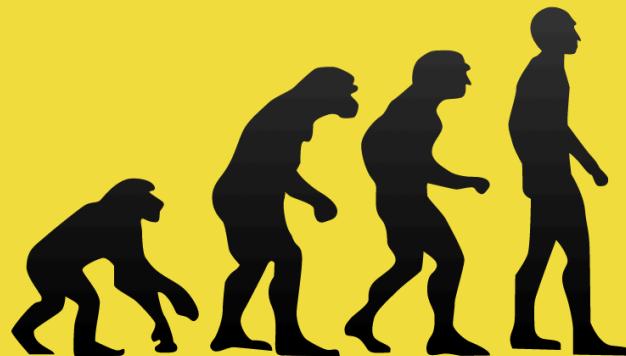


# ES7



## The Evolution of JavaScript

# Who is Jafar Husain?

- Architect of Falcor, Netflix's OSS data platform
- Netflix representative TC-39
- Formerly worked for MS and GE

**NETFLIX**

ES6 spec will be finalized in



Isn't it *too early* to talk about ES7?

No.

ES7 features are *already* starting  
to roll into browsers.



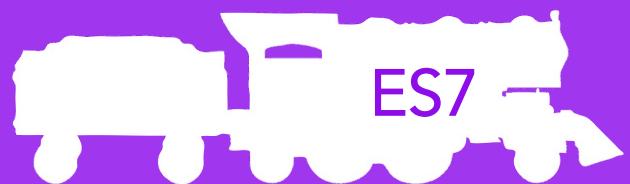
ES5

ES6

ES7

# ES6

2 BIG !



#1

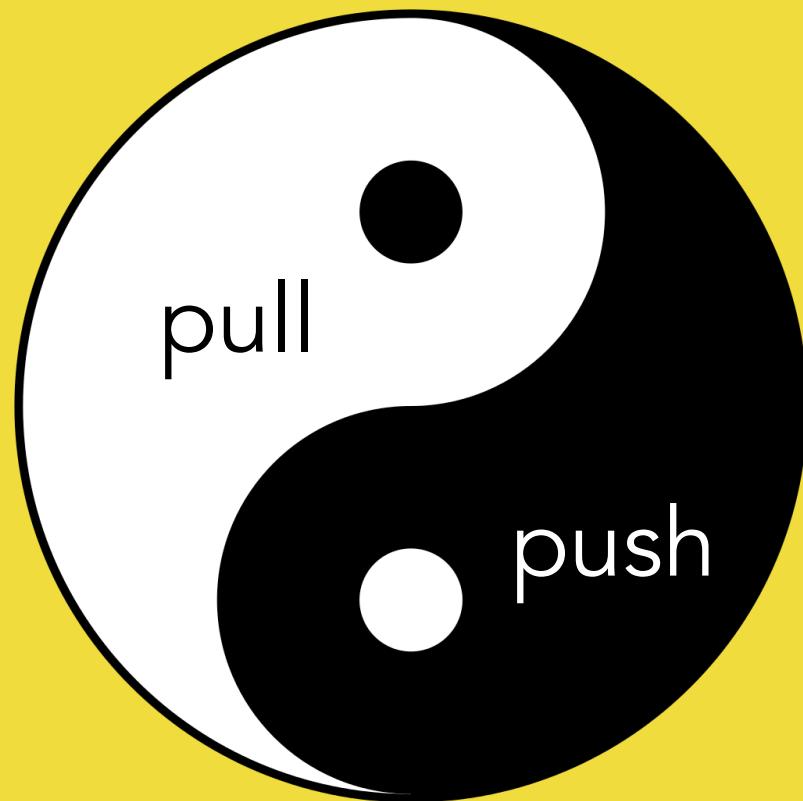
# Easier Model/View Synchronization

Model

View

#2

# Easier Async Programming



# Arrow Functions

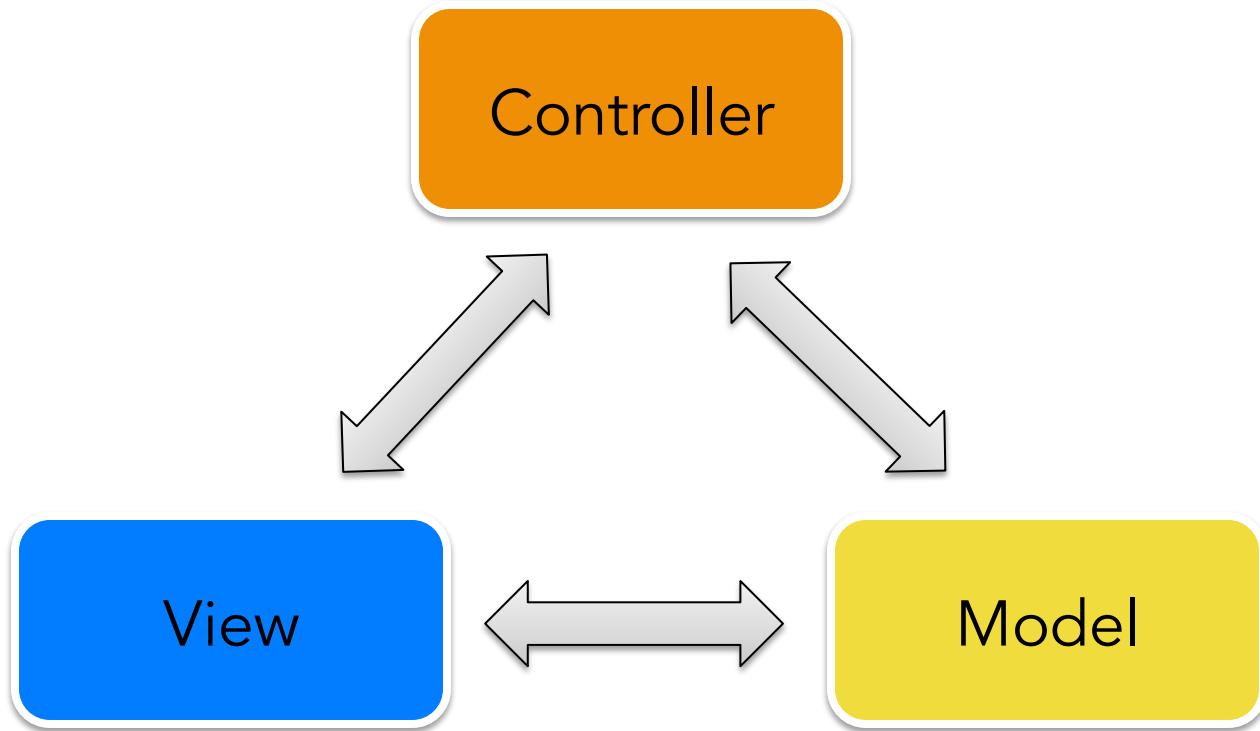
=>



# Arrow Functions

```
function(x) { return x + 1; }  
function(x, y) => return x + y; }
```

ES5  
6



*ember*



Why not natively support  
change notification?



# Object.observe



# Object.observe

```
var person = {};
```

```
var changesHandler = changes => console.log(changes);  
Object.observe(person, changesHandler);
```



# Object.observe

```
person = {  
    name: "Dana MilMæller",  
}; employer: "Tommy's Box Store"
```



```
[  
  {  
    "type": "dataFigure",  
    "name": "employer",  
    }  "oldValue: "DemayMslBex"Store"  
]  }  
]
```

# Object.unobserve

```
Object.unobserve(person, changesHandler);
```



Dictionary   Thesaurus   Medical   Encyclo.

