Practical JavaScript

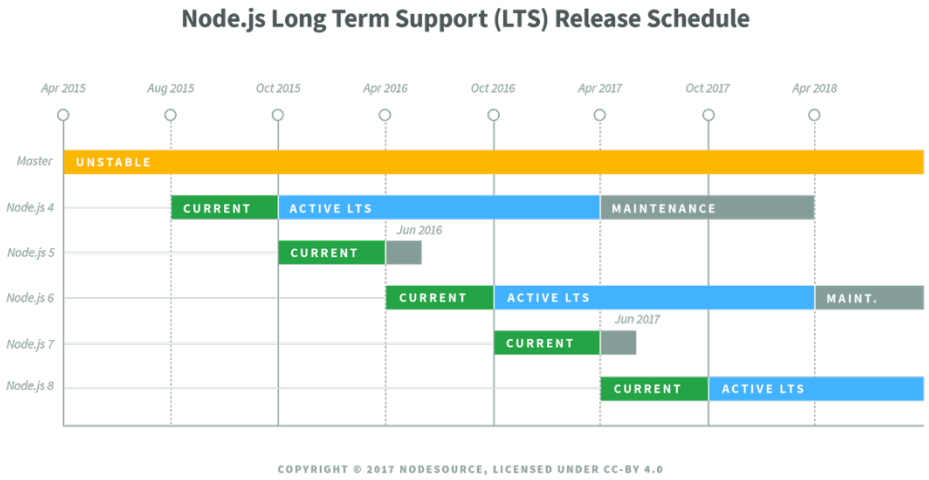
Ethan Brown

Language Versions & Terminology

- JavaScript (JS) = ECMAScript (ES), for all practical intents and purposes
- ES2015 = ES6 = Harmony
- ES2016, ES2017: yearly language updates moving forward
- ES5: High browser compatibility
- TC39: Language specification committee
- Ben McCormick on JavaScript versioning
- Kangax compatibility table

Node.js

- What's Node?
- LTS = Long-term Support
- Current LTS: 6.x (Boron)
- nvm: Easy way to switch between Node versions
- What's npm?
  - Package manager
  - Not just for Node



JavaScript Ecosystem: The Big Players

- Express: Node-based web server
- React: Front-end framework (Facebook)
  - Unopinionated, flexible
  - Offers path to other platforms with React Native and React VR
  - Flux and Redux: important, powerful paradigms, good for larger projects
- Angular: Front-end framework (Google)
  - Relatively opinionated, complex, robust
  - Possibly path to mobile with NativeScript

What about persistence?

- MongoDB: robust, stable, popular
- Firebase (Google): simple, offers other services like authentication and storage
- DynamoDB (Amazon): robust, good choice if using AWS
- CoucheDB (Apache): up-and-coming, good synchronization/offline support
- Redis: key-value pair database, good for caching, sessions

What You'll Need: Local Environment

	Windows	Linux	OSX
Version control	Git	Git OR: sudo apt-get git (Ubuntu/Mint) OR: sudo yum install git (Fedora)	Git OR: brew install git
Terminal (Bash)	Git Bash (installed with Git)	Bulit-in	Built-in (⌘**+Space, type** terminal)
Text editor	Visual Studio Code (recommended) Vim / Emacs Atom Notepad++	Visual Studio Code (recommended) Vim / Emacs Atom	Visual Studio Code (recommended) Vim / Emacs Atom Brackets
Node	Node 6.x LTS	Node 6.x LTS	Node 6.x LTS
Web browser	Chrome Firefox Opera	Chrome Firefox Opera	Chrome Firefox Opera

Cloud 9

1. Go to https://c9.io
2. Sign in or sign up if you don't already have an account
3. Once you have logged in, click on "Create a new workspace"
4. Give your workspace a name (example: "practical-javascript")
5. Choose "Node.js" template
6. Press "Create workspace"
7. In the terminal, type `nvm install 6`
8. In the terminal, type `npm install -g yarn` (ignore deprecation warning)

Test Frameworks

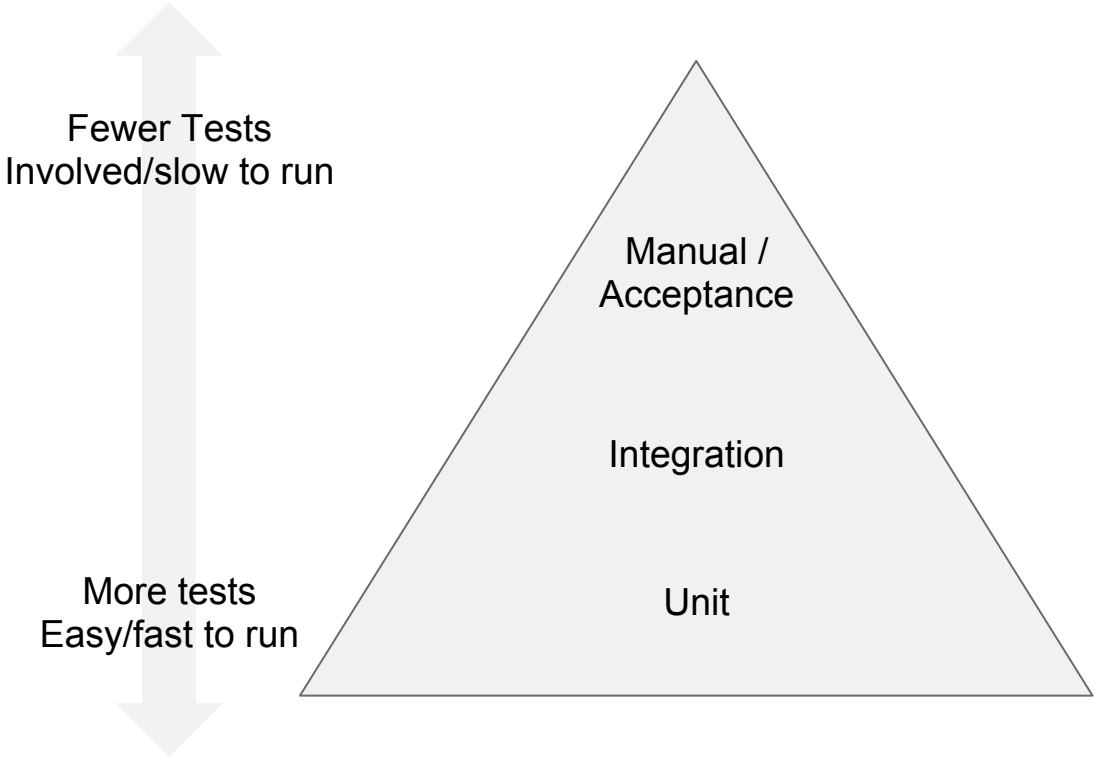
- Jest
  - Newcomer
  - Designed for React development, but can be used in any context
  - Great tools
- Jasmine
  - Popular, robust
  - Used by Jest, by default
- Mocha

