

KYLE SIMPSON

GETIFY@GMAIL.COM

RETHINKING ASYNC

Async Patterns

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

Parallel vs Async

Async Patterns

Threads

Async Patterns: parallel vs async

Single Thread

Async Patterns: parallel vs async

Concurrency

Async Patterns: parallel vs async

1
2
3
4

1
2
3

done!

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
- ...



Async Patterns: inversion of control

Not Reasonable

Async Patterns: callbacks

```
1 start task1:
2     do some stuff
3     pause
4
5 start task2:
6     do some other stuff
7     pause
8
9 resume task1:
10    do more stuff
11    pause
12
13 resume task2:
14    finish stuff
15
16 resume task1:
17    finish stuff
```

Async Patterns: not **reasonable**

```
1  start task1:
2      do some stuff
3      pause
4
5      resume task1:
6          do more stuff
7          pause
8
9          resume task1:
10             finish stuff
11
12
13 start task2:
14     do some other stuff
15     pause
16
17     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
```

Async Patterns: not **reasonable**

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**

Running Example: "The Meaning Of Life"

Async Patterns: callbacks

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

Async Patterns: nested-callback tasks

Async Patterns

Thanks

Async Patterns

```
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

```
1 var get10 = makeThunk(getData,10);
2 var get30 = makeThunk(getData,30);
3
4 get10(function(num1){
5     var x = 1 + num1;
6     get30(function(num2){
7         var y = 1 + num2;
8
9         var getAnswer = makeThunk( getData,
10             "Meaning of life: " + (x + y)
11         );
12
13         getAnswer(function(answer){
14             console.log(answer);
15             // Meaning of life: 42
16         });
17     });
18 });
```

Async Patterns: nested-thunk tasks

(exercise #2: 5min)

Promises

Future Values

“Completion Events”

Async Patterns

```
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?

Async Patterns: (native) promises

Promise Trust:

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



Async Patterns: (native) promises

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
```

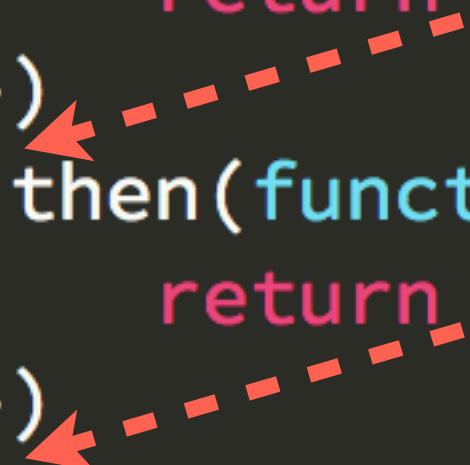
```
1 try {
2     doFirstThing();
3     doSecondThing();
4     doThirdThing();
5     complete();
6 }
7 catch (err) {
8     error();
9 }
```

Async Patterns: promise flow control

Chaining Promises

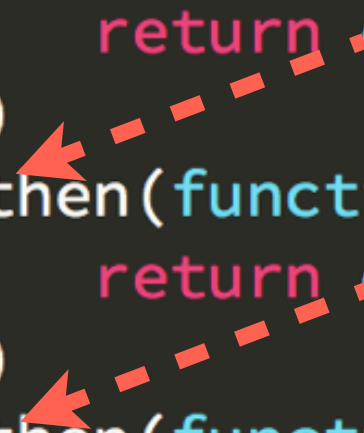
Async Patterns: promise flow control

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

A diagram illustrating promise flow control. Two red dashed arrows originate from the 'return' statements on lines 3 and 6. The first arrow points from 'return doSecondThing();' to the first '.then()' call on line 5. The second arrow points from 'return doThirdThing();' to the second '.then()' call on line 7. This visualizes how the return value of one promise becomes the argument for the next promise in the chain.

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 });
```

A diagram consisting of two dashed red arrows. The first arrow starts at the 'return delay(50);' line (line 9) and points to the first '.then()' block (lines 11-13). The second arrow starts at the 'return delay(200);' line (line 12) and points to the second '.then()' block (lines 14-16). This illustrates how the execution of the next promise in the chain is delayed until the current promise's returned promise is resolved.

