KYLE SIMPSON @GETIFY

# RETHINKING ASYNC

# Async Patterns

- Parallel vs Async
- Callbacks
- Thunks
- Promises
- Generators / Coroutines
- Event Reactive (observables)
- CSP (channel-oriented concurrency)

# Parallel
# vs
# Async

# Threads

Async Patterns: parallel vs async

# Single Thread

# Concurrency

## Async Patterns: parallel vs async

1
2
3
4

1
2
3

done!

done!

Async Patterns: parallel vs async

1

1

2

2

3 done!

3

4 done!

Async Patterns: parallel vs async

# Async Patterns

# Callbacks

Async Patterns

```
1 setTimeout(function(){
2     console.log("callback!");
3 },1000);
```

# Async Patterns: callbacks

# Callbacks

# ==

# Continuations

Async Patterns: callbacks

(exercise #1: 5min)

```
1 setTimeout(function(){
2     console.log("one");
      setTimeout(function(){
          console.log("two");
          setTimeout(function(){
              console.log("three");
          },1000);
      },1000);
9 },1000);
```

# Async Patterns: “callback hell”

```
1  function one(cb) {
2      console.log("one");
3      setTimeout(cb,1000);
4  }
5  function two(cb) {
6      console.log("two");
7      setTimeout(cb,1000);
8  }
9  function three() {
10     console.log("three");
11 }
12
13 one(function(){
14     two(three);
15 });
```

# Async Patterns: “callback hell”

# Two Problems

Async Patterns: “callback hell”

# Inversion of Control

Async Patterns: “callback hell”

```
1 // line 1
2 setTimeout(function(){
3     // line 3
4     // line 4
5 },1000);
6 // line 2
```

# Async Patterns: inversion of control

```
1 trackCheckout(
2       purchaseInfo,
3       function finish() {
4           chargeCreditCard(purchaseInfo);
5           showThankYouPage();
6       }
7 );
```

Async Patterns: inversion of control

```
1  var hasBeenCalled = false;
2
3  trackCheckout(
4      purchaseInfo,
5      function finish() {
6          if (!hasBeenCalled) {
7              hasBeenCalled = true;
8              chargeCreditCard(purchaseInfo);
9              showThankYouPage();
10         }
11     }
12 );
```

Async Patterns: inversion of control

# Trust:

1. not too early
2. not too late
3. not too many times
4. not too few times
5. no lost context
6. no swallowed errors

...

# Not Reasonable

```
start task1:
    do some stuff
    pause

start task2:
    do some other stuff
    pause

resume task1:
    do more stuff
    pause

resume task2:
    finish stuff

resume task1:
    finish stuff
```

Async Patterns: not reasonable

```
start task1:
    do some stuff
    pause

    resume task1:
        do more stuff
        pause

        resume task1:
            finish stuff


start task2:
    do some other stuff
    pause

    resume task2:
```

Async Patterns: not reasonable

# We Write:

```
1 console.log("First half of my program");
2
3 setTimeout(function(){
4
5     console.log("Second half of my program");
6
7 },1000);
```

Async Patterns: not reasonable

# We Think:

```
1 console.log("First half of my program");
2
3 block(1000);
4
5 console.log("Second half of my program");
6
7
```

# JavaScript Thinks:

```
1 console.log("First half of my program");
2
3 // do lots of other stuff
4
5 console.log("Second half of my program");
6
7
```

Async Patterns: not reasonable

# Sync-Looking Async

Synchronous
Sequential
Blocking

Async Patterns: not reasonable

# Non Fixes

```
1  function trySomething(ok,err) {
2      setTimeout(function(){
3          var num = Math.random();
4          if (num > 0.5) ok(num);
5          else err(num);
6      },1000);
7  }
8
9  trySomething(
10     function(num){
11         console.log("Success: " + num);
12     },
13     function(num){
14         console.log("Sorry: " + num);
15     }
16 );
```

# Async Patterns: separate callbacks

```
1  function trySomething(cb) {
2      setTimeout(function(){
3          var num = Math.random();
4          if (num > 0.5) cb(null,num);
5          else cb("Too low!");
6      },1000);
7  }
8
9  trySomething(function(err,num){
10     if (err) {
11         console.log(err);
12     }
13     else {
14         console.log("Number: " + num)
15     }
16 });
```

## Async Patterns: “error-first style”

# Running Example: "The Meaning Of Life"

```
function getData(d,cb) {
    setTimeout(function(){ cb(d); },1000);
}

getData(10,function(num1){
    var x = 1 + num1;
    getData(30,function(num2){
        var y = 1 + num2;
        getData(
            "Meaning of life: " + (x + y),
            function(answer){
                console.log(answer);
                // Meaning of life: 42
            }
        );
    });
});
```

# Async Patterns: nested-callback tasks

Code Me

# Async Patterns

# Thunks

```
1 function add(x,y) {
2     return x + y;
3 }
4
5 var thunk = function() {
6     return add(10,15);
7 };
8
9 thunk(); // 25
```

Async Patterns: thunks

```
1  function addAsync(x,y,cb) {
2      setTimeout(function(){
3          cb( x + y );
4      },1000);
5  }
6
7  var thunk = function(cb) {
8      addAsync(10,15,cb);
9  };
10
11 thunk(function(sum){
12     sum; // 25
13 });
```

Async Patterns: thunks

```
1 function makeThunk(fn) {
2     var args = [].slice.call(arguments,1);
3     return function(cb) {
4         args.push(cb);
5         fn.apply(null,args);
6     };
7 }
```