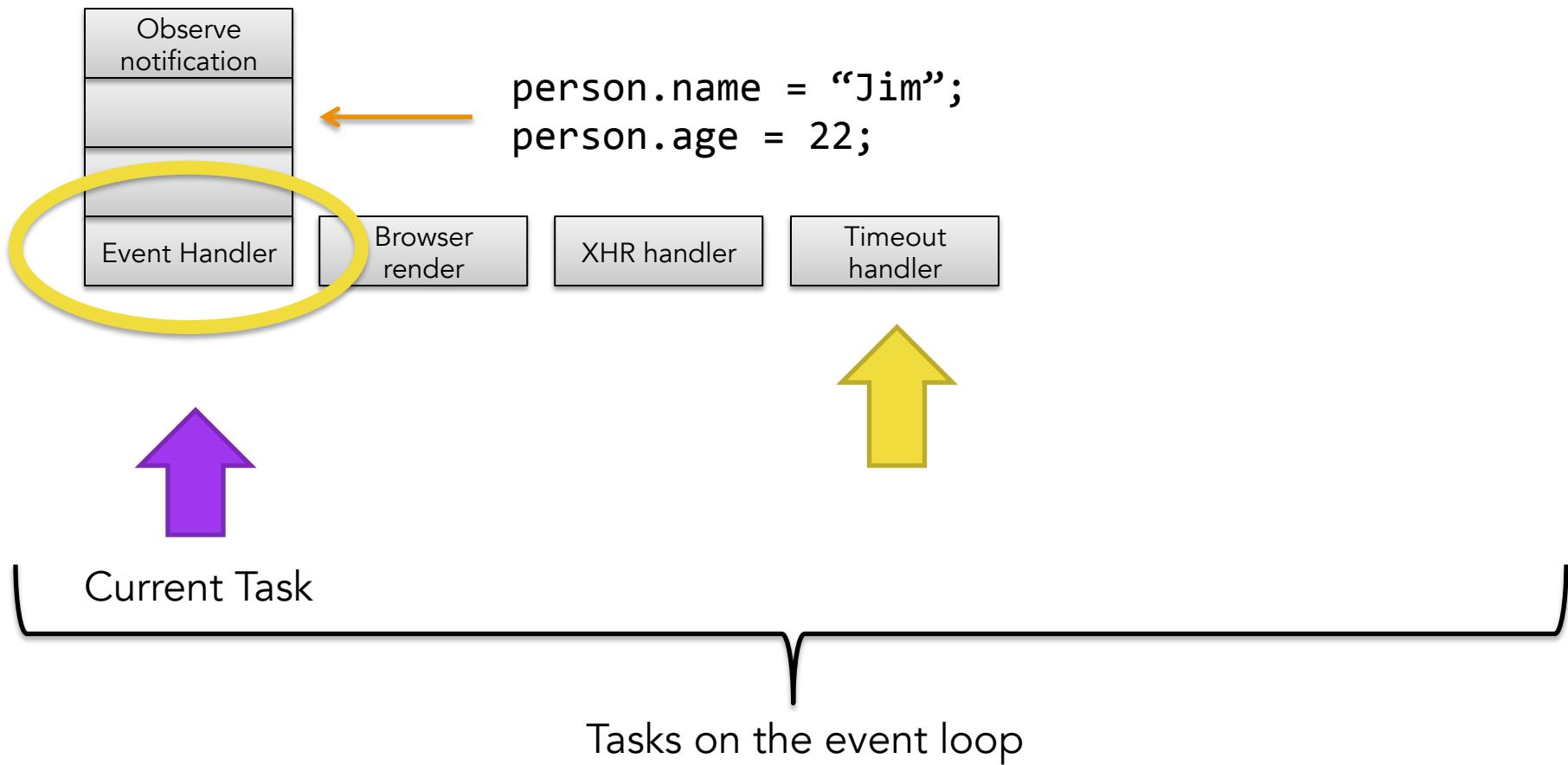
unobserve

A red square icon containing a white magnifying glass symbol, used for search functions.

unobserve

The word you've entered isn't in the dictionary.

# The Event Loop Queue



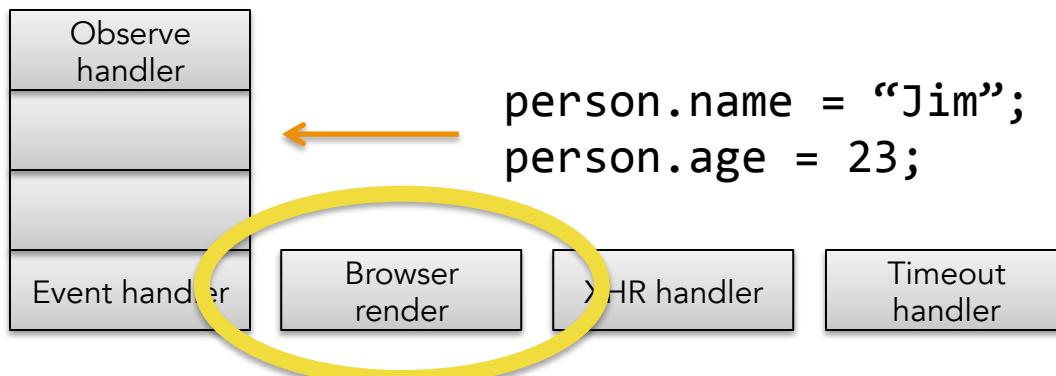


How to avoid the  
**Notification Storm?**

# Microtasks



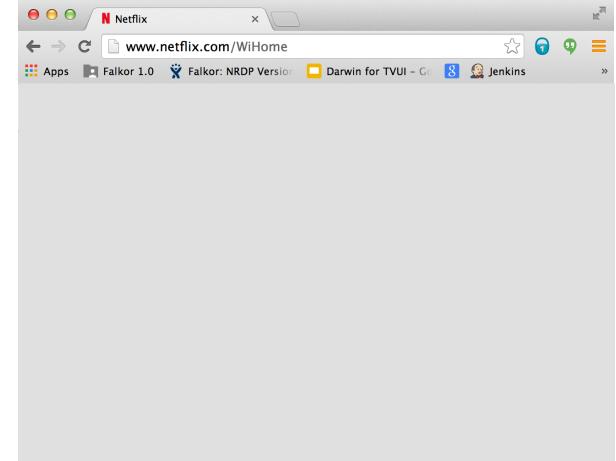
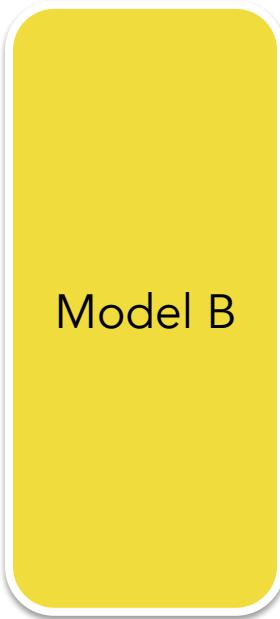
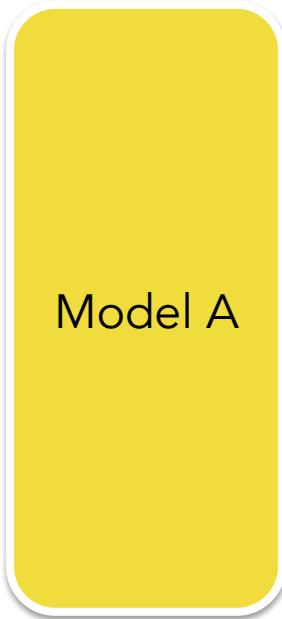
# Event Loop Queue + Microtasks



Current Task

Tasks on the event loop

# Object.observe on Microtasks



# Try Object.observe

A screenshot of a Mac OS X desktop showing a Google Chrome window. The title bar says "chrome://flags". The address bar also shows "chrome://flags". The main content area displays the "Experimental JavaScript" flag, which is currently enabled. Below it is the "Experimental Web Platform Features" flag, which is also enabled. A message at the bottom states that changes will take effect upon relaunch. The URL "chrome://flags/#" is visible in the bottom left corner.

chrome://flags

chrome://flags

Apps Falkor 1.0 Falkor: NRDP Version Darwin for TVUI - Go Jenkins

prefixes-encrypted-media

[Enable](#) [javascri](#) 3 of 4

**Enable Experimental JavaScript** Mac, Windows, Linux, Chrome OS, Android  
Enable web pages to use experimental JavaScript features. [#enable-javascript-harmony](#)

[Enable](#)

Enable experimental Web Platform features. Mac, Windows, Linux, Chrome OS, Android  
Enable experimental Web Platform features that are in development. [#enable-experimental-web-platform-features](#)

[Disable](#)

Your changes will take effect the next time you relaunch Google Chrome.

[Relaunch Now](#)

chrome://flags/#





# ES Feature Maturity Stages

- 0. Strawman
- 1. Proposal
- 2. Draft
- 3. Candidate
- 4. Finished

# Maturity Stage 2: Draft

*“Committee expects the feature to be developed and...included in the standard.”*



How can we make async  
programming easier?

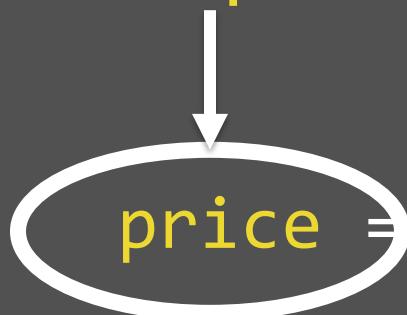


# Blocking is Easy

```
function getStockPrice(name) {  
    var symbol = getStockSymbol(name);  
    var price = getStockPrice(symbol);  
    return price;  
}
```

# Blocking means *Pulling*

Data delivered in  
**return position**



What if we want to wait  
instead of block?



# Waiting means *Pushing*

Data delivered in  
**argument position**

```
getStockPrice("Devine Inc.", price => { ... });
```



push



# Blocking/Pulling

```
function getStockPrice(name) {  
    var symbol = getStockSymbol(name);  
    var price = getStockPrice(symbol);  
    return price;  
}
```

# Waiting/Pushing

```
function getStockPrice(name, cb) {  
    getStockSymbol(name, (error, symbol) => {  
        if (error) {  
            cb(error);  
        }  
        else {  
            getStockPrice(symbol, (error, price) => {  
                if (error) {  
                    cb(error);  
                }  
                else {  
                    cb(price);  
                }  
            })  
        }  
    })  
}
```

# Pulling and Pushing



*are symmetrical*

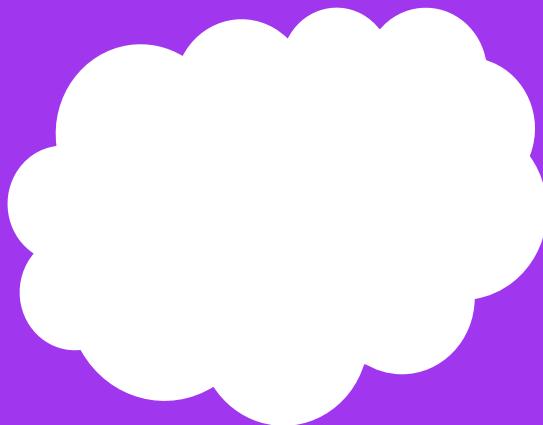
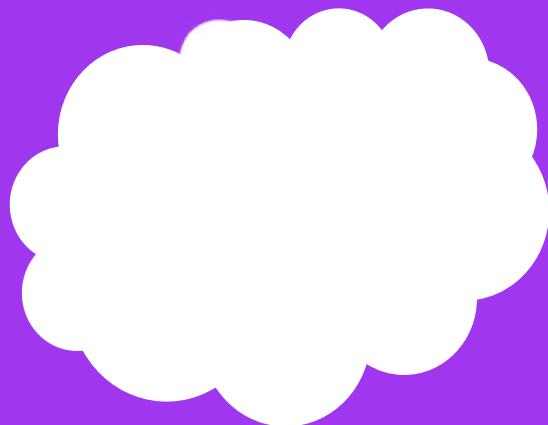
Why must *waiting* be so  
much *harder* than *blocking*?