Async Patterns: promise flow control

```
1 function getData(d) {
2     return new Promise(function(resolve, reject){
3         setTimeout(function(){ resolve(d); }, 1000);
4     });
5 }
6
7 var x;
8
9 getData(10)
10 .then(function(num1){
11     x = 1 + num1;
12     return getData(30);
13 })
14 .then(function(num2){
15     var y = 1 + num2;
16     return getData("Meaning of life: " + (x + y));
17 })
18 .then(function(answer){
19     console.log(answer);
20     // Meaning of life: 42
21 });
```



The diagram illustrates the flow of a promise chain. Red dashed arrows indicate the sequence of execution: from the `getData(10)` call on line 9 to the first `.then` block on line 10, then to the `return getData(30)` statement on line 12, then to the second `.then` block on line 14, then to the `return getData("Meaning of life: " + (x + y))` statement on line 16, and finally to the third `.then` block on line 18. A red dashed circle highlights the `resolve(d)` call on line 3, indicating the point where the promise is fulfilled.

Async Patterns: promise flow control



(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

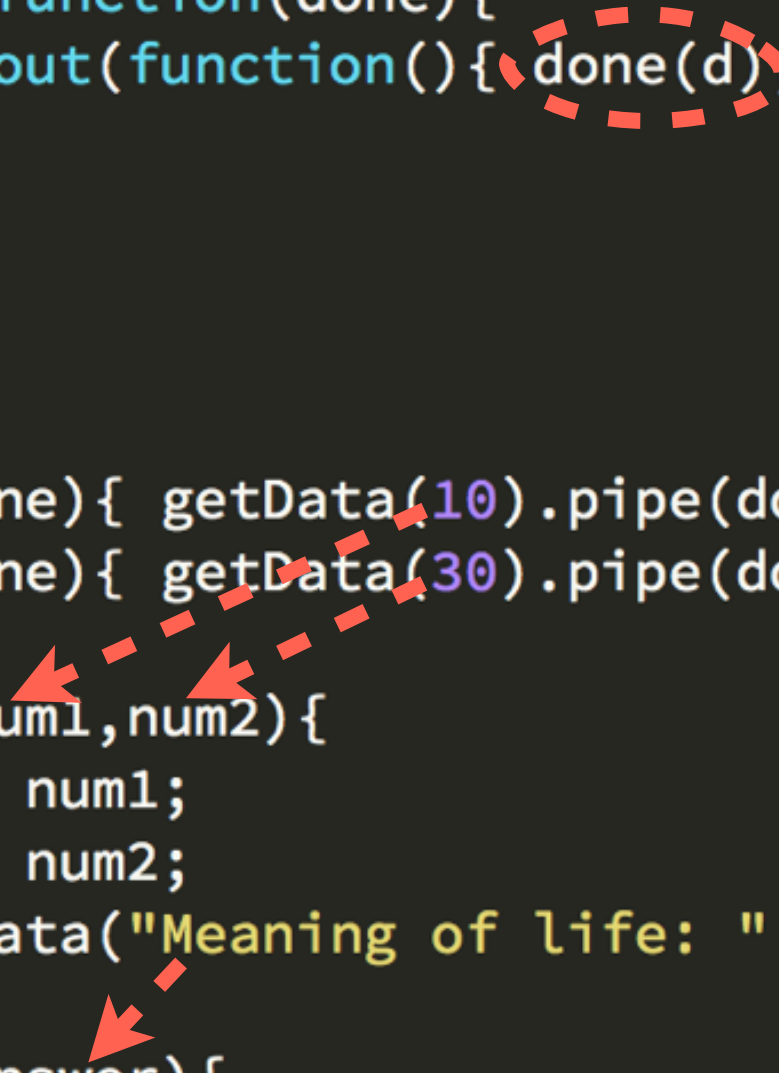
**sequence = automatically
chained promises**

Async Patterns: promises sequence


```
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 });
```

Async Patterns: sequences & gates

```
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

(exercise #5: 5min)

(exercise #6: 5min)

davidwalsh.name/asynquence-part-1

Async Patterns: learn more

Async Patterns

Generators (yield)

Async Patterns