# Async Patterns: thunks

```
1  function addAsync(x,y,cb) {
2      setTimeout(function(){
3          cb( x + y );
4      },1000);
5  }
6
7  var thunk = makeThunk(addAsync,10,15);
8
9  thunk(function(sum){
10     console.log(sum); // 25
11 });
```

Async Patterns: thunks

```
var get10 = makeThunk(getData,10);
var get30 = makeThunk(getData,30);

get10(function(num1){
    var x = 1 + num1;
    get30(function(num2){
        var y = 1 + num2;

        var getAnswer = makeThunk( getData,
            "Meaning of life: " + (x + y)
        );

        getAnswer(function(answer){
            console.log(answer);
            // Meaning of life: 42
        });
    });
});
```

# Async Patterns: nested-thunk tasks

Code Me

(exercise #2: 5min)

# Promises

## Future Values

## “Completion Events”

```
1  function finish(){
2      chargeCreditCard(purchaseInfo);
3      showThankYouPage();
4  }
5
6  function error(err){
7      logStatsError(err);
8      finish();
9  }
10
11 var listener = trackCheckout(purchaseInfo);
12
13 listener.on("completion",finish);
14 listener.on("error",error);
```

# Async Patterns: "completion event"

```
1  function trackCheckout(info) {
2      return new Promise(
3          function(resolve,reject){
4              // attempt to track the checkout
5
6              // if successful, call resolve()
7              // otherwise, call reject(error)
8          }
9      );
10 }
```

# Async Patterns: (native) promises

```
1  function finish(){
2      chargeCreditCard(purchaseInfo);
3      showThankYouPage();
4  }
5
6  function error(err){
7      logStatsError(err);
8      finish();
9  }
10
11 var promise = trackCheckout(purchaseInfo);
12
13 promise.then(
14     finish,
15     error
16 );
```

# Async Patterns: (native) promises

# Still callbacks?

# Promise Trust:

1. only resolved once
2. either success OR error
3. messages passed/kept
4. exceptions become errors
5. immutable once resolved

# unInversion of Control

Async Patterns: (native) promises

# Flow Control

Async Patterns: (native) promises

```
1 doFirstThing
2    then doSecondThing
3    then doThirdThing
4    then complete
5 or error
```

Async Patterns: promise flow control

# Chaining Promises

```
1  doFirstThing()
2  .then(function(){
3      return doSecondThing();
4  })
5  .then(function(){
6      return doThirdThing();
7  })
8  .then(
9      complete,
10     error
11 );
```

Async Patterns: promise flow control

```
1  function delay(num) {
2      return new Promise(function(resolve,reject){
3          setTimeout(resolve,num);
4      });
5  }
6
7  delay(100)
8  .then(function(){
9      return delay(50);
10 })
11 .then(function(){
12     return delay(200);
13 })
14 .then(function(){
15     console.log("all done!");
16 });
```

# Async Patterns: promise flow control

Code Me

```
function getData(d) {
    return new Promise(function(resolve,reject){
        setTimeout(function(){ resolve(d); },1000);
    });
}

var x;

getData(10)
.then(function(num1){
    x = 1 + num1;
    return getData(30);
})
.then(function(num2){
    var y = 1 + num2;
    return getData("Meaning of life: " + (x + y));
})
.then(function(answer){
    console.log(answer);
    // Meaning of life: 42
});
```

# Async Patterns: promise flow control

Code Me

(exercise #3: 5min)

(exercise #4: 5min)

# Abstractions

Async Patterns: promises

```
1  Promise.all([
2      doTask1a(),
3      doTask1b(),
4      doTask1c()
5  ])
6  .then(function(results){
7      return doTask2(
8          Math.max(
9              results[0],
10             results[1],
11             results[2]
12         );
13     );
14 });
```

Async Patterns: promise "gate"

```
1  var p = trySomeAsyncThing();
2
3  Promise.race([
4      p,
5      new Promise(function(_,reject){
6          setTimeout(function(){
7              reject("Timeout!!");
8          },3000);
9      })
10 ])
11 .then(
12     success,
13     error
14 );
```

Async Patterns: promise timeout

# sequence = automatically chained promises

# github.com/getify/asynquence

```
1  ASQ()
2  .then(function(done){
3      setTimeout(done,1000);
4  })
5  .gate(
6      function(done){
7          setTimeout(done,1000);
8      },
9      function(done){
10         setTimeout(done,1000);
11     }
12 )
13 .then(function(done){
14     console.log("2 seconds passed!");
15 });
```

```
1  function getData(d) {
2      return ASQ(function(done){
3          setTimeout(function(){ done(d); },1000);
4      });
5  }
6
7  ASQ()
8  .waterfall(
9      function(done){ getData(10).pipe(done); },
10     function(done){ getData(30).pipe(done); }
11 )
12 .seq(function(num1,num2){
13     var x = 1 + num1;
14     var y = 1 + num2;
15     return getData("Meaning of life: " + (x + y));
16 })
17 .val(function(answer){
18     console.log(answer);
19     // Meaning of life: 42
20 });
```

# Async Patterns: sequence tasks

Code Me

(exercise #5: 5min)