Assertion Libraries

- Node's built-in assert: convenient in a pinch
- Jest assertions (expect): ideal if you're using Jest
- Chai: popular, robust, can be used with Jest
- Must
- Should.js

Types of Test and the Test Pyramid

- **Unit tests**
  - Numerous!
  - Fast and easy to run!
  - Written by devs
  - High coverage
- **Integration/acceptance tests**
  - Much fewer than unit tests
  - Complicated to write/ automate



Testing Front-End Code

- Historically difficult problem
- Selenium/WebdriverIO automates browser testing
- BrowserStack/Sauce Labs make Selenium easier (for $$)
- React has *great* picture for front-end testing in Jest/Enzyme

The project: weighted scoring matrix (WSM)

Criteria	Weight	Scott	Julie	Chelsea
Documentation	30%	8	9	9
Maintainability	40%	6	9	8
Clarity	20%	9	7	9
Elegance	10%	5	8	9
SCORE	**100%**	**7.1**	**8.5**	**8.6**

Add Criteria Add Person

Your first unit test

- Create directory for tests: mkdir -p app/__tests__
- Create new file (right click on app/__tests__ in Workspace browser, click on “New File”, enter wsm.test.js)
- Double-click on newly created file, and enter these contents:

```
const wdm = require('../wsm.js')

test('wsm module should export a function', () => {
    expect(typeof wsm).toBe('function')
})
```

Install Jest and configure project

- Install Jest: yarn add -D jest
- Edit package.json file, and add the following (anywhere is fine; I usually do it between “author” and “dependencies”):

```
"author": "Mostafa Eweda <mo.eweda@gmail.com>",
"scripts": {
  "test": "jest"
},
"jest": {
  "roots": ["<rootDir>/app"]
},
"dependencies": {
```

- You can now run tests: npm test (or yarn test)...they’ll fail!

Pass the tests!

- Create new file (right click on app in Workspace browser, click on “New File”, enter wsm.js)
- Double-click on newly created file, and enter these contents:

```
function wsm() {
}

module.exports = wsm
```

- You can now run tests: npm test (or yarn test)...they’ll fail!

Other Useful Resources

- CodePen: great for font-end developers, portfolios
- jsBin: similar to CodePen, less designer-y
- ESLint: makes working on a team less contentious!
- Airbnb styleguide (minus semicolons)
- Standard JS (minus comma-dangle)
- Stack Overflow (obviously)
- Front-End Focus

Practical JavaScript

Ethan Brown

Callbacks: JavaScript Institution Since 1996

Declaration (or definition) is not *referencing* is not *invocation* (or calling):

Declaration:

```
function hello() {
  console.log('hello!')
}
```

Reference:

```
hello
```

Invocation:

```
hello()
```

Functions are first-class citizens

Aliasing/storing:

```
const anotherHello = hello
```

Passing one function to another:

```
anotherFunction(hello)
```

Returning one function from another:

```
function yetAnotherFunction() {
        return hello
}
```

Callbacks: JavaScript Institution Since 1996

Declaration:

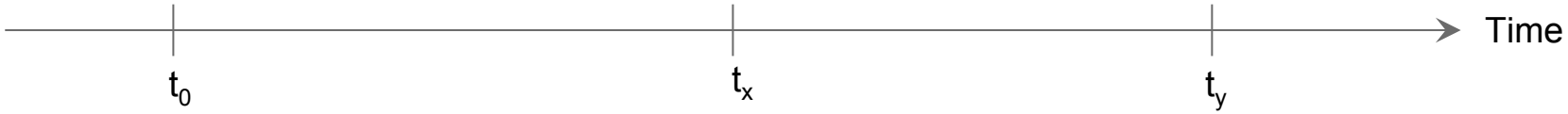
```
function hello() {
  console.log('hello!')
}
```

Reference:

```
setTimeout(hello, 1000)
```

Invocation (by *setTimeout*):

```
hello()
```



What the heck is async anyway?

Synchronous

Code	Execution
console.log(1)	0:00.001: 1
console.log(2)	0:00.002: 2
console.log(3)	0.00.003: 3

Asynchronous

Code	Execution
setTimeout(() => console.log(1), 5)	0:00.001: 2
console.log(2)	0:00.002: 3
console.log(3)	0.00.006: 1

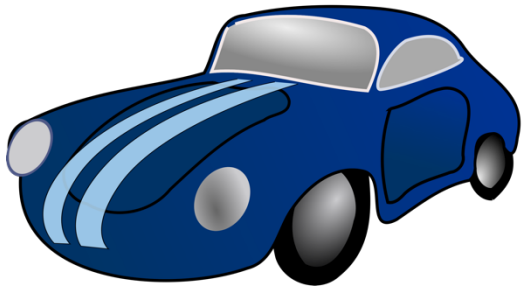
Why do we need async?

