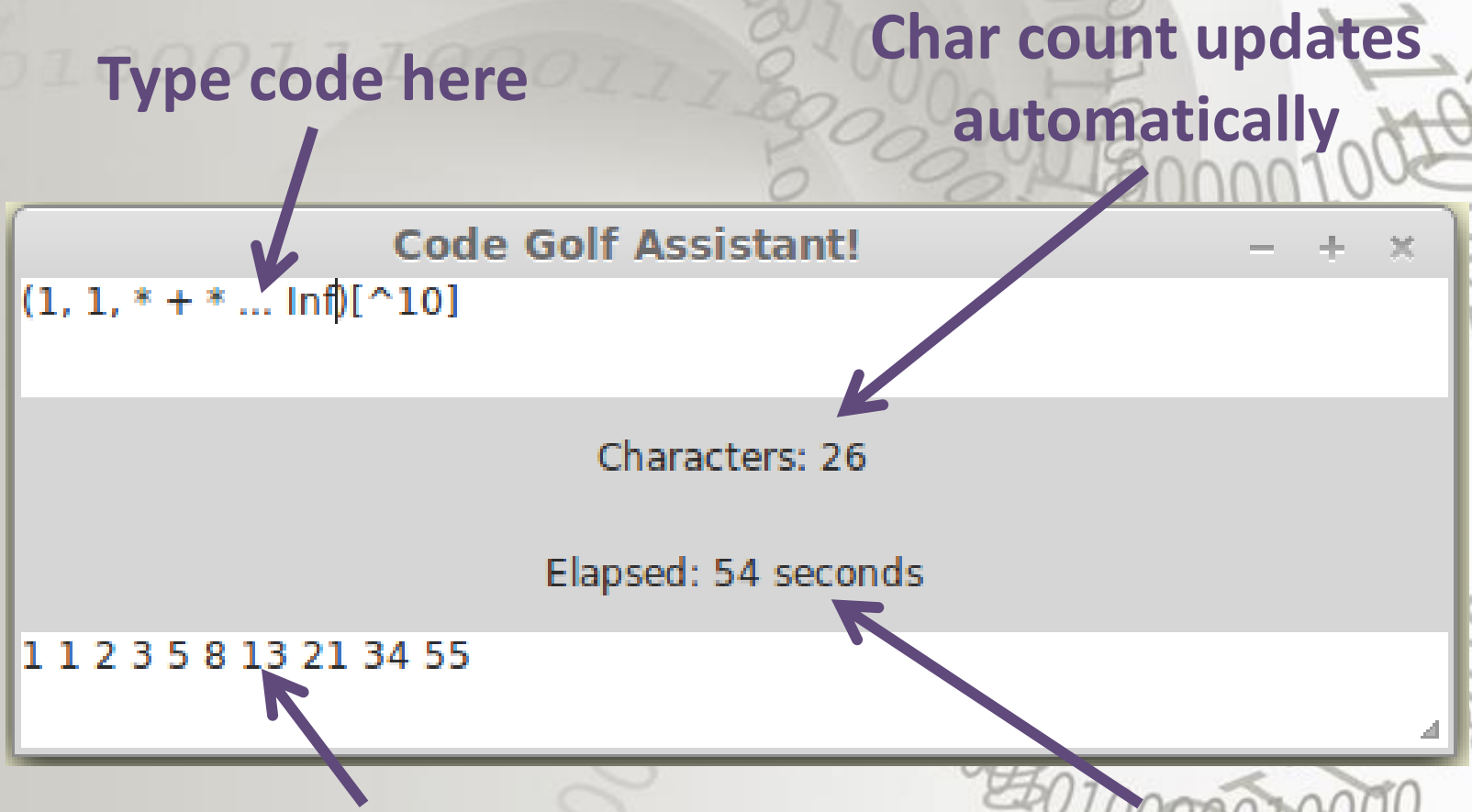
**(Plus a single, pure Perl 6 module to resize, at 65 lines)**

# One more thing: a GUI app

It's also feasible to expose UI events as supplies

This can allow for some quite powerful things to be done with very little code

# Code golf assistant



# Set up the UI

## Just create a few controls

```
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(),
));
```

# Events are supplies

## Can easily tap into the asynchronous event stream

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

# Time is a Supply

**Get a `Supply` that pushes an incrementing value every second, tap it, update UI.**

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

# But wait!

**Timer ticks might not come on the UI thread! So we must tap it on the UI thread:**

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

# Running the code

## The easiest way to do it is:

```
$source.changed.tap({
    $results.text = (try EVAL .text) // $!.message
});
```

## However, this will evaluate on every keystroke *and* lock up the user interface! ☹

# Running the code

**Thankfully, there is a way to wait for the value to have been stable for a time period**

```
$source.changed.stable(1).tap({
    $results.text = (try EVAL .text) // $!.message
});
```

# Running the code

**Then, we can kick it off to run on a background thread**

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

**This is fine, but now different evaluations may race!**

# Running the code

From `start`, we get a `Supply` of `Supply`. The `migrate` method only pays attention to the latest one.