# Promises



# Promises



```
promise.then(value => {...}, error => {...});
```

sync

```
function getStockPrice(name) {  
    var symbol = getStockSymbol(name);  
    var price = getStockPrice(symbol);  
    return price;  
}
```

async

```
function getStockPrice(name) {  
    return getStockSymbol(name).  
        then(symbol => getStockPrice(symbol));  
}
```

Not bad, but...

...why does waiting have to  
look so different than blocking?



sync

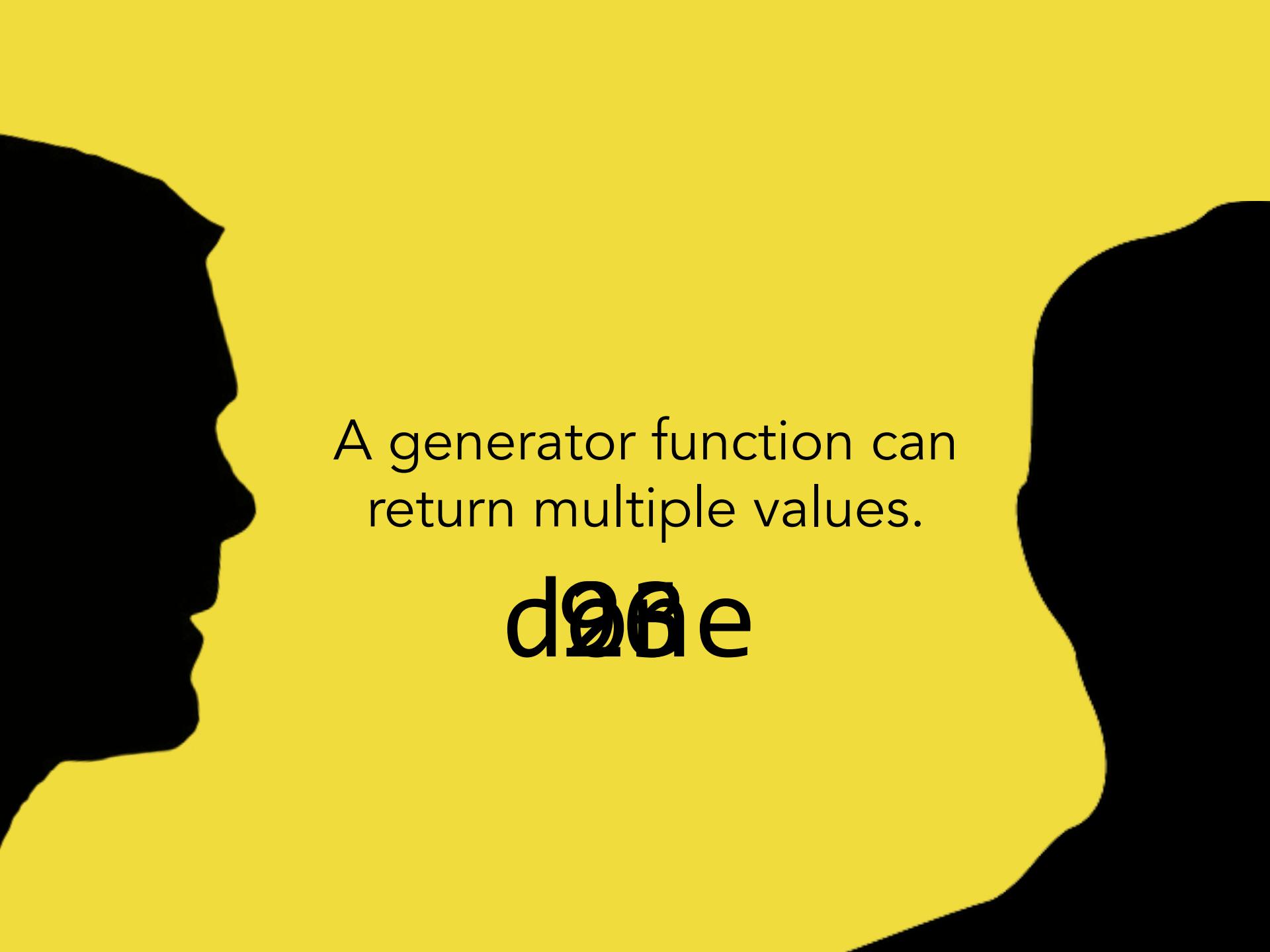
```
function getStockPrice(name) {  
    var symbol = getStockSymbol(name);  
    var price = getStockPrice(symbol);  
    return price;  
}
```

```
function* getStockPrice(name){  
    var symbol = yield getStockSymbol(name);  
    var price = yield getStockPrice(symbol);  
    return price;  
}
```

async

# Generator function





A generator function can  
return multiple values.

doe

# Generator Functions in ES6

```
function* getNumbers() {  
    yield 42;  
    yield 32;  
    return 19;  
}
```

# Retrieving function\* results

```
function* getNumbers() {  
    yield 42;  
    yield 32;  
    return 19;  
}  
  
> var iterator = getNumbers(); █  
  
> console.log(iterator.next()); █  
  
> { value: 42, done: false }  
> █onsole.log(iterator.next()); █  
  
> { value: 32, done: false }  
> █onsole.log(iterator.next()); █  
  
> { value: 19, done: true }  
> █
```

# Iteration

Consumer

Producer

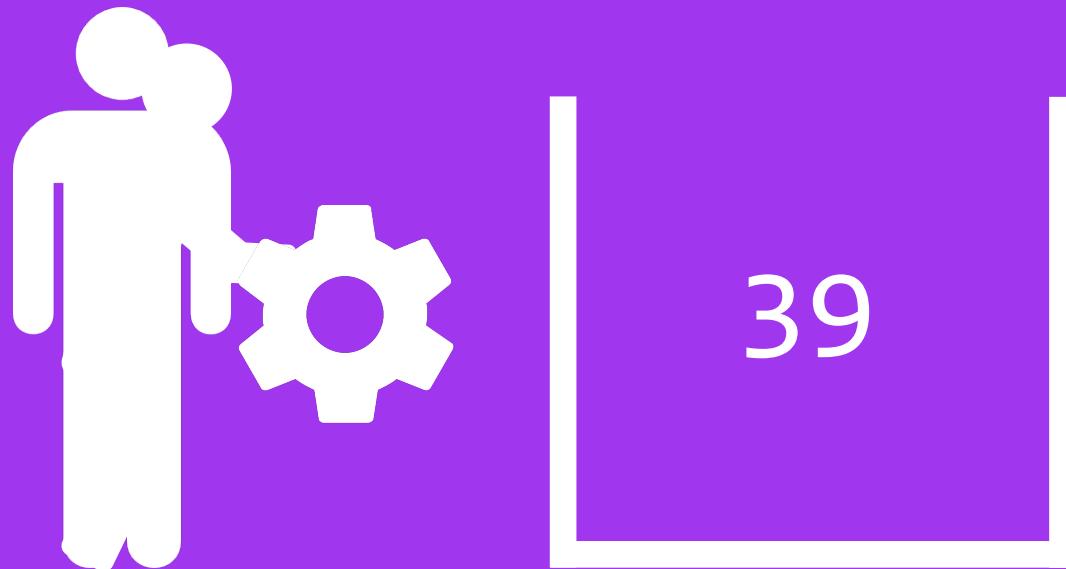


```
var iterator = producer.iterator();
```

# Iteration

Consumer

Producer



```
var result = iterator.next()
```

# Iteration

Consumer

Producer



```
var result = iterator.next()
```

# How Yield Works

```
function*fibonacciSequence() {
    return {
        next: function() {
            let val1 = 0, val2 = 1, swap, state = 0;
            switch(state) {
                case 0:
                    state = 1;
                    yield val1;
                    return {value: val1, done: false};
                case 1:
                    state = 2;
                    yield val2;
                    return {value: val2, done: false};
                while(true) {
                    default:
                        swap = val1 + val2; swap = val1 + val2;
                        val2 = swap; val2 = swap;
                        val1 = val2; val1 = val2;
                        yield swap;
                        return {value: swap, done: false};
                }
            }
        }
    }
}
```

All ES6 collections are Iterable.



# Iterator Pattern

```
> var iterator = [42, 39][@@iterator]();  
> console.log(iterator.next());  
> { value: 42, done: false }  
> console.log(iterator.next());  
> { value: 39, done: false }  
> console.log(iterator.next());  
> { done: true }  
>
```

# Consuming Iterables with `for..of`

```
> for(var x of [42,39,17]) {  
    console.log(x);  
}  
  
> 42  
  
> 39  
  
> 17  
  
>
```

If generator functions  
return iterators....

...why aren't they called  
Iterator functions?





# A Generator is an Iterator

yield value

```
generator.next().value;
```

throw an error

```
try { generator.next(); }  
catch(err) { ... }
```

return value

```
var pair = generator.next();  
if (pair.done) alert(pair.value);
```



pull

# A Generator is an **Observer**

receive data

```
generator.next(5);
```

receive an error

```
generator.throw("fail");
```

receive return value

```
generator.return(5);
```



push

The background features two black silhouettes of human profiles facing each other. The silhouette on the left is on the left side of the frame, and the silhouette on the right is on the right side. They are set against a solid yellow background.