- User actions (clicking, typing, etc.)
- Network activity (requests take time)
- Disk activity (disks are are slow)
- Long-running processes

Why is async hard?

- Cognitive dissonance: we write code in one sequence, it executes in another
- Data synchronization becomes an issue
- Determining the source of errors becomes more difficult

```
function createCar() {
  return {
    state: {
      ignition: 'off',
      brakes: 'off',
      accelerator: 'off',
    },
    start() { this.state.ignition = 'on' },
    stop() { this.state.ignition = 'off' },
    brake(t) {
      this.state.brakes = 'on'
      setTimeout(() => this.state.brakes = 'off',  t)
    },
    accelerate(t) {
      this.state.accelerator = 'on'
      setTimeout(() => this.state.accelerator = 'off', t)
    },
  }
}
```



segment-2/example-01.js

Car callbacks

```
function createCar(onStart, onStop, onBrake, onAccelerate) {
  return {
    //...
    start() {
      this.state.ignition = 'on'
      onStart()
    }
    //...
}
const c = createCar(() => console.log('car started!'))
c.start()
```

3 callback invoked!

1 callback provided

2 will result in callback!

segment-2/example-02.js

Car events

segment-2/example-03.js

```
function createCar() {
  return {
    state: {
      ignition: 'off',
      brakes: 'off',
      accelerator: 'off',
    },
    eventHandlers: {
      start: [],
    },
    on(name, handler) {
      this.eventHandlers[name].push(handler)
    },
    start() {
      this.state.ignition = 'on'
      this.eventHandlers[‘start’].forEach(h => h())
    },
}
const c = createCar()
c.on('start', () => console.log('started!')
c.on('start', () => console.log('varoom!')

c.start()
```

3 events emited!

1 event handlers registered

2 will result in events being handled!

EventEmiter

Documentation: https://nodejs.org/api/events.html

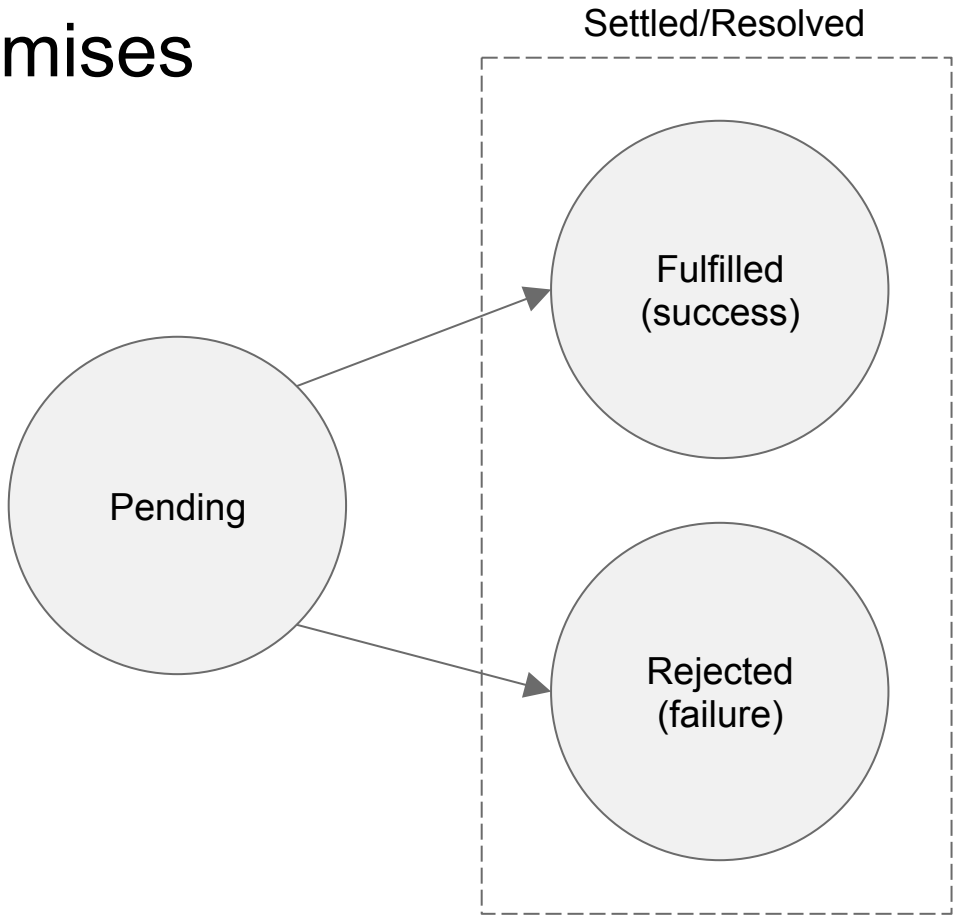
```
const EventEmitter = require('events')
```

class Foo extends EventEmitter (if using traditional OOP)

Object.assign(Object.create(EventEmitter.prototype), *objContents*)