```
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


```
1 function *main() {  
2     yield 1;  
3     yield 2;  
4     yield 3;  
5 }  
6  
7 var it = main();  
8  
9 it.next(); // { value: 1, done: false }  
10 it.next(); // { value: 2, done: false }  
11 it.next(); // { value: 3, done: false }  
12  
13 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

```
1 var run = coroutine(function*(){
2     var x = 1 + (yield);
3     var y = 1 + (yield);
4     yield (x + y);
5 });
6
7 run();
8 run(10);
9 console.log(
10     "Meaning of life: " + run(30).value
11 );
```

The diagram illustrates the flow of generator messages between a coroutine function and its caller. Red dashed arrows indicate the sequence of messages:

- An arrow from the `run()` call on line 7 to the first `yield` statement on line 4.
- An arrow from the `run(10)` call on line 8 to the second `yield` statement on line 3.
- An arrow from the `run(30)` call on line 10 to the final `yield` statement on line 4.

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

Async Patterns: async generators

yield promise

Async Patterns: async generators

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

Async Patterns: generator+sequence tasks

```
1 function getData(d) {
2     return new Promise(function(resolve){
3         setTimeout(function(){ resolve(d); }, 1000);
4     });
5 }
6
7 async function meaningOfLife() {
8     var x = 1 + (await getData(10));
9     var y = 1 + (await getData(30));
10    var answer = await (getData(
11        "Meaning of life: " + (x + y)
12    ));
13    return answer;
14 }
15
16 meaningOfLife()
17 .then(function(answer){
18     console.log(answer);
19     // Meaning of life: 42
20 });
```

Async Patterns: async functions

(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

fasy

github.com/getify/fasy

Async Patterns: async iterations

```
1 ASQ().runner(function *loadFiles(){
2     // request all files concurrently
3     var filePrs = yield FA.concurrent.map(
4         getFile,
5         ["file1","file2","file3"]
6     );
7
8     // output file contents serially
9     yield FA.serial.forEach(function *each(pr){
10         output(yield pr );
11     });
12
13     output("Complete!");
14 });
```

Async Patterns: async iterations

```
1  async function loadFiles() {  
2      // request all files concurrently  
3      var filePrs = await FA.concurrent.map(  
4          getFile,  
5          ["file1","file2","file3"]  
6      );  
7  
8      // output file contents serially  
9      await FA.serial.forEach(async function each(pr) {  
10         output(await pr );  
11     });  
12  
13     output("Complete!");  
14 }  
15  
16 loadFiles();
```

Async Patterns: async iterations

Async Patterns

Concurrency: Events (+ Promises)?

Async Patterns


```
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 });
```

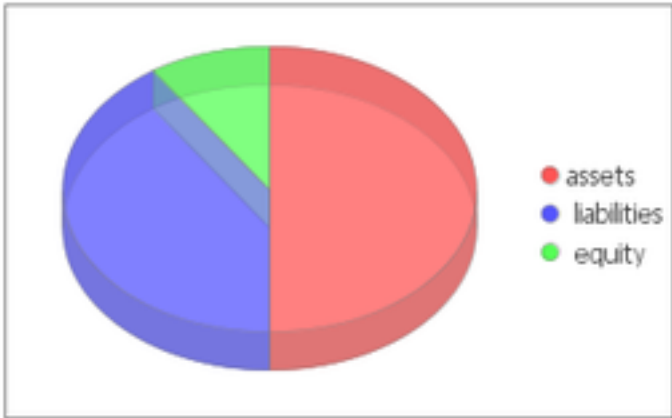
Async Patterns: events + promises