(exercise #6: 5min)

davidwalsh.name/asynquence-part-1

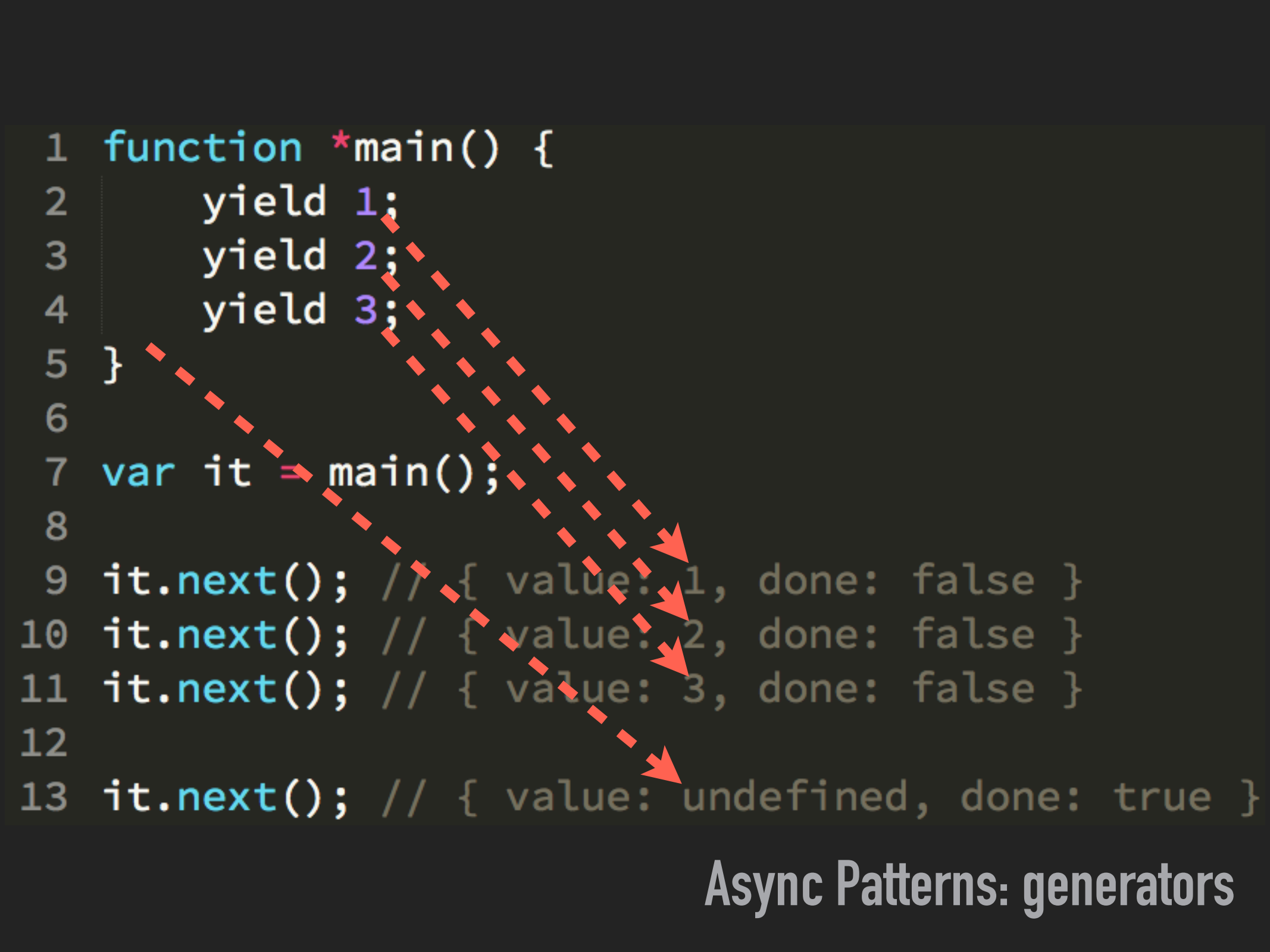
Async Patterns: learn more

# Async Patterns

# Generators (yield)

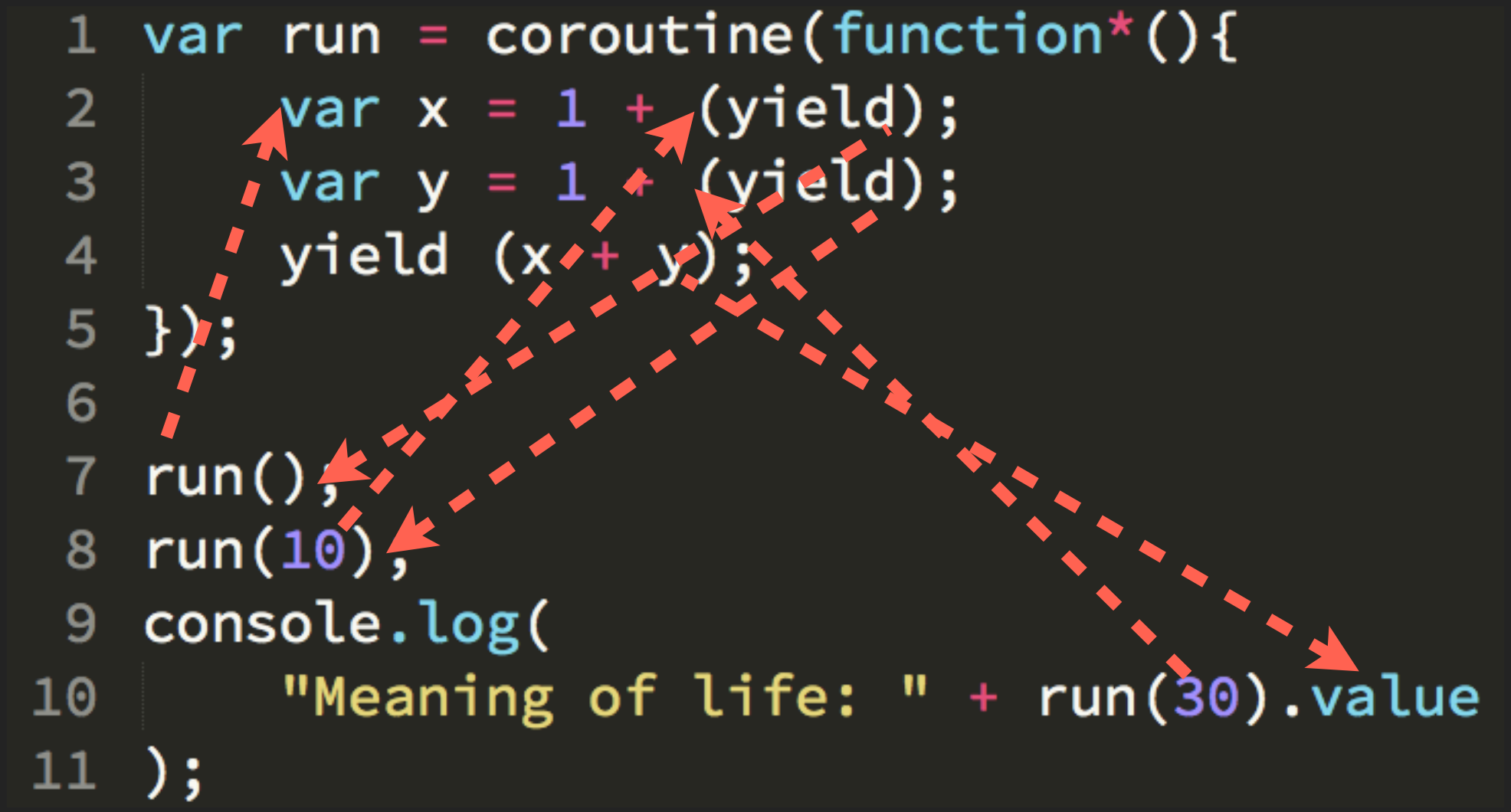
```
1 function* gen() {
2     console.log("Hello");
3     yield;
4     console.log("World");
5 }
6
7 var it = gen();
8 it.next(); // Hello
9 it.next(); // World
```

# Async Patterns: generators


function *main() {
yield 1;
yield 2;
yield 3;
}
var it = main();
it.next(); // { value: 1, done: false }
it.next(); // { value: 2, done: false }
it.next(); // { value: 3, done: false }
it.next(); // { value: undefined, done: true }
Async Patterns: generators

```
1 function coroutine(g) {
2     var it = g();
3     return function(){
4         return it.next.apply(it,arguments);
5     };
6 }
```

# Async Patterns: generator coroutines




# Async Patterns: generator messages

```
1  function getData(d) {
2      setTimeout(function(){ run(d); },1000);
3  }
4
5  var run = coroutine(function*(){
6      var x = 1 + (yield getData(10));
7      var y = 1 + (yield getData(30));
8      var answer = (yield getData(
9          "Meaning of life: " + (x + y)
10     ));
11     console.log(answer);
12     // Meaning of life: 42
13 });
14
15 run();
```

Async Patterns: yield tasks

# generators + promises

# yield promise

```
function getData(d) {
    return ASQ(function(done){
        setTimeout(function(){ done(d); },1000);
    });
}

ASQ()
.runner(function*(){
    var x = 1 + (yield getData(10));
    var y = 1 + (yield getData(30));
    var answer = yield (getData(
        "Meaning of life: " + (x + y)
    ));
    yield answer;
})
.val(function(answer){
    console.log(answer);