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

# Running the code

## Finally, we punt the result to the UI thread and show it:

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

# And there we have it!

**A UI application, handling UI events, doing time-based updates, running code on a background thread, and showing the results…**

# And there we have it!

**A UI application, handling UI events, doing time-based updates, running code on a background thread, and showing the results…**

***…in 28 lines of code!***

# Status

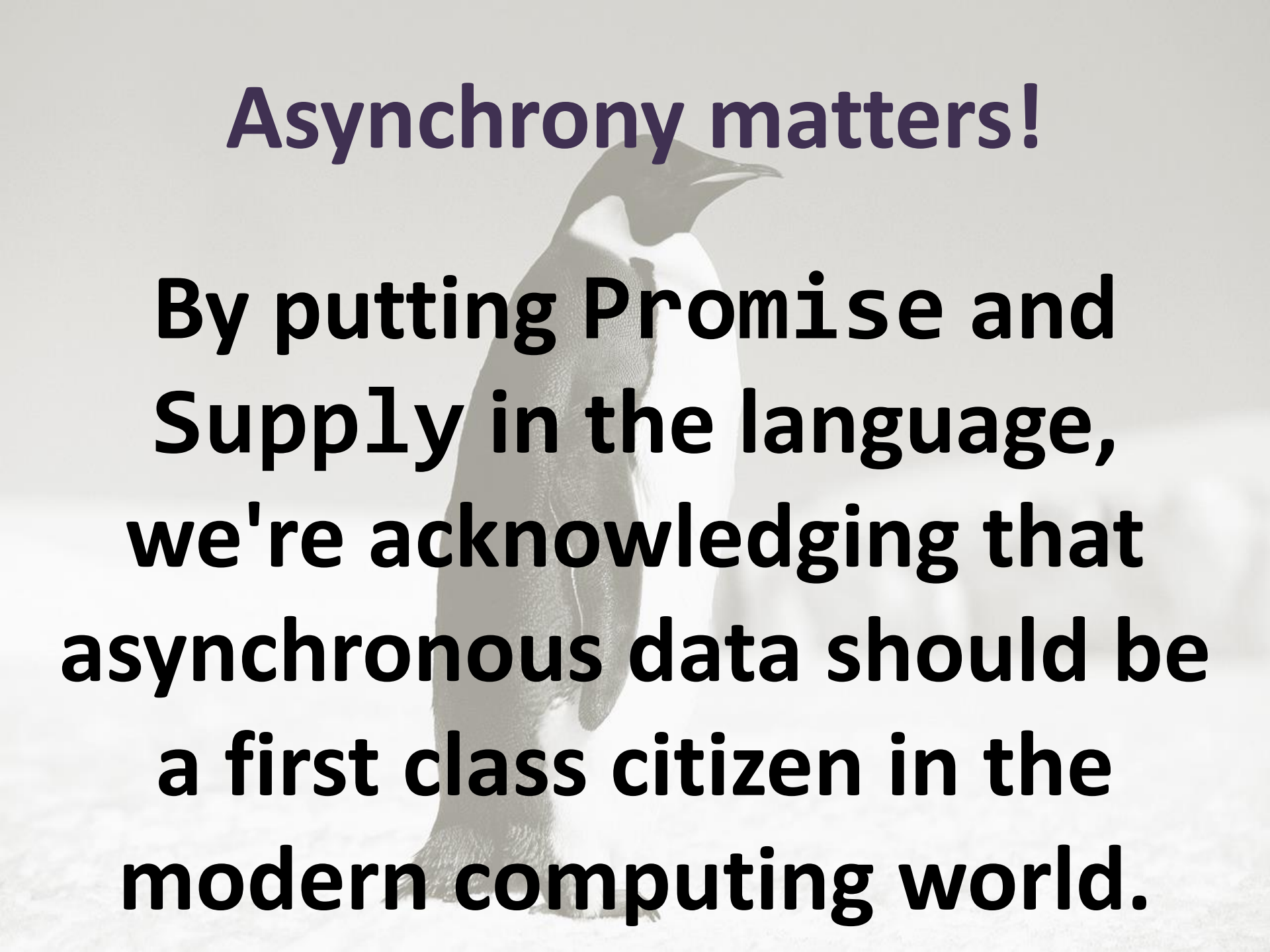
All you've seen today is working code

Pretty solid support on JVM; MoarVM provides the features, and we're polishing

# Composable mechanisms

**When code uses scalars, arrays, etc. we have common data structures, and are able to compose things. We're making it that way for asynchrony too.**

# Asynchrony matters!



**By putting `Promise` and `Supply` in the language, we're acknowledging that asynchronous data should be a first class citizen in the modern computing world.**


Questions?