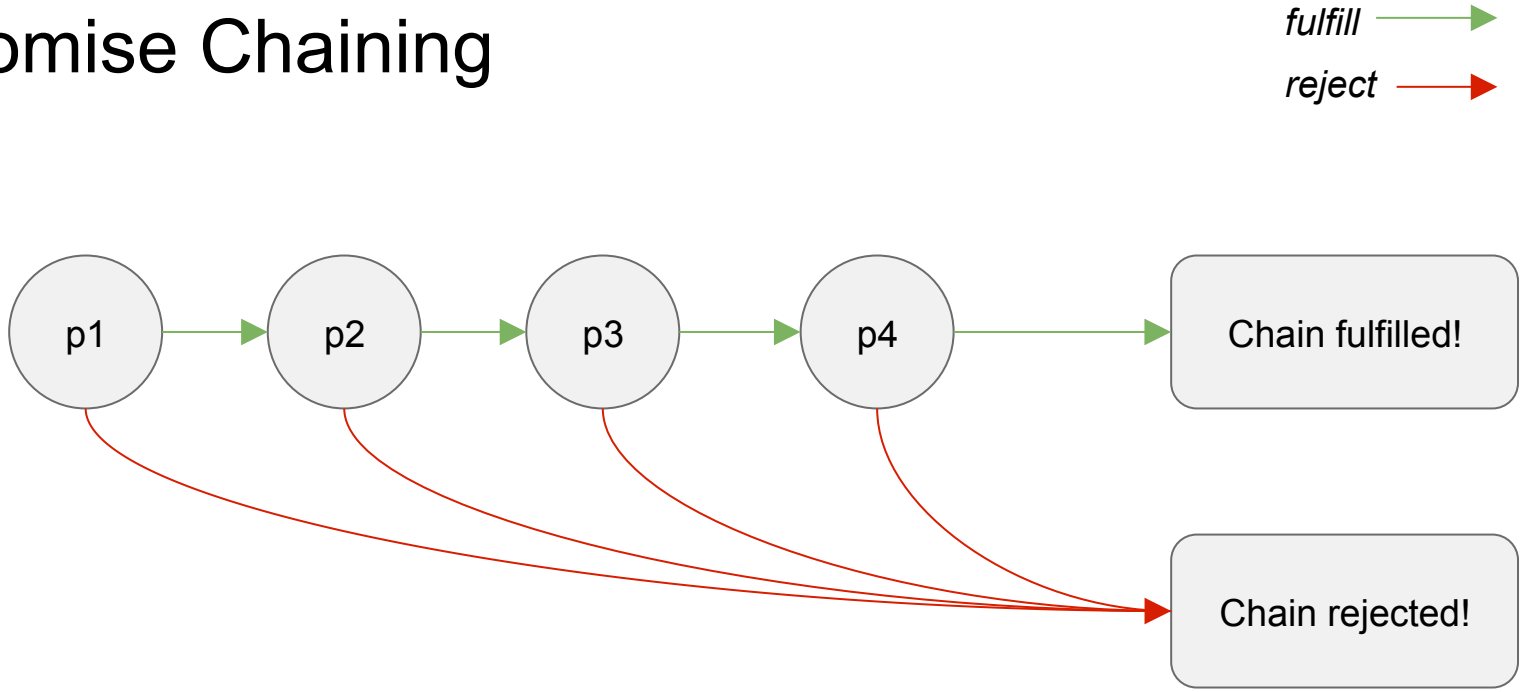
See *segment-2/example-04.js*

Promises



A promise can ONLY ever be in one of these states. Note that once a promise has settled (fulfilled or rejected), it cannot move back to the pending state, or move between fulfilled and rejected.