    // Meaning of life: 42
});
```

# Async Patterns: generator+sequence tasks

Code Me

(exercise #7: 10min)

# Quiz

1. What is “callback hell”? Why do callbacks suffer from “inversion of control” and "unreasonability"?
2. What is a Promise? How does it solve inversion of control issues?
3. How do you pause a generator? How do you resume it?
4. How do we combine generators and promises for flow control?

davidwalsh.name/es6-generators

Async Patterns: learn more

# Async Patterns

# Concurrency:
# Events (+ Promises)?

```
1  var p1 = new Promise(function(resolve,reject){
2      $("#btn").click(function(evt){
3          var className = evt.target.className;
4          if (/foobar/.test(className)) {
5              resolve(className);
6          }
7          else {
8              reject();
9          }
10     });
11 });
12
13 p1.then(function(className){
14     console.log(className);
15 });
```

Async Patterns: events + promises

```
1  $("#btn").click(function(evt){
2      var className = evt.target.className;
3      var p1 = new Promise(function(resolve,reject){
4          if (/foobar/.test(className)) {
5              resolve(className);
6          }
7          else {
8              reject();
9          }
10     });
11     p1.then(function(className){
12         console.log(className);
13     });
14 });
```

Async Patterns: events + promises

```
1  $("#btn").click(function(evt){
2
3      [evt]
4       .map(function mapper(evt) {
5           return evt.target.className;
6       })
7       .filter(function filterer(className) {
8           return /foobar/.test(className);
9       })
10      .forEach(function(className){
11          console.log(className);
12      });
13
14 });
```