Iteration only allows data  
to flow one-way.

doe



Generators allow feedback.

done

# Waiting with Async Iteration

```
function* getStockPrice(name) {  
    var symbol = yield getStockSymbol(name);  
    var price = yield getStockPrice(symbol);  
    return price;  
}
```

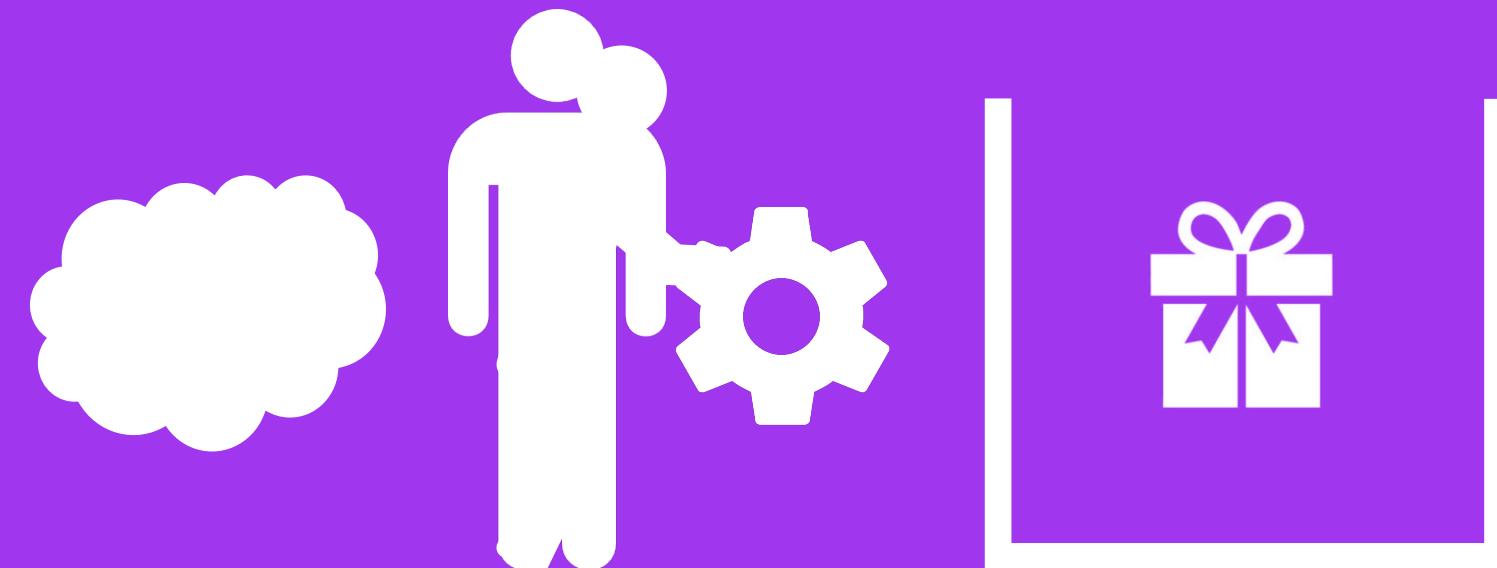
```
> spawn(getStockprice("Pfizer")).  
then(  
    price => console.log(price),  
    error => console.error(error));
```

```
> 353.22
```

# Async Iteration

Consumer

Producer

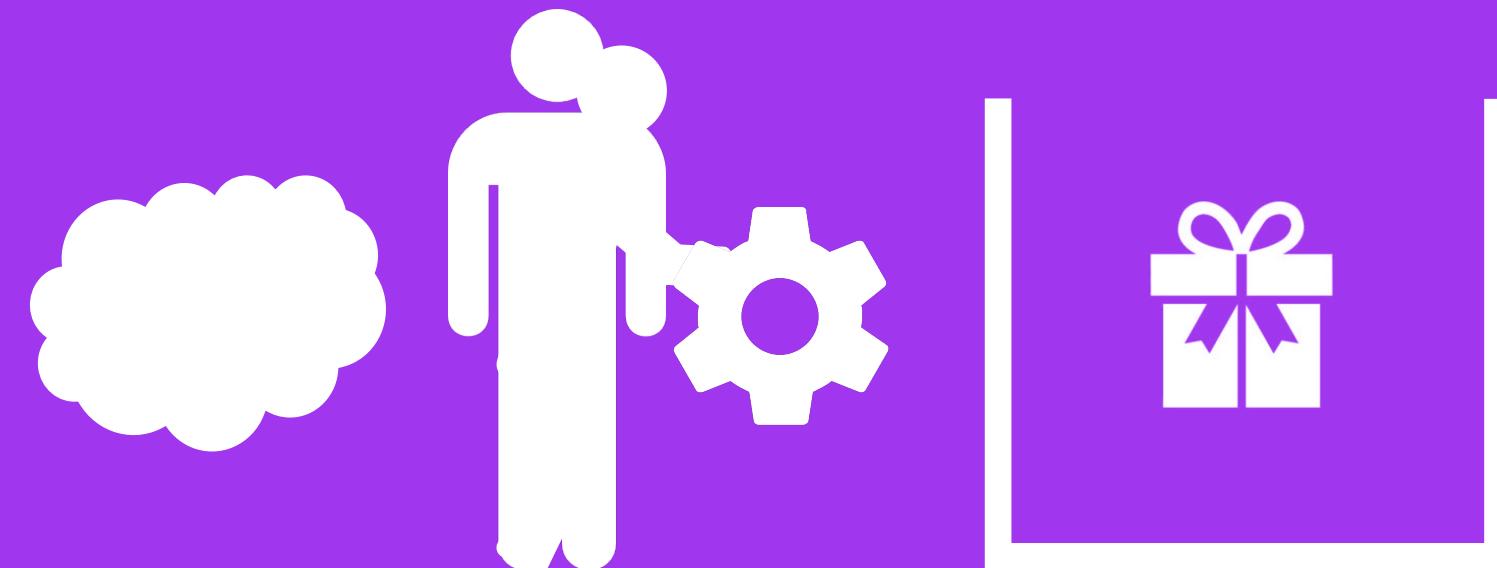


```
result = await data.getStockSymbol('Pfizer'); var symbol = yield getStockSymbol('Pfizer');
```

# Async Iteration

Consumer

Producer



```
result = await (generator.next("PDF"))
```

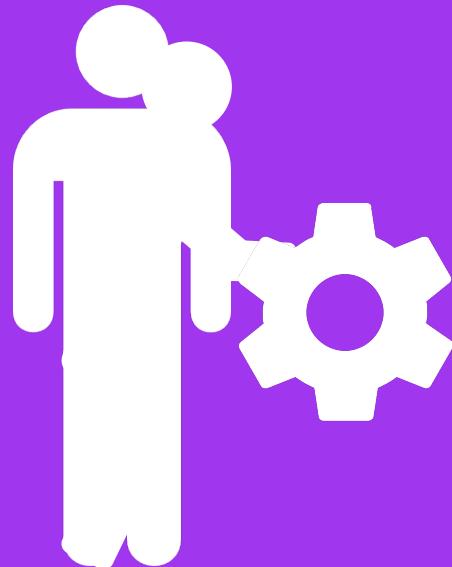
```
var symbol=yielded.value; if (symbol === 'PDF')
```

# Async Iteration

Consumer

Producer

27.83



done

```
var result = generator.next("27.83")
```

```
return pipe(result.value, getStockPrice(symbol));
```

# Asynchronous Iteration

```
function* getStockPrice(name) {  
    var symbol = "JNJ";  
    var price = 420.25;  
    return price;  
}
```

producer



```
function spawn(generator) {  
    return new Promise((accept, reject) => {  
        var onResult = 420.25 => {  
            var {value, done} = {value: 420.25, done: true};  
            if (!done) {  
                value.then(onResult, reject);  
            }  
            else accept(value);  
        };  
        onResult();  
    });  
}
```

consumer



```
> spawn(getStockPrice("Johnson and Johnson")).then(console.log);  
> 420.25
```



# Waiting with Task.js

```
function* getStockPrice(name) {  
    var symbol = yield getStockSymbol(name);  
    var price = yield getStockPrice(symbol);  
    return price;  
}  
  
var result =  
  spawn(getStockPrice.bind(null, "Pfizer"));  
  
result.then(console.log, console.error);
```

# Get Task.js

A screenshot of a web browser window showing the taskjs.org homepage. The address bar at the top left contains the URL "taskjs.org". To its right is a dark grey button with the word "mozilla" in white. The main content area features a large red "task.js" logo with a small sunburst icon to its left. Below the logo is the text "generators + promises = tasks". A descriptive paragraph follows, stating that task.js is an experimental library for ES6 that makes sequential, blocking I/O simple and beautiful, using the power of JavaScript's new yield operator. Another paragraph explains that Tasks are interleaved like threads, but they are cooperative rather than pre-emptive, and provides an example using jQuery. A code snippet is shown in a dark blue box.

taskjs.org

mozilla

 task.js

**generators + promises = tasks**

task.js is an experimental library for ES6 that makes sequential, blocking I/O simple and beautiful, using the power of JavaScript's new `yield` operator.

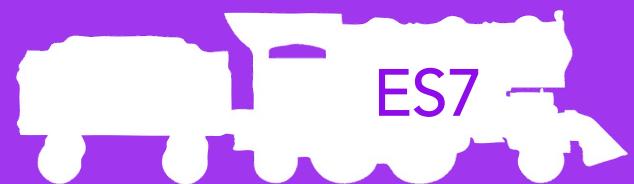
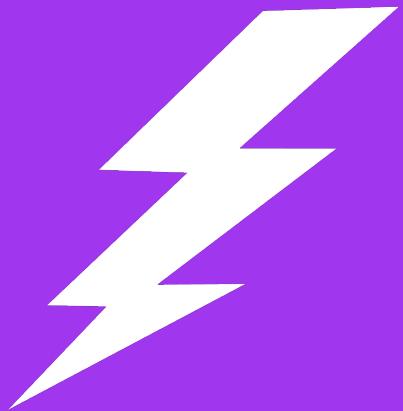
Tasks are interleaved like threads, but they are **cooperative** rather than **pre-emptive**: they block on **promises** with `yield`. Here's an example using **jQuery**:

```
spawn(function*() {
  var data = yield $.ajax(url);
  // do something with data
})
```

Why should easy waiting  
require a library?