Promise Chaining

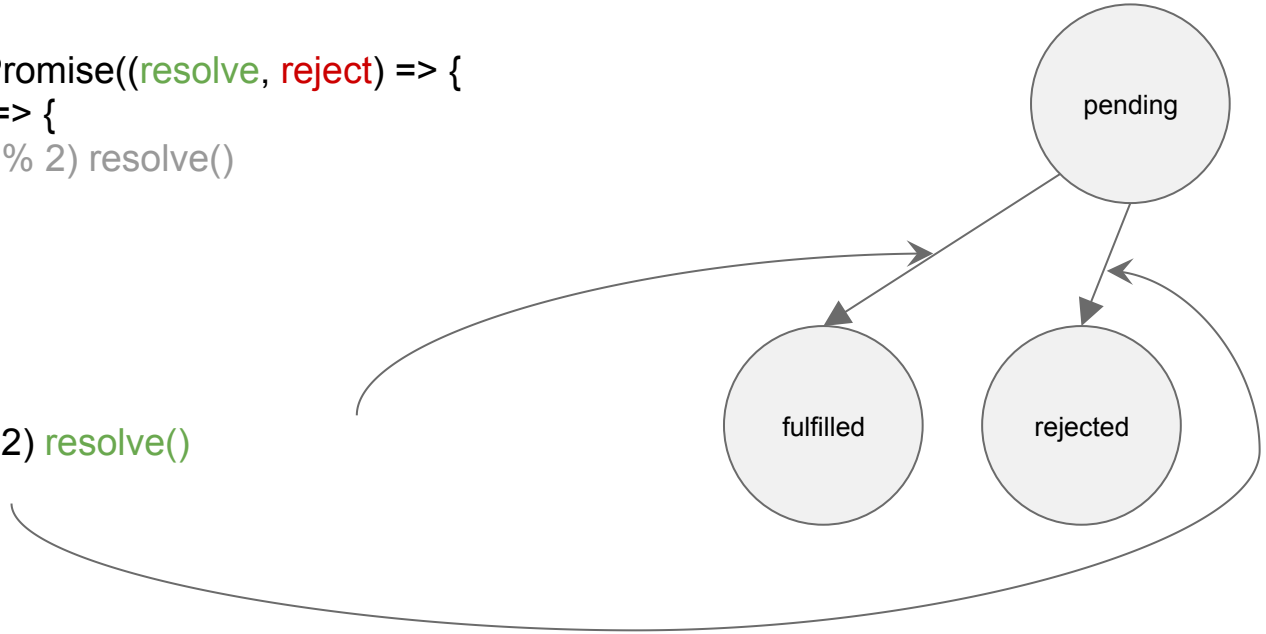


Promise Syntax: Creation and Resolution

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    if(Date.now() % 2) resolve()
    else reject()
  }, 500)
})
```

After ~ 500 ms:

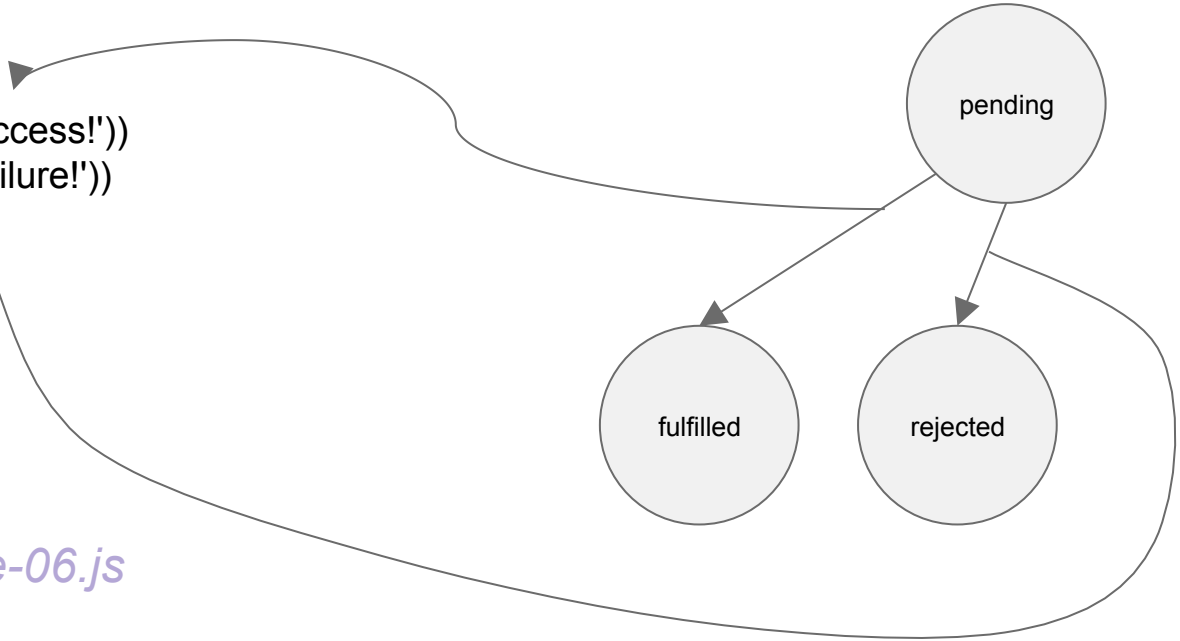
```
if(Date.now() % 2) resolve()
else reject()
```



See *segment-2/example-05.js*

Promise Syntax: Detecting Resolution

```
p
  .then(() => console.log('success!'))
  .catch(() => console.log('failure!'))
```



See *segment-2/example-06.js*

Promise Resolution is Asynchronous

```
p1
  .then(() => console.log("Promise 1 fulfilled!")

p2
  .then(() => console.log("Promise 2 fulfilled!")
```

Is there any guarantee about which message you will see first?

See *segment-2/example-07.js*

Promise Resolution is Asynchronous

```
p1
  .then(() => {
    console.log("Promise 1 fulfilled!")
    p2
      .then(() => console.log("Promise 2 fulfilled!")
  })
```

Now is there any guarantee about which message you will see first?

See *segment-2/example-08.js*

Promise Syntax: Chaining

Anatomy of a promise-returning function (PRF):

```
function prf() {
  return new Promise((resolve, reject) => {
    // do successful stuff and resolve() or
    // reject() on failure
  }
}
```

Chaining PRFs:

```
prf1()
  .then(prf2)
  .then(prf3)
  .then(prf4)
  .catch(err => console.error(err.message))
```

Advantages:

- Easy to read!
- Compact!

Disadvantages:

- Flow of data is obscured
- Less flexible
- prf1 is handled differently than prf2-4

Promise-Returning Functions: Minor Improvement

Chaining PRFs with initial fulfillment:

```
Promise.resolve()
  .then(prf1)
  .then(prf2)
  .then(prf3)
  .then(prf4)
  .catch(err => console.error(err.message))
```

Advantages:

- Easy to read!
- Compact!
- Symetrical (prf1 is handled just like prf2-4)

Disadvantages:

- Flow of data is obscured
- Less flexible

See *segment-2/example-13.js*

Promise Pitfalls: Premature Invocation

```
Promise.resolve()
  .then(prf1())
  .then(prf2())
  .then(() => console.log('all done!'))
```

What's wrong here?

See *segment-2/example-11.js*

Promises & Error Handling: The Wrong Way

```
try {
  Promise.resolve()
    .then(prf1)
    .then(prf2)
} catch(err) {
  console.log('Whoops!')
}
```

This won't work as expected...why?

See *segment-2/example-16.js*

Promises & Error Handling: The Right Way

```
Promise.resolve()
  .then(prf1)
  .then(prf2)
  .catch(err => console.error(err.message))
```

Sync and Async: Don't Mix & Match

```
function biasedFunction(n) {
  if(n % 2 === 0) return console.log('have a happy day!')
  return new Promise(resolve => {
    setTimeout(() => {
      console.log('you are very odd')
    }, 2000)
  })
}
```

Bad idea...why?

See *segment-2/example-18.js*

Sync and Async: Don't Mix & Match

```
function biasedFunction(n) {
  if(n % 2 === 0) return new Promise(resolve => {
    console.log('have a happy day!')
  })
  return new Promise(resolve => {
    setTimeout(() => {
      console.log('you are very odd')
    }, 2000)
  })
}
```

Consistent interface!

See *segment-2/example-20.js*

Summary 1

- Callbacks, events, and promises are all techniques for dealing with asynchronous code
- Callbacks are now relatively passe, but….
- Promises require callbacks...
- Events require callbacks…
- So callbacks are okay as long as we're using them with promises or events!

Summary 2

Good

```
prf1
  .then(prf2)
  .then(prf3)
```

Better...