Async Patterns: events + lists

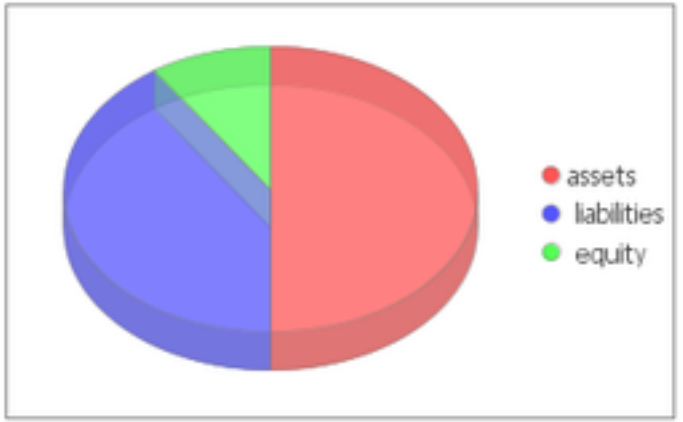
# Observables

Async Patterns

## Ratio Analysis

Values in grey cells are automatically calculated using predefined formula, no alterations required.

| Line Item | Beginning of Year |
|---|---|
| Inventory | $12,500 |
| Total assets | $120,000 |
| Owners' equity | $29,000 |
| Number of common shares | 25,000 |



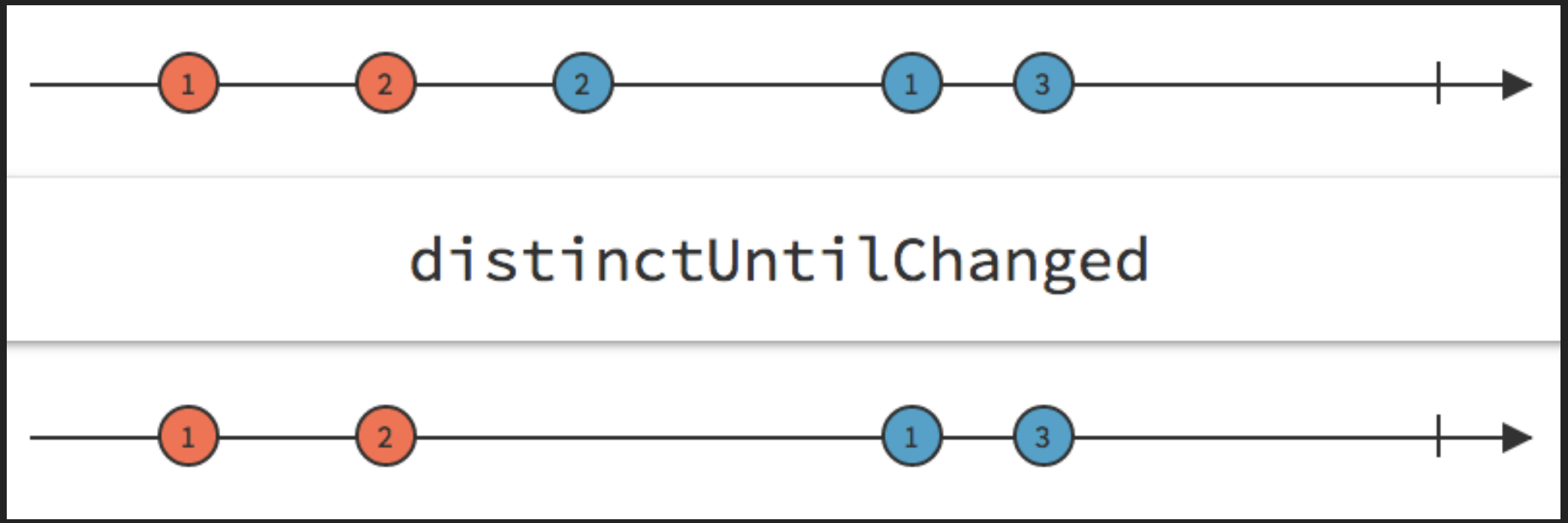

| Line Item | Q1 | Q2 | Q3 | Q4 | Annual |
|---|---|---|---|---|---|
| Current assets | 45,000 | 46,000 | 46,500 | 56,000 | $56,000 |
| Fixed assets | 80,000 | 80,000 | 80,000 | 80,000 | $80,000 |
| Total assets | 125,000 | 126,000 | 126,500 | 136,000 | $136,000 |
| Average total assets | 122,500 | 123,000 | 123,250 | 128,000 | $128,000 |
| Cash and cash equivalents | 15,000 | 18,000 | 16,500 | 14,350 | $14,350 |
| Inventory | 15,000 | 18,000 | 16,500 | 14,350 | $14,350 |
| Average inventory | 13,750 | 15,250 | 14,500 | 13,425 | $13,425 |
| Current liabilities | 23,000 | 25,000 | 22,500 | 25,600 | $25,600 |
| Total liabilities | 125,000 | 125,000 | 125,000 | 110,000 | $110,000 |
| Owners' equity | 28,000 | 30,900 | 32,000 | 26,000 | $26,000 |
| Number of common shares | 25,000 | 25,000 | 25,000 | 25,000 | 25,000 |
| Average number of common shares | 25,000 | 25,000 | 25,000 | 25,000 | 25,000 |
| Average owners' equity | 28,500 | 29,950 | 30,500 | 27,500 | $27,500 |