# Async Functions



# Async Functions in ES7

```
async function getStockPrice(name) {
  var symbol = await getStockSymbol(name);
  var price = await getStockPrice(symbol);
  return price;
}

var result = spawn(getStockPrice.bind(null,("PFE")));
result.then(console.log, console.error);
```

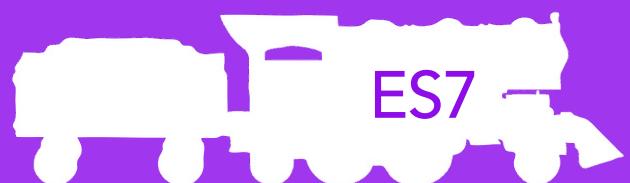
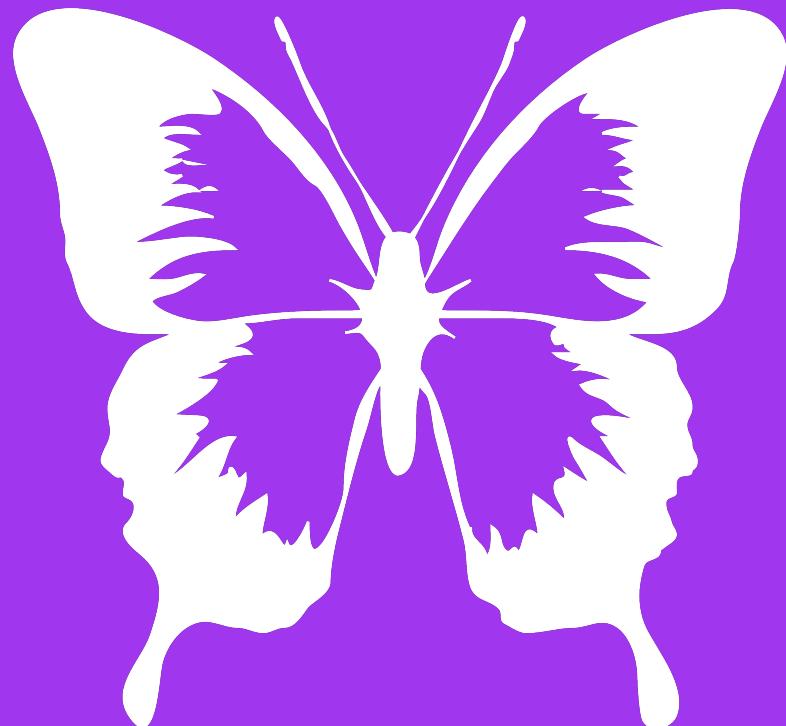
sync

```
function getStockPrice(name) {  
    var symbol = getStockSymbol(name);  
    var price = getStockPrice(symbol);  
    return price;  
}
```

async

```
async function getStockPrice(name) {  
    var symbol = await getStockSymbol(name);  
    var price = await getStockPrice(symbol);  
    return price;  
}
```

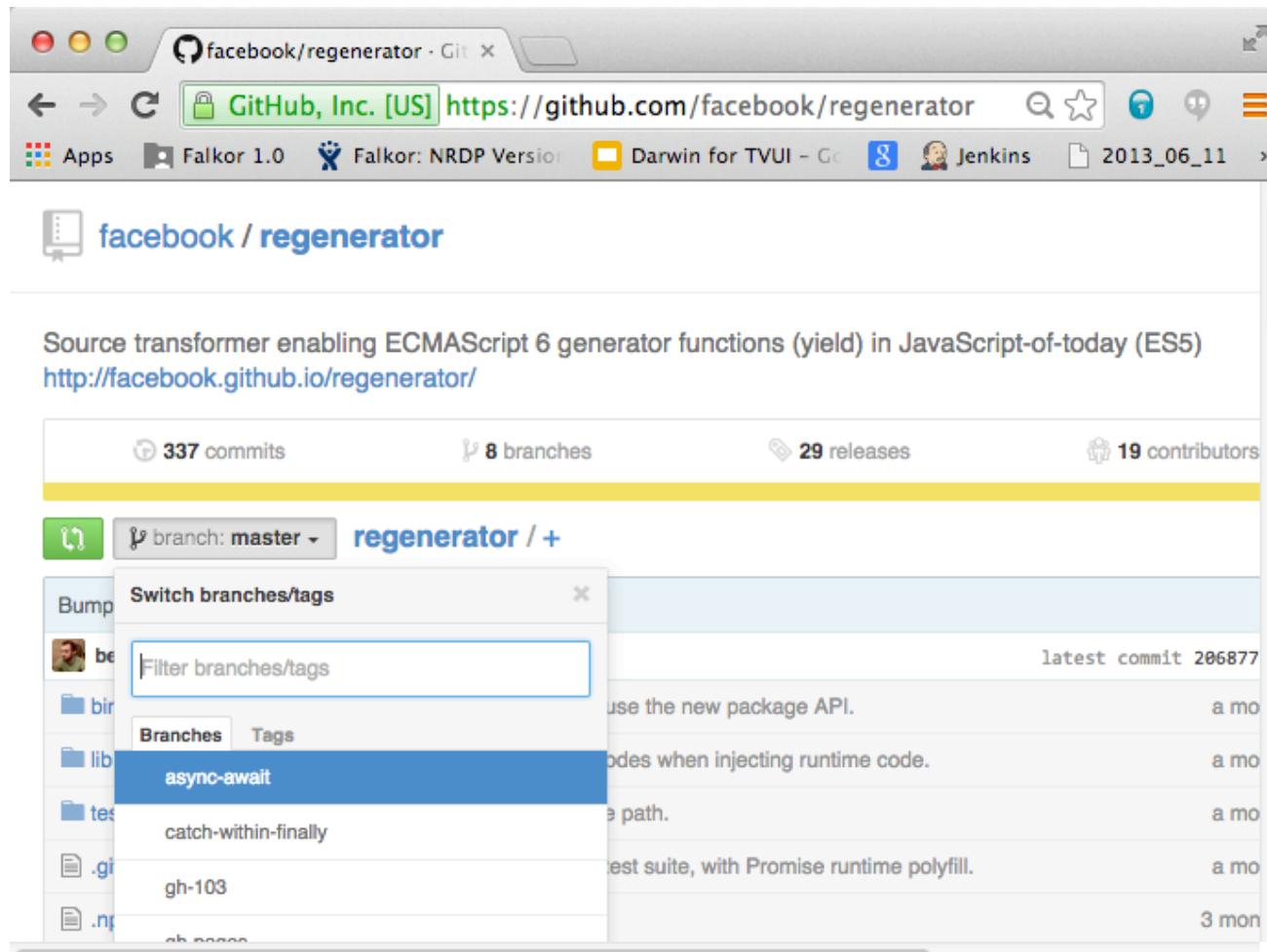
Symmetrical support for  
push and pull functions.





Async Functions

# Try Async Functions



The `await` keyword makes it easy  
to wait on an `async` value...

...but what if you need to wait on  
an async **stream** of values?



# Waiting on a stream with `for...on`

```
async function getNextPriceSpike(stock, threshold) {
  var delta,
    oldPrice,
    price;

  for(var price on new WebSocket("/prices/" + stock)) {
    if (oldPrice == null) {
      oldPrice = price;
    }
    else {
      delta = Math.abs(price - oldPrice);
      oldPrice = price;
      if (delta > threshold) {
        return { price: price, oldPrice: oldPrice };
      }
    }
  }
}

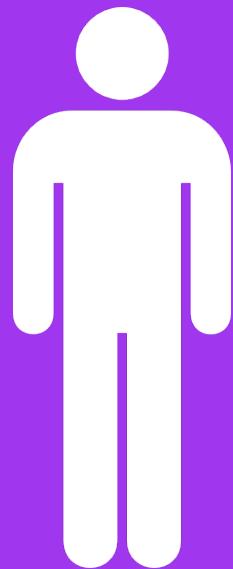
> getNextPriceSpike("JNJ", 2.50).then(diff => console.log(diff));
> { price: 420.22, oldPrice: 423.19 }
```

Unfortunately there's a problem.

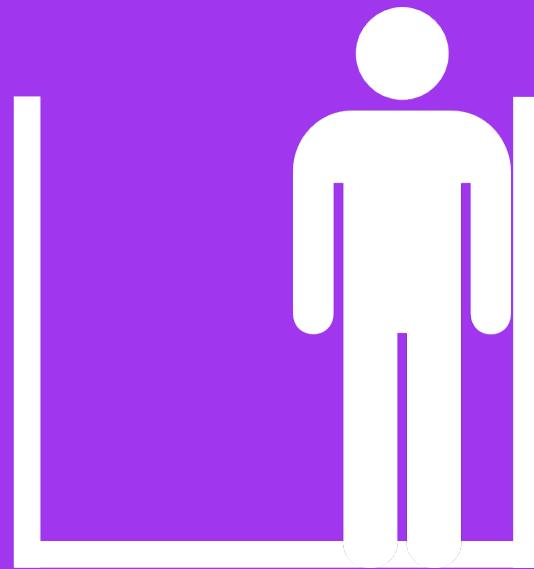
The web has no standard  
**Observable** interface.