```
Promise.resolve()
  .then(prf1)
  .then(prf2)
  .then(prf3)
```

Best!

```
Promise.resolve()
  .then(() => prf1())
  .then(() => prf2())
  .then(() => prf3())
```

(Uglier, but more flexible and less prone to developer error; more OOP-friendly.)

Exercises

exercise-01.js: writing a function that uses a callback

exercise-02.js: writing a function that returns a promise

exercise-03.js: writing a function that returns a promise that is conditionally rejected

Practical JavaScript

Ethan Brown

Array Basics

Two ways to create arrays:

- const arr = [1, 2, 3, /*...*/] (array literal)
- const arr = new Array(10) (constructor...best used with #fill(value))

Other basics:

- Get/set array values with squre brackets (arr[3] / arr[3] = 'hello')
- Arrays in JS are nonhomogenous
- Assigning past the last element of the array increases the size of the array
  See *segment-3/example-01.js* and *segment-3/example-02.js*
- Uninitialized entries have value undefined *but behave differently; avoid!*

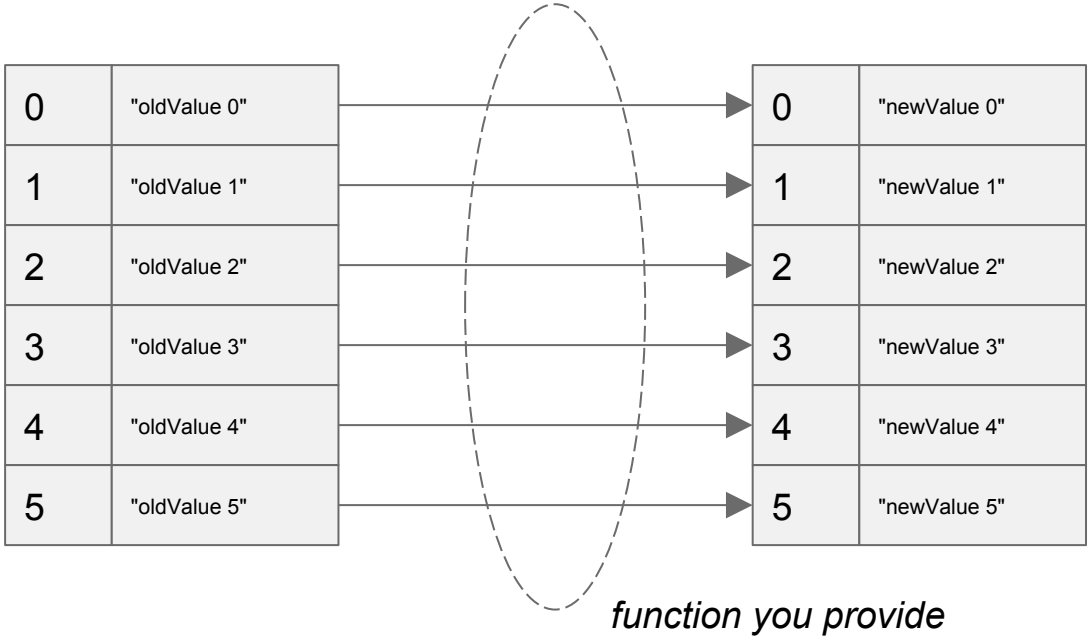
See *segment-3/example-03.js*

Common Methods for Array Mutation

Immutability is the new hotness: I recommend learning to think immutable. But array mutation methods do still have their place:

Mutate	**Non-mutating equivalent**
arr.push(1, 2, ,3)	arr2 = arr.concat(1, 2, 3)
arr.pop()	arr[arr.length - 1]
arr.unshift(1, 2, 3)	arr2 = [1, 2, 3].concat(arr)
arr.shift()	arr[0]
arr.splice(2, 0, 2.5)	arr2 = [...arr.slice(0, 2), 2.5, ...arr.slice(2)]
arr.splice(1, 2)	arr2 = [...arr.slice(0, 1), ...arr.slice(3)]
arr.splice(1, 1, 2.5)	arr2 = [...arr.slice(0, 1), 2.5, ...arr.slice(2)]

Array#map: Easy, Fun, Powerful



See *segment-3/example-04.js*

Array#map: The Rules

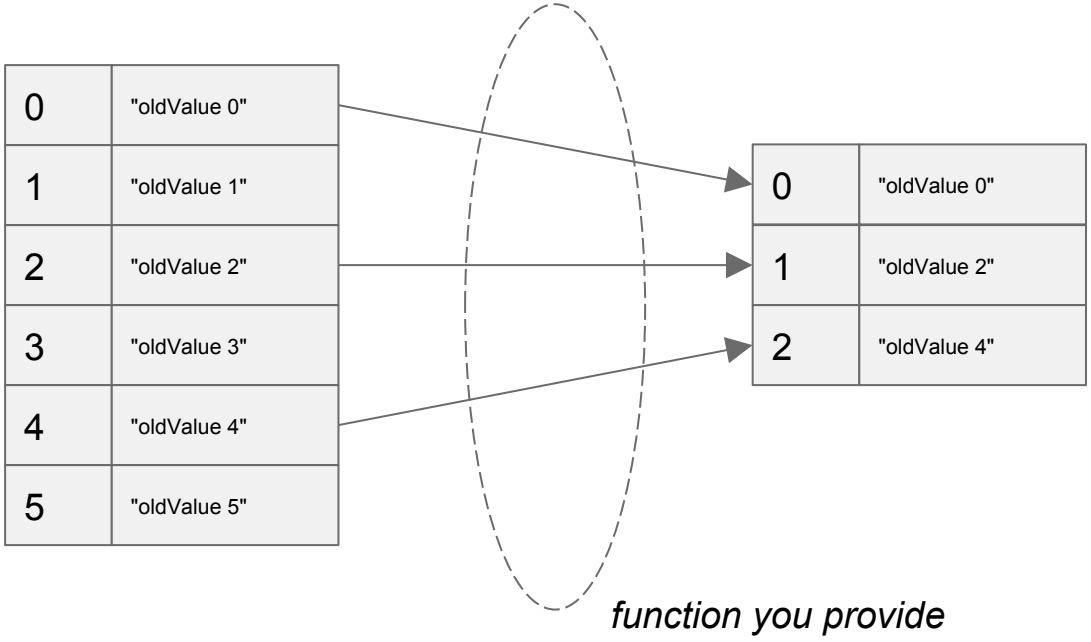
- *n* things in, *n* things out
- Everything gets mapped*
- Empty arrays map fine
- Any value / type can map to any different value / type
- Okay, not *everything* gets mapped...*unassigned values* don't...avoid unassigned values!

*But this is easy to get around...if you don't want to map certain elements, just map map them to themselves!

Array#map: The Syntax

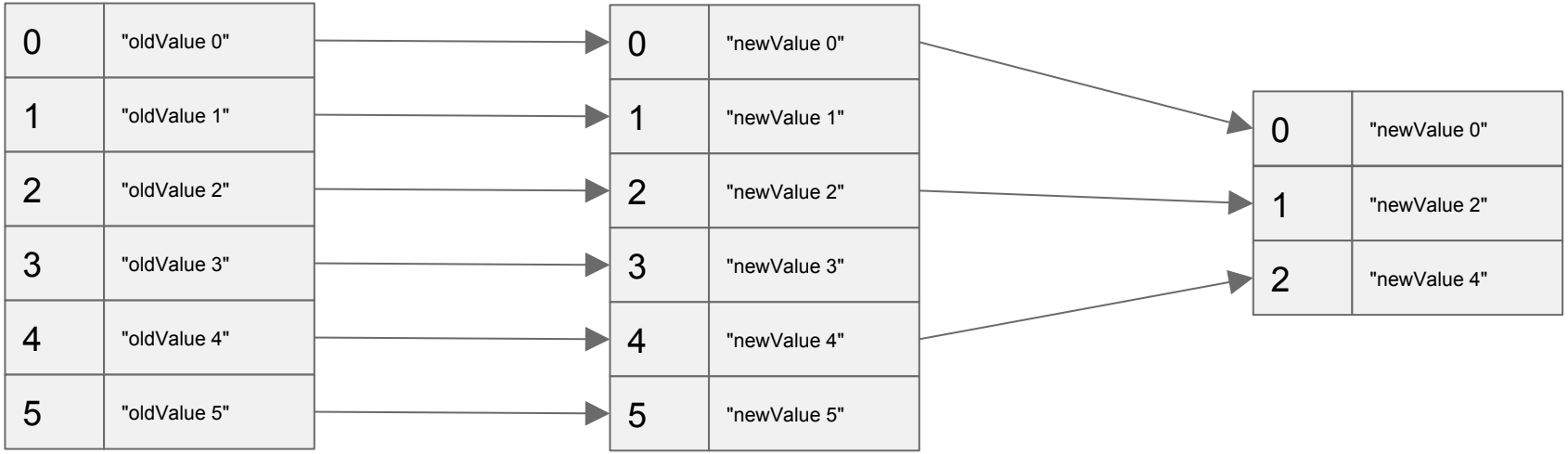
```
arr.map(x => value)                    // most common
arr.map((x, idx) => value)             // 2nd most common
arr.map((x, idx, arrRef) => value)     // almost never used