# Async Patterns: observables

# rxjs.codeplex.com

```
1  var obsv = Rx.Observable.fromEvent(btn,"click");
2
3  obsv
4      .map(function mapper(evt) {
5          return evt.target.className;
6      })
7      .filter(function filterer(className) {
8          return /foobar/.test(className);
9      })
10     .distinctUntilChanged()
11     .subscribe(function(data){
12         var className = data[1];
13         console.log(className);
14     });
```

# RxMarbles.com




Async Patterns: RxJS observables

# asynquence:
# Reactive Sequences

Async Patterns: events

```
function fromEvent(el,eventType) {
    return ASQ.react(function(proceed){
        $(el).bind(eventType,proceed);
    })
}

// aka: observable
var rsq = fromEvent(btn,"click");

rsq
    .val(function(evt){
        return evt.target.className;
    })
    .then(function(done,className){
        if (/foobar/.test(className)) {
            done(className);
        }
    })
    .val(function(className){
        console.log(className);
    });
```

# Async Patterns: reactive sequences

```
1 function fromEvent(el,eventType) {
2     var rsq = ASQ.react.of();
3     $(el).bind(eventType,rsq.push);
4     return rsq;
5 }
```

Async Patterns: reactive sequences

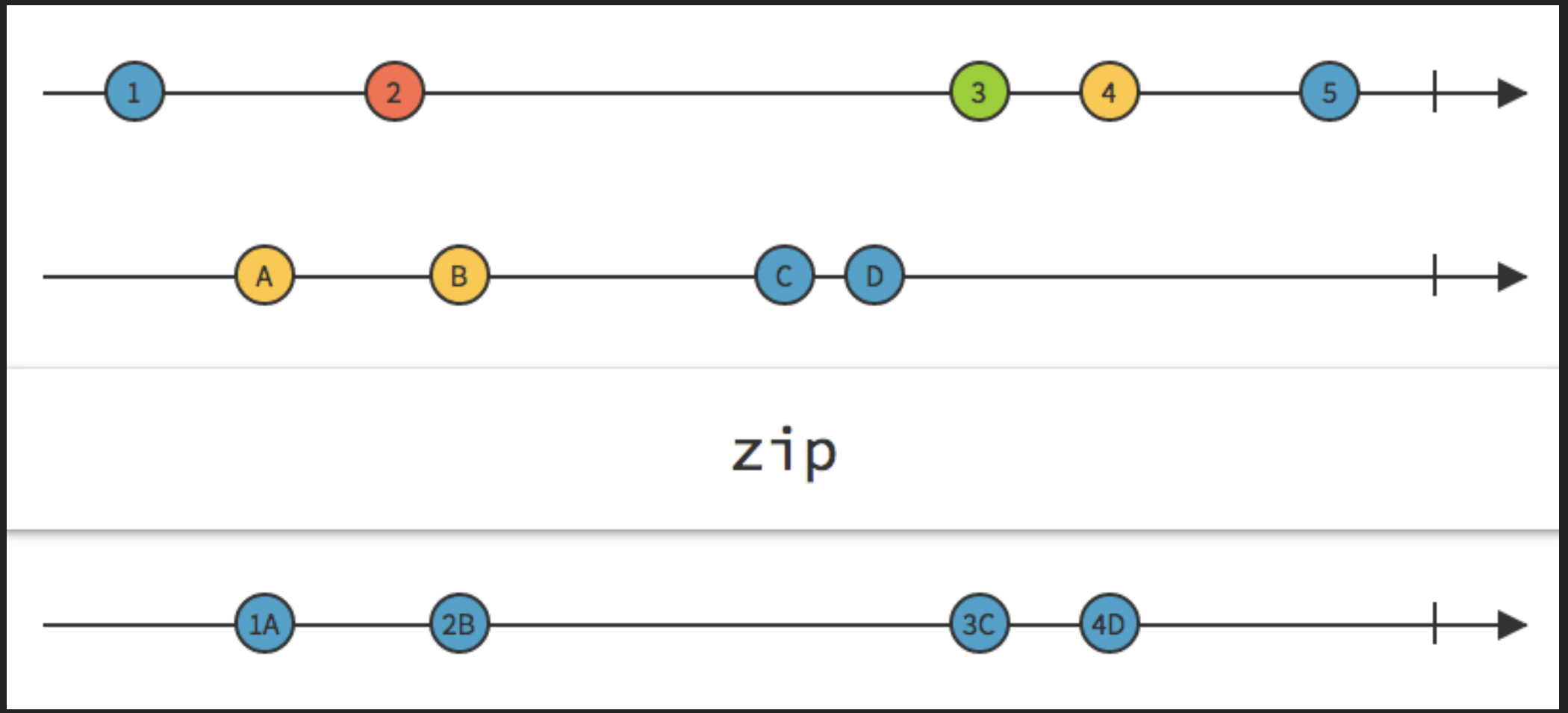
```
1  var rsq1 = ASQ.react.of();
2  var rsq2 = ASQ.react.of(1,2,3);
3  var x = 10;
4
5  setInterval(function(){
6      rsq1.push(x++);
7      rsq2.push(x++);
8  },500);
9
10 rsq1.val(function(v){
11     console.log("1:",v);
12 });
13 // 1: 10  1: 12  1: 14 ...
14
15 rsq2.val(function(v){
16     console.log("2:",v);
17 });
18 // 2: 1  2: 2  2: 3  2: 11  2: 13 ...
```