```
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

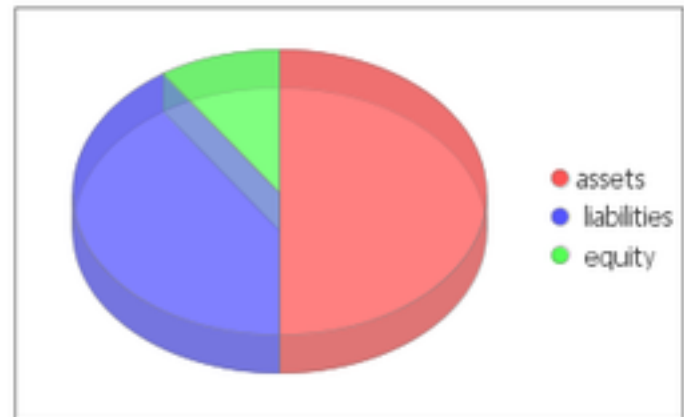
	A	B	C	D	E	F	G	H	I	J
1		Ratio Analysis								
2										
3										
4		Values in grey cells are automatically calculated using predefined formula, no alterations required.								
5										
6		Line Item		Beginning of Year						
7		Inventory		\$12,500						
8		Total assets		\$120,000						
9		Owners' equity		\$29,000						
10		Number of common shares		25,000						
11										
12		Line Item		Q1	Q2	Q3	Q4	Annual		
13		Current assets		45,000	46,000	46,500	56,000	\$56,000		
14		Fixed assets		80,000	80,000	80,000	80,000	\$80,000		
15		Total assets		125,000	126,000	126,500	136,000	\$136,000		
16		Average total assets		122,500	123,000	123,250	128,000	\$128,000		
17		Cash and cash equivalents		15,000	18,000	16,500	14,350	\$14,350		
18		Inventory		15,000	18,000	16,500	14,350	\$14,350		
19		Average inventory		13,750	15,250	14,500	13,425	\$13,425		
20		Current liabilities		23,000	25,000	22,500	25,600	\$25,600		
21		Total liabilities		125,000	125,000	125,000	110,000	\$110,000		
22		Owners' equity		28,000	30,900	32,000	26,000	\$26,000		
23		Number of common shares		25,000	25,000	25,000	25,000	25,000		
24		Average number of common shares		25,000	25,000	25,000	25,000	25,000		
25		Average owners' equity		28,500	29,950	30,500	27,500	\$27,500		



assets

liabilities

equity



Async Patterns: observables

```
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



`distinctUntilChanged`



Async Patterns: RxJS observables

asynquence: **Reactive Sequences**

Async Patterns: events


```
1  function fromEvent(el, eventType) {
2      return ASQ.react(function(proceed) {
3          $(el).bind(eventType, proceed);
4      })
5  }
6
7  // aka: observable
8  var rsq = fromEvent(btn, "click");
9
10 rsq
11   .val(function(evt) {
12       return evt.target.className;
13   })
14   .then(function(done, className) {
15       if (/foobar/.test(className)) {
16           done(className);
17       }
18   })
19   .val(function(className) {
20       console.log(className);
21   });
```

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

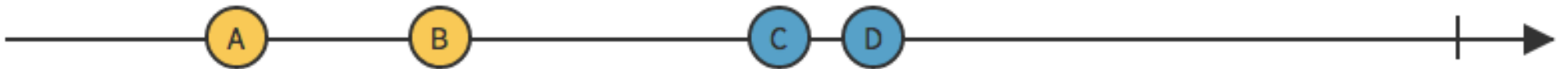
```
1  var rsq1 = ASQ.react.of();
2  var rsq2 = ASQ.react.of(1,2,3);
3  var x = 10;
4
5  setInterval(function(){
6      rsq1.push(x++);
7      rsq2.push(x++);
8  },500);
9
10 rsq1.val(function(v){
11     console.log("1:",v);
12 });
13 // 1: 10  1: 12  1: 14  ...
14
15 rsq2.val(function(v){
16     console.log("2:",v);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