# Observation

Consumer



Prosumer



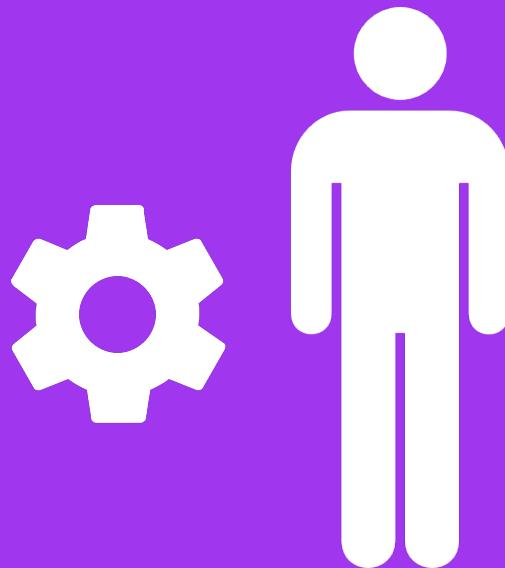
# Observation

Producer

Consumer



`observer.next(data)`



`producer.observer(observer);`

# The Web's Many Observable APIs

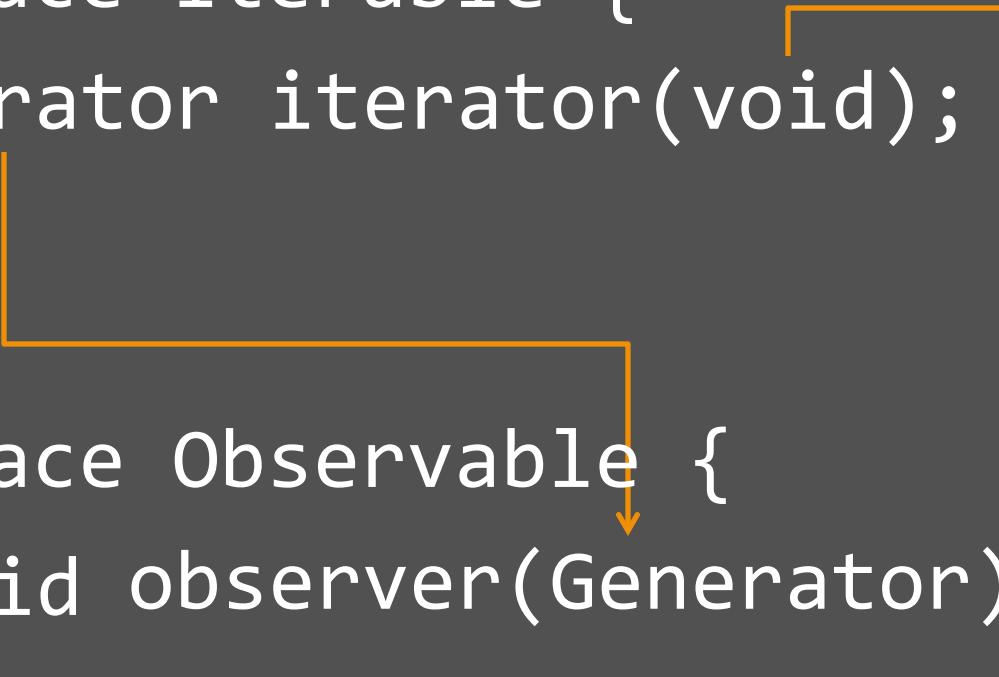
- DOM Events
- Websockets
- Server-sent Events
- Node Streams
- Service Workers
- jQuery Events
- XMLHttpRequest
- setInterval



push

# Introducing Observable

```
interface Iterable {  
    Generator iterator(void);  
}  
  
interface Observable {  
    void observer(Generator);  
}
```



# Observer pattern

```
> nums().observer({  
  next(v) { console.log(v); },  
  return(v) { console.log("done:" + v); }  
  throw (e) { console.error(e); },  
}); █  
> 1  
> 2  
> done: 3 █
```

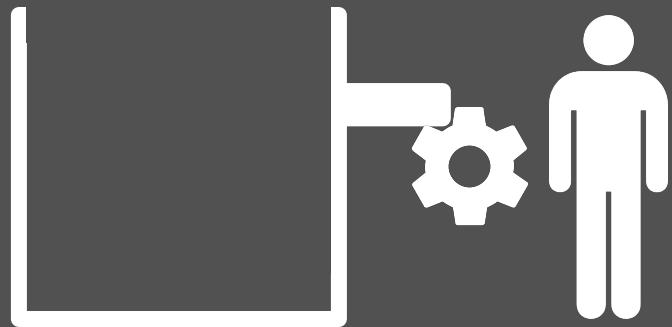


# Adding Sugar to Observation

```
nums().observer({
  next(v) { console.log(v); },
  return(v) { console.log("done:" + v); }
  throw (e) { console.error(e); },
});

nums().
  forEach(v => console.log(v)).          // next
  then(    v => console.log("done:"), // return
            e => console.error(e));      // throw

(async function writeNums() {
  try {
    for(var v on nums()) {
      console.log(v);
    }
  } catch(e) { console.error(e); }
  console.log("done");
})();
```



# Push APIs implement Observable

DOM Events

Websockets

Server-sent Events

Node Streams

Service Workers

jQuery Events

XMLHttpRequest

setInterval

Observable

# Implementing Observable

```
WebSocket.prototype.observer = function(generator) {
  var cleanUp,
    message = v => {
      var pair = generator.next(v);
      if (pair.done)
        cleanUp();
    },
    error = e => {
      cleanUp();
      generator.throw(e);
    },
    close = v => {
      cleanUp();
      generator.return(v);
    };

  cleanUp = () => {
    this.removeEventListener("message", message);
    this.removeEventListener("error", error);
    this.removeEventListener("close", close);
  };
  this.addEventListener("message", message);
  this.addEventListener("error", error);
  this.addEventListener("close", close);
};
```

# Consuming Observables in ES7

```
> (async function() {  
  for(var member on new WebSocket("/signups")){  
    console.log(JSON.stringify(member));  
  }  
}());   
> { firstName: "Tom" ... }  
> { firstName: "John" ... }  
> { firstName: "Micah" ... }  
> { firstName: "Alex" ... }  
> { firstName: "Yehuda" ... }  
> { firstName: "Dave" ... }  
> { firstName: "Ben" ... }  
> { firstName: "Dave" ... }
```

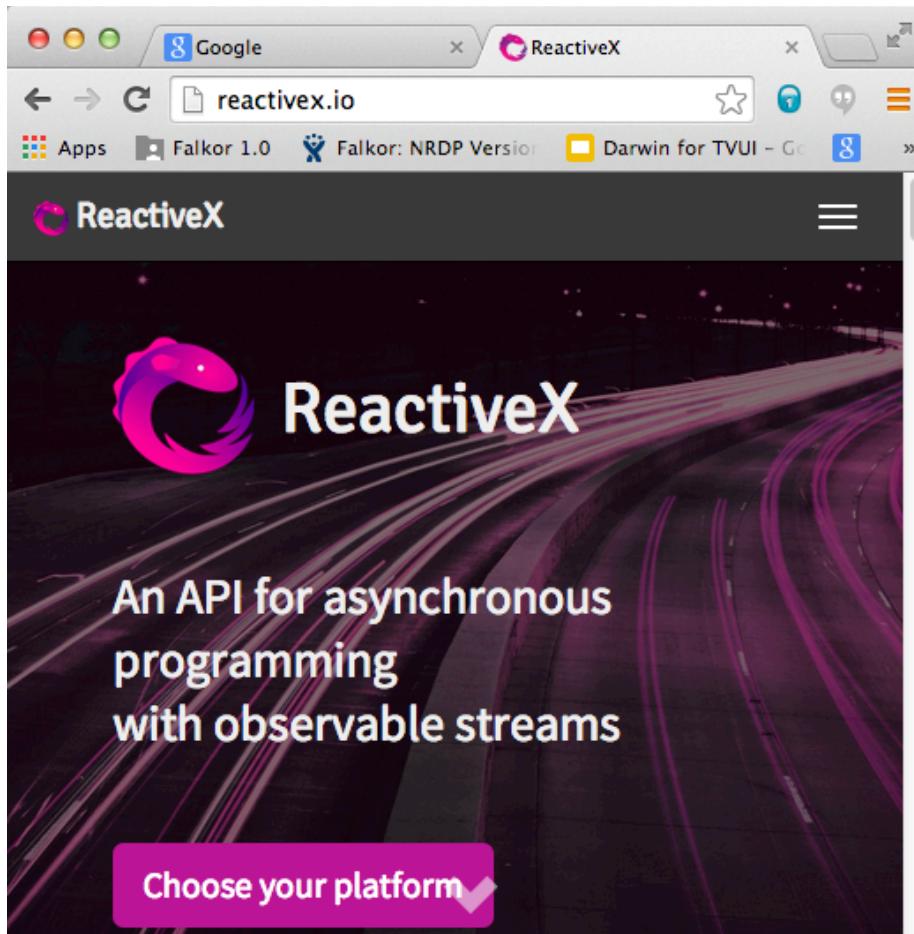
# Observable Composition

```
async function drag(element) {
  var downs = Observable.fromEvent(element, "mousedown");
  var ups = Observable.fromEvent(document, "mouseup");
  var moves = Observable.fromEvent(document, "mousemove");

  var drags =
    downs.
      map(down => moves.takeUntil(ups)).
      flatten();

  for(var drag on drags) {
    element.style.top = drag.offsetY;
    element.style.left = drag.offsetX;
  }
}
```

# Try Observable



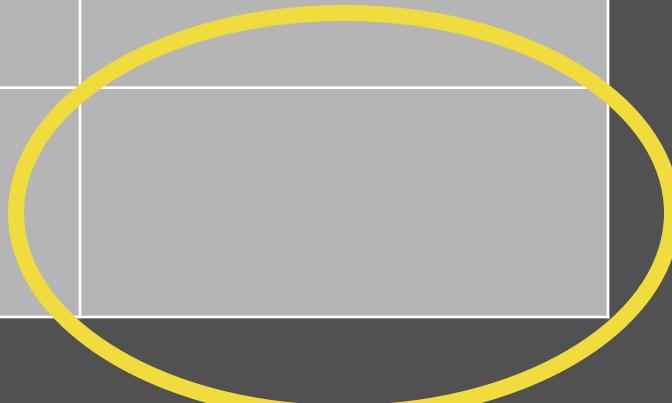
If an **async** function returns a Promise,  
and a **function\*** returns an Iterator...