Async Patterns: reactive sequences

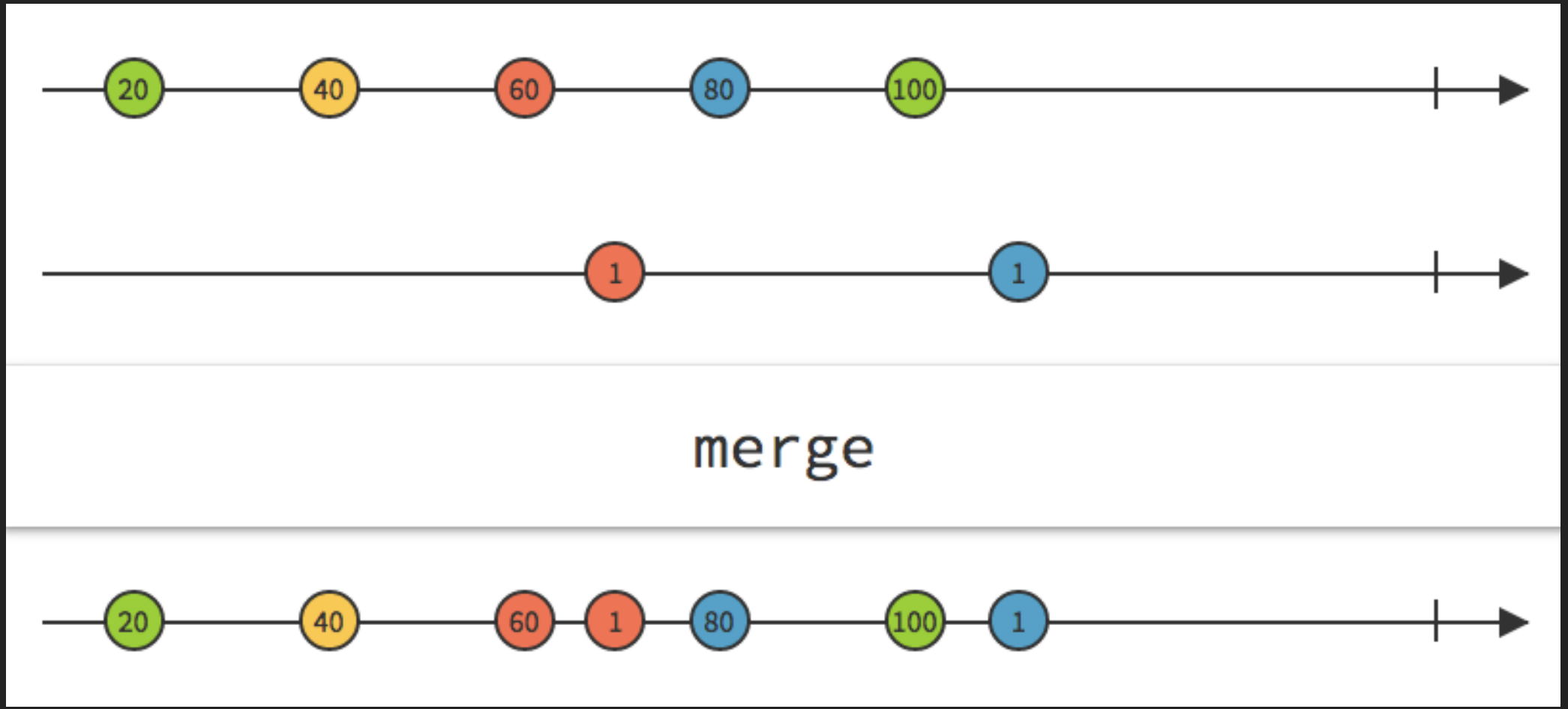
```
1  var rsq1 = fromEvent(btn,"click"),
2      rsq2 = fromEvent(inp,"keypress"),
3
4      rsq3 = ASQ.react.all(rsq1,rsq2),
5      rsq4 = ASQ.react.any(rsq1,rsq2);
6
7  rsq3.val(function(evt1,evt2){
8      // ..
9  });
10
11 rsq4.val(function(evt){
12     // ..
13 });
```

Async Patterns: reactive sequences

# aka "all"



# aka "any"



(exercise #8: 20min)

# Concurrency (+ Channels)

Async Patterns

# CSP: Communicating Sequential Processes

(aka go-style concurrency)

```
1  var ch = chan();
2
3  function *process1() {
4      yield put(ch,"Hello");
5      var msg = yield take(ch);
6      console.log(msg);
7  }
8
9  function *process2() {
10     var greeting = yield take(ch);
11     yield put(ch,greeting + " World");
12     console.log("done!");
13 }
14
15 // Hello World
16 // done!
```

Async Patterns: channel CSP

```
1  csp.go(function*(){
2      while (true) {
3          yield csp.put(ch,Math.random());
4      }
5  });
6
7  csp.go(function*(){
8      while (true) {
9          yield csp.take( csp.timeout(500) );
10         var num = yield csp.take(ch);
11         console.log(num);
12     }
13 });
```

Async Patterns: channel CSP

```
1 csp.go(function*(){
2       while (true) {
3             var msg = yield csp.alts([ch1,ch2,ch3]);
4             console.log(msg);
5       }
6 });
```

# Async Patterns: channel CSP

```
1  csp.go(function* () {
2      var table = csp.chan();
3
4      csp.go(player, ["ping", table]);
5      csp.go(player, ["pong", table]);
6
7      yield csp.put(table, {hits: 0});
8      yield csp.timeout(1000);
9      table.close();
10 });
11
12 function* player(name, table) {
13     while (true) {
14         var ball = yield csp.take(table);
15         if (ball === csp.CLOSED) {
16             console.log(name + ": table's gone");
17             return;
18         }
19         ball.hits += 1;
20         console.log(name + " " + ball.hits);
21         yield csp.timeout(100);
22         yield csp.put(table, ball);
23     }
24 }
```