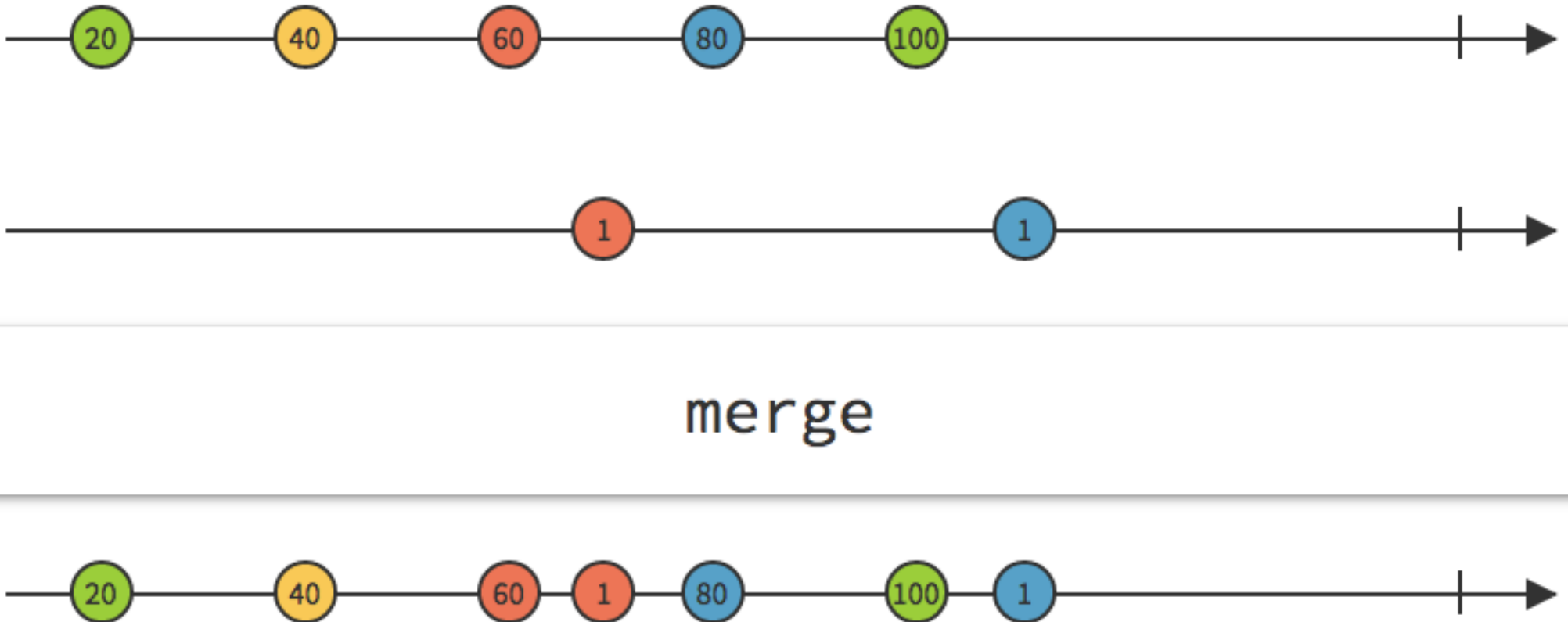


zip



Async Patterns: reactive sequences

aka "any"



Async Patterns: reactive sequences

(exercise #8: 20min)

Concurrency (+ Channels)

Async Patterns

CSP: Communicating Sequential Processes

(aka **go**-style concurrency)

Async Patterns: 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

Async Patterns: asynquence

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

Async Patterns: asynquence CSP

(exercise #9: 20min)

Callbacks / Thunks

Promises

Generators

Observables

CSP go-routines

Async Patterns

A Tale Of Three Lists (Callbacks)

Donec quam orci, aliqu...

Pellentesque habitant m...

Nunc interdum, urna at ...

Suspendisse potenti. Cu...

pause list

Nullam pharetra est nunc, a accumsan metus
pellentesque ut. Duis auctor justo sit amet
tincidunt iaculis. Pellentesque sollicitudin
mauris ut ligula suscipit sagittis.

Praesent egestas tortor et nibh rutrum
accumsan. Suspendisse potenti. Proin
vehicula massa id pretium aliquet.

Pellentesque egestas ultrices tempus.
Vestibulum interdum accumsan nulla quis
ornare. Duis cursus vel ipsum nec mattis.

Integer turpis nulla, rutrum a nunc non,
maximus malesuada massa. Suspendisse vel
egestas felis. Donec vehicula neque augue, sit
amet mattis nulla pellentesque eu.

In id interdum velit. Du...

Vestibulum id sodales ...

Vestibulum et turpis tin...

Maecenas quis egestas ...

Ut sem lorem, rhoncus ...

resume

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

Async Patterns

THANKS!!!!

KYLE SIMPSON GETIFY@GMAIL.COM

RETHINKING
ASYNC