arr.map(x => value)     // DOESN'T need explicit return
arr.map(x => {
  return value                                    // DOES need explicit return
})
```

Array#filter: Easy, Boring, Useful!



See *segment-3/example-10.js*

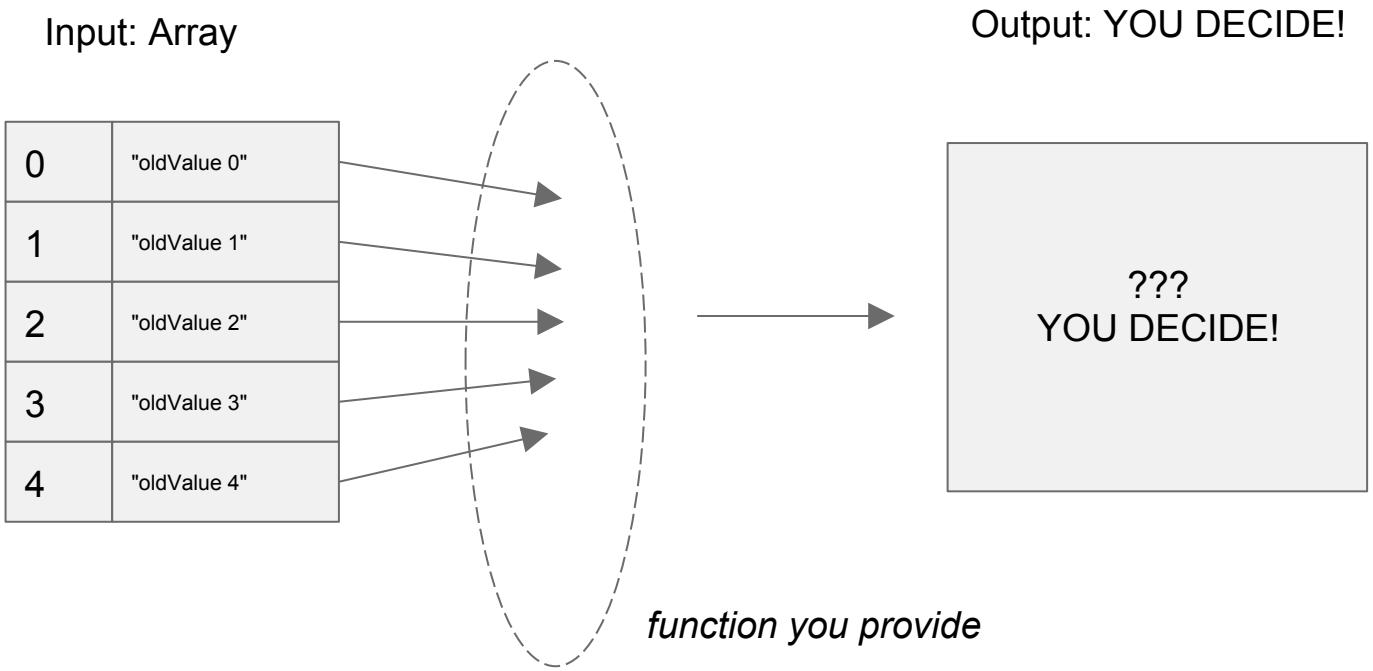
Array#map and Array#filter: Hand, Meet Glove



Array#reduce: The God Method

- Intimidating at first
- SUPER POWERFUL
- Can do everything that map AND filter can do and then some
- Name is somewhat misleading: think "transform" in your head
- No point in using if you can accomplish same thing with map/filter

Array#reduce: Hard to Draw



See *segment-3/example-14.js*

Array#reduce: Common Examples

Input	Output	Common usage
Array of numbers/objects: [3, 17, 5, 16, 12, 9.7]	Number	Average, variance / standard deviation, sum, product sum, etc.
Array of objects: [{ name: "foo" }, { name: "bar" }, /*...*/]	Object/Map { foo: { name: "foo" }, bar: { name: "bar" }, /*...*/ }	Creating map from array of objects

Array#reduce: Syntax