# Async Patterns: channel CSP

```
1  function fromEvent(el,eventType) {
2      var ch = csp.chan();
3      $(el).bind(eventType,function(evt){
4          csp.putAsync(ch,evt);
5      });
6      return ch;
7  }
8
9  csp.go(function*(){
10     var ch = fromEvent(el,"mousemove");
11     while (true) {
12         var evt = yield csp.take(ch);
13         console.log(
14             evt.clientX + "," + evt.clientY
15         );
16     }
17 });
```

Async Patterns: channel CSP

# asynquence CSP

```
ASQ().runner(
    ASQ.csp.go(function *process1(ch){
        yield ASQ.csp.put(ch,"Hello");
        var msg = yield ASQ.csp.take(ch);
        console.log(msg);
    }),
    ASQ.csp.go(function *process2(ch){
        var greeting = yield ASQ.csp.take(ch);
        yield ASQ.csp.put(ch,greeting + " World");
        console.log("done!");
    })
);
// Hello World
// done!
```

Async Patterns: asynquence CSP

(exercise #9: 20min)

Callbacks / Thunks

Promises

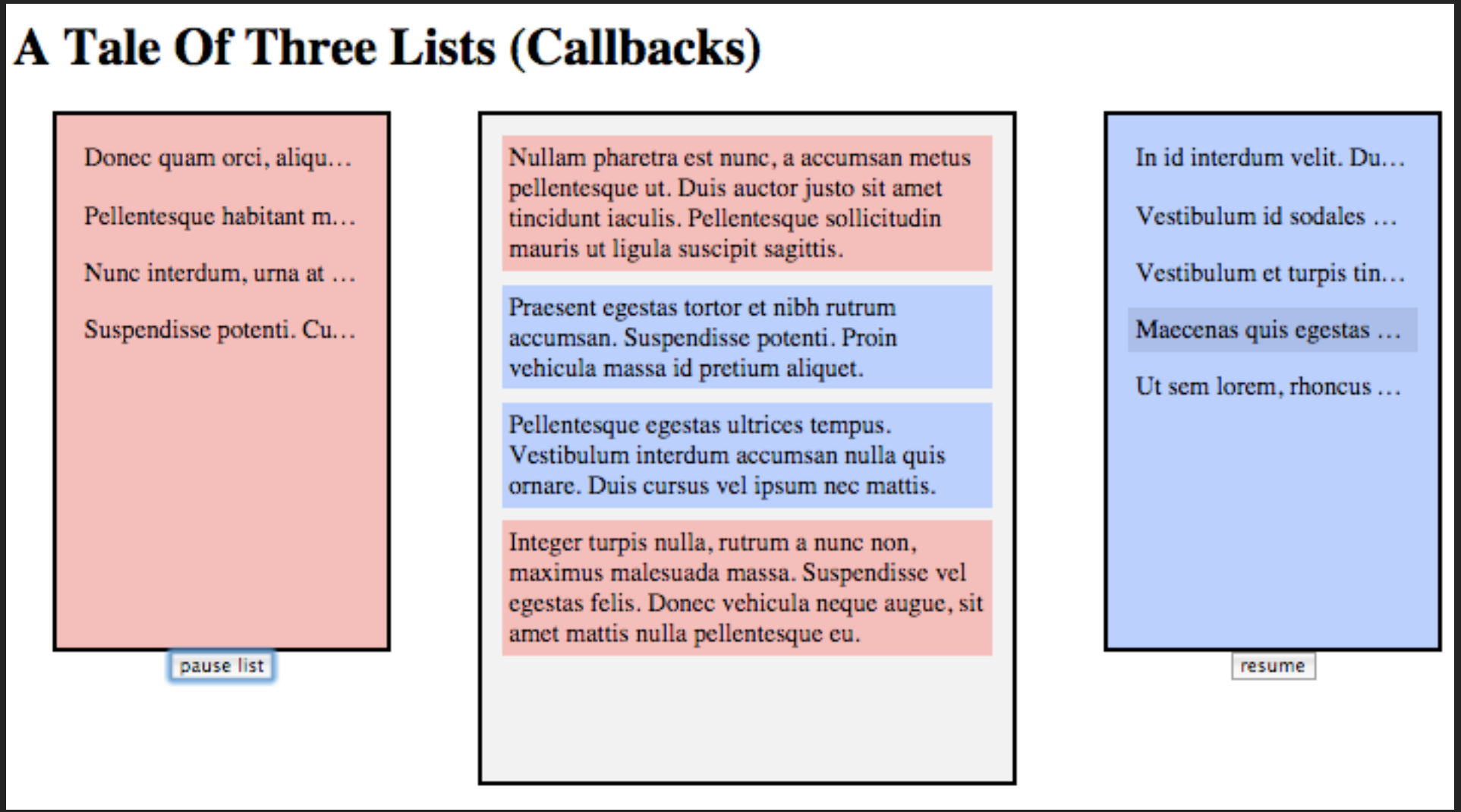
Generators

Observables

CSP go-routines

Async Patterns

# github.com/getify/a-tale-of-three-lists




Async Patterns

Callbacks / Thunks +
Promises +
Generators +
Observables +
CSP go-routines

THANKS!!!!

KYLE SIMPSON @GETIFY

# RETHINKING ASYNC