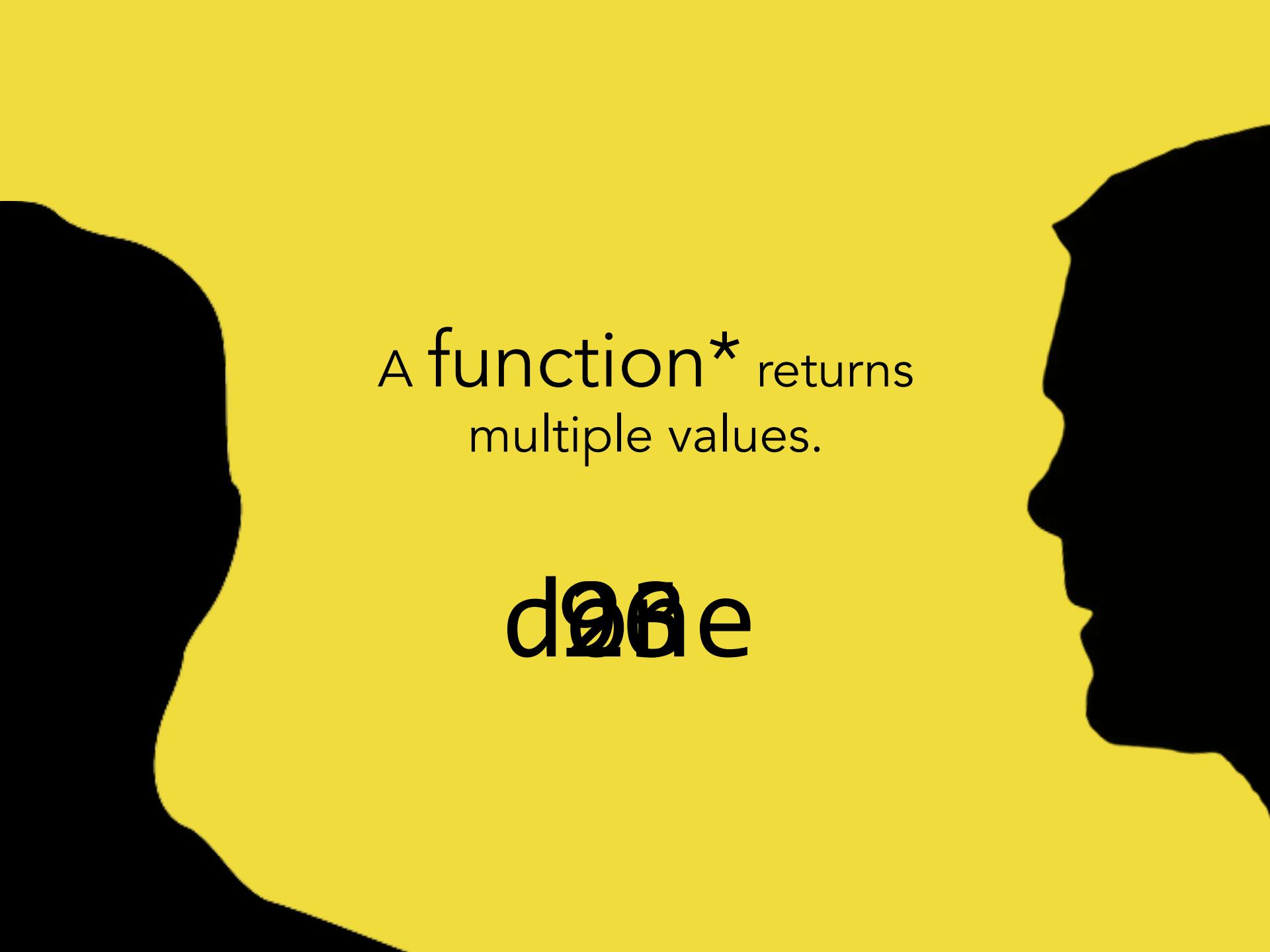
...what does an **async function\*** return?



# Function Types in ES7

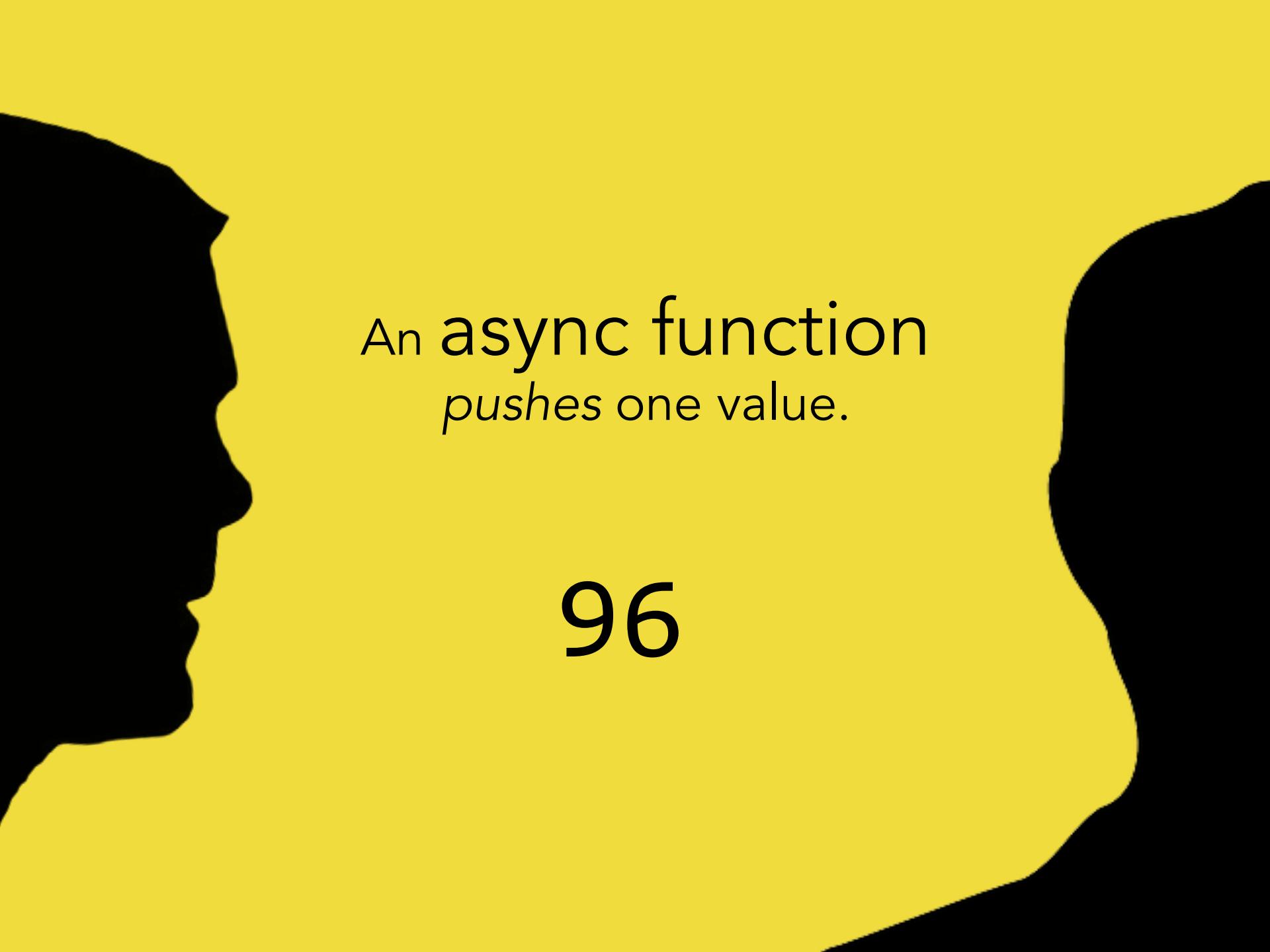
	Synchronous	Asynchronous
function	T	Promise
function*	Iterator	





A function\* returns  
multiple values.

done



An `async` function  
pushes one value.

96



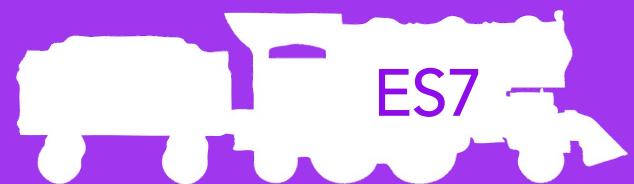
An **async** function\*  
*pushes multiple values.*

data

An async generator function  
returns an Observable.



# Async generator functions



# async function\* in action

```
async function* getPriceSpikes(stock, threshold) {
  var delta,
      oldPrice,
      price;

  for(var price on new WebSocket("/prices/" + stock)) {
    if (oldPrice == null) {
      oldPrice = price;
    }
    else {
      delta = Math.abs(price - oldPrice);
      oldPrice = price;
      if (delta > threshold) {
        yield { price: price, oldPrice: oldPrice };
      }
    }
  }
}
```

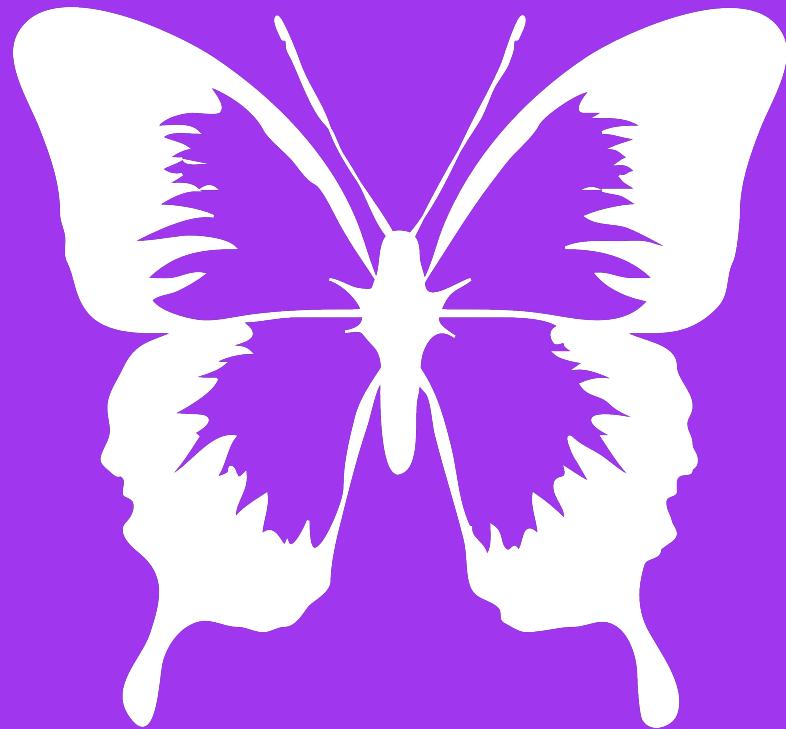
# Consuming Observables in ES7

```
> async function printSpikes() {
    for(var spike on getPriceSpikes("JNJ", 2.00)) {
        console.log(priceSpike);
    }
}());
```



```
> { oldPrice: 392.55, price: 396.82 }
> { oldPrice: 397.14, price: 395.03 }
> { oldPrice: 394.18, price: 392.01 }
> { oldPrice: 392.55, price: 400.92}
> { oldPrice: 401.53, price: 403.12}
> { oldPrice: 404.22, price: 407.74}
> { oldPrice: 403.28, price: 405.88}
> { oldPrice: 406.18, price: 409.29}
```

ES7 should have symmetrical support for push and pull streams.



How can we make **async I/O** easier?



# Sync I/O is easy with function\*

```
function* getStocks() {
    var reader = new FileReader("stocks.txt");
    try {
        while(!reader.eof) {
            var line = reader.readLine();
            yield JSON.parse(line);
        }
    }
    finally {
        reader.close();
    }
}

function writeStockInfos() {
    var writer = new FileWriter("stocksAndPrices.txt");
    try {
        for(var name of getStocks()) {
            var price = getStockPrice(name);
            writer.writeLine(JSON.stringify({name, price}));
        }
    }
    finally {
        writer.close();
    }
}
```

producer



consumer



# Async I/O with `async` function\*

```
async function* getStocks() {
    var reader = new AsyncFileReader("stocks.txt");
    try {
        while(!reader.eof) {
            var line = await reader.readLine();
            await yield JSON.parse(line);
        }
    } finally {
        await reader.close();
    }
}

async function writeStockInfos() {
    var writer = new AsyncFileWriter("stocksAndPrices.txt");
    try {
        for(var name on getStocks()) {
            var price = await getStockPrice(name);
            await writer.writeLine(JSON.stringify({name, price}));
        }
    } finally {
        await writer.close();
    }
}
```

producer



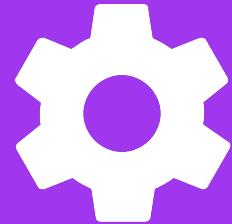
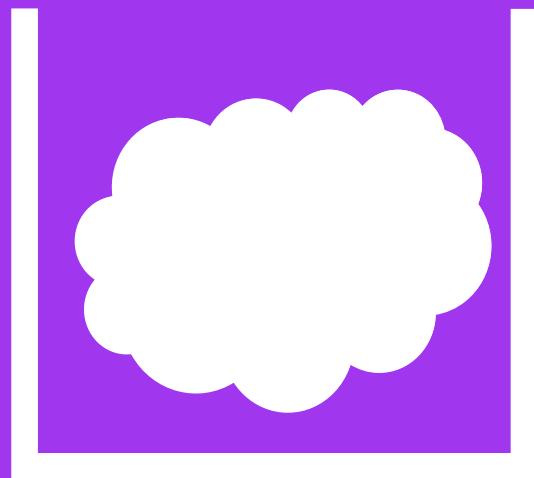
consumer



# Async Observation

Producer

Consumer



# Async IO with `async` function\*

```
async function* getStocks() {
  var reader = new AsyncFileReader("stocks.txt");
  try {
    while(!reader.eof) {
      var line = await reader.readLine();
      await yield JSON.parse(line);
    }
  } finally {
    await reader.close();
  }
}

async function writeStockInfos() {
  var writer = new AsyncFileWriter("stocksAndPrices.txt");
  try {
    for(let stock of await getStocks()) {
      var price = await getStockPrice(stock.name);
      await writer.writeLine(JSON.stringify({name, price}));
    }
  } finally {
    await writer.close();
  }
}
```

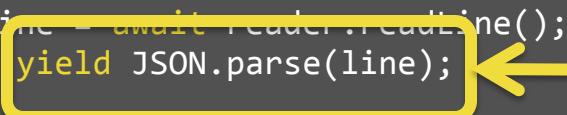
promise returned  
from `async` next fn

# Async IO with `async` function\*

```
async function* getStocks() {
  var reader = new AsyncFileReader("stocks.txt");
  try {
    while(!reader.eof) {
      var line = await reader.readLine();
      await yield JSON.parse(line);
    }
  } finally {
    await reader.close();
  }
}

async function writeStockInfos() {
  var writer = new AsyncFileWriter("stocksAndPrices.txt");
  try {
    await getStocks().forEach(async function(name) {
      var price = await getStockPrice(name);
      await writer.writeLine(JSON.stringify({name, price}));
    });
  } finally {
    await writer.close();
  }
}
```

promise returned  
from `async` next fn



```
await yield JSON.parse(line);
```



```
await getStocks().forEach(async function(name) {
```

# Function Types in ES7

	Synchronous	Asynchronous
function	T	Promise
function*	Iterator	Observable



# More Info on async function\*

A screenshot of a web browser window showing a GitHub repository page. The address bar at the top displays "GitHub, Inc. [US] https://github.com/jhusain/asyncgenerator". The page header includes the GitHub logo, navigation links for "This repository" and "Search", and a horizontal menu with "Explore", "Gist", "Blog", and "Help".



Asynchronous Generators for ES7 — Edit



async function\*

# ES Maturity Stage 0: Strawman

*“Allow for feedback”*



# Resources

- esdiscuss.org
- <https://facebook.github.io/regenerator/>
- <http://taskjs.org/>
- <http://github.com/jhusain/asyncgenerator>

# Questions?



# Observer Pattern with Events

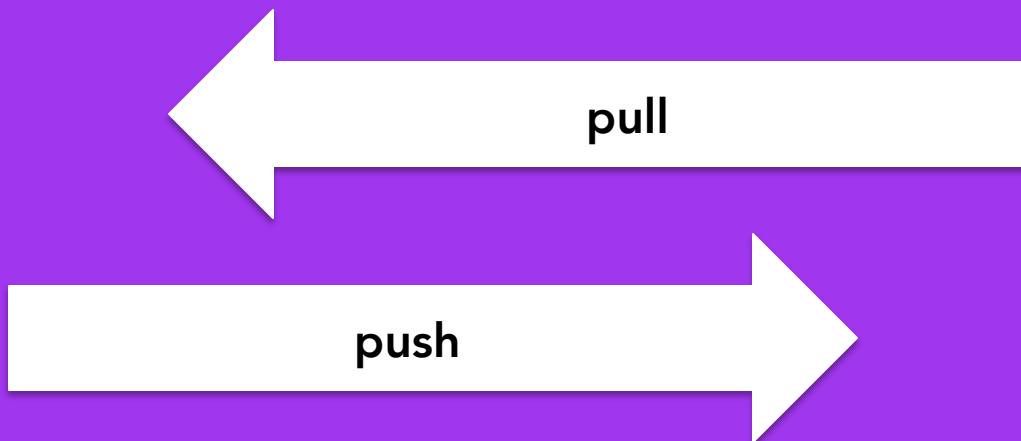
```
> document.addEventListener(  
  "mousemove",  
  function next(e) {  
    console.log(e);  
  }); □
```

```
> { clientX: 425, clientY: 543 }  
> { clientX: 450, clientY: 558 }  
> { clientX: 455, clientY: 562 }  
> { clientX: 460, clientY: 743 }  
> { clientX: 476, clientY: 760 }  
> { clientX: 476, clientY: 760 }  
> { clientX: 476, clientY: 760 }
```

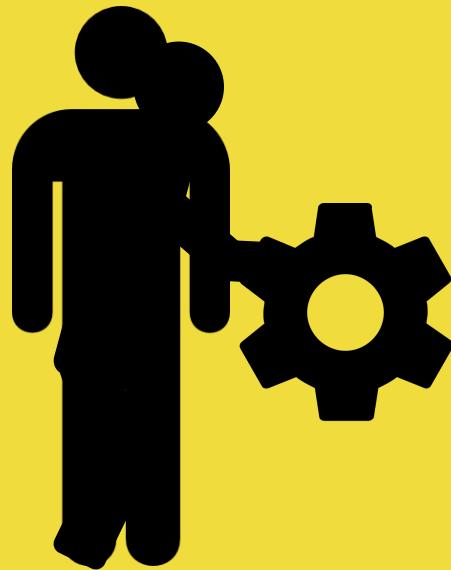
# Observer Pattern with setInterval

```
> var counter = 0;  
  setInterval(function next(e) {  
    console.log(counter++);  
  }, 20);   
> 0  
> 1  
> 2  
> 3  
> 4  
> 5  
> 6  
> 7
```

The decision to push or pull is  
*orthogonal* to most algorithms.



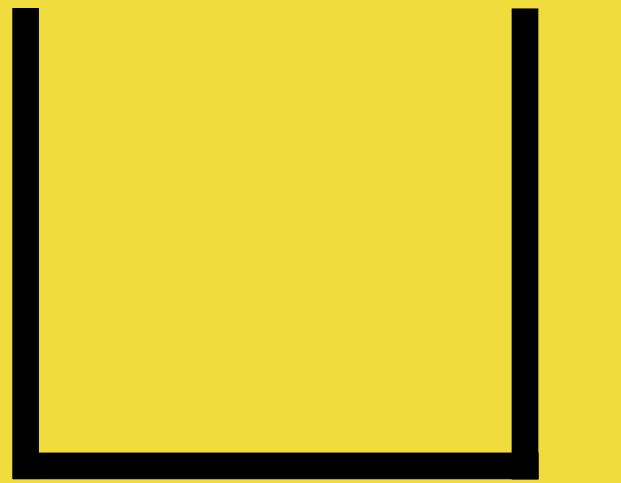
# Iteration is Pull



```
var result = generator.next();
```

pull

# Observation is Push



generator.next(5);

push

# Iteration and Observation



are *symmetrical*