```
arr.reduce((a, x, i) => {
  // a is the "accumulator"
  // x is the current element in the array
  // i is its index
  return a      // don't forget to return a!
}, initialValue)
```

Array#reduce: What's Actually Happening?

Array Contents

0	"item 0"
1	"item 1"
2	"item 2"
3	"item 3"
4	"item 4"

Arguments Passed to Array#reduce Function

0: (initialValue, "item 0", 0)

1: (*return value from 0*, "item 1", 1)

2: (*return value from 1*, "item 2", 2)

3: (*return value from 2*, "item 4", 3)

4: (*return value from 3*, "item 4", 4)

Final return value: *return value from 4*

Array#reduce: "sum" example

[0, 1, 2, 3, 4].reduce((a, x) => a + x, 0)

Array Contents

0	0
1	1
2	2
3	3
4	4

Arguments Passed to Array#reduce Function

0: (initialValue, 0, 0) -> 0

1: (*return value from 0*, 1, 1) -> 1

2: (*return value from 1*, 2, 2) -> 3

3: (*return value from 2*, 3, 3) -> 6

4: (*return value from 3*, 4, 4) -> 10

Final return value: *return value from 4*: 10

Map: why do we need it?

Object in JavaScript are maps, with some limitations:

- Only strings and symbols can be used as keys
- Counting number of properties in an object is difficult
- Prototypal inheritance can result in unexpected properties

See *segment-3/example-15.js*

Map: pros and cons

Pros:

- Keys can be anything (objects, for example)
- Properties are easy to count
- Easy to construct from array of pairs

Cons:

- More "wordy" than objects

Set: Let's Get Mathematical

Sets: very much like arrays, but duplicates are ignored:

```
const s = new Set()
s.add("red")
s.add("blue")
s.add("green")
console.log(s)    // Set { "red", "blue", "green" }
set.add("blue")   // no error!
console.log(s)    // Set { "red", "blue", "green" }
```

Exercises

exercise-01.js: using Array#map

exercise-02.js: using Arrray#map together with Array#filter

exercise-03.js: using Array#reduce

Practical JavaScript

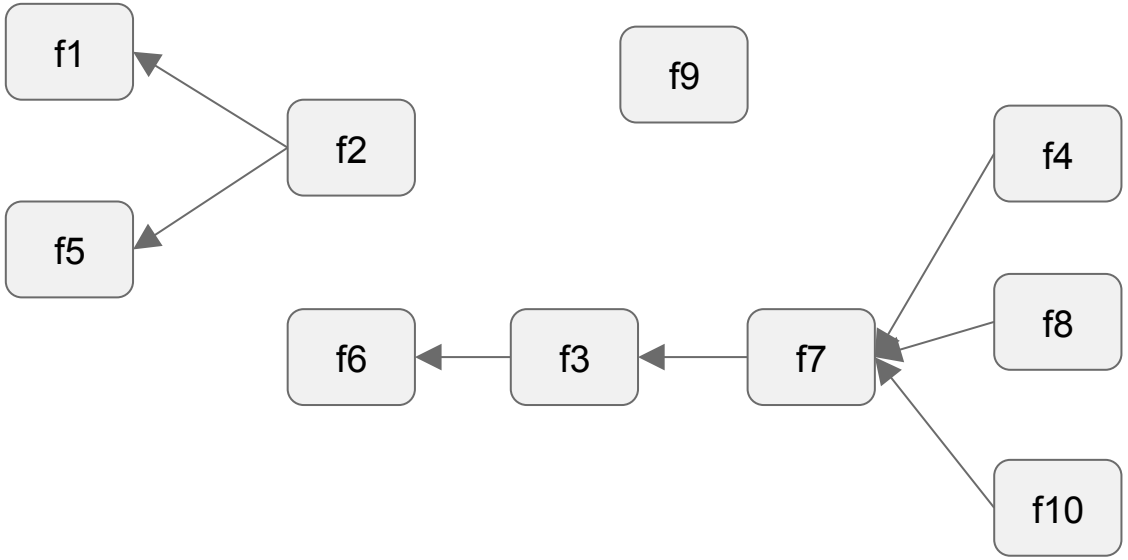
Ethan Brown

Parallel Promise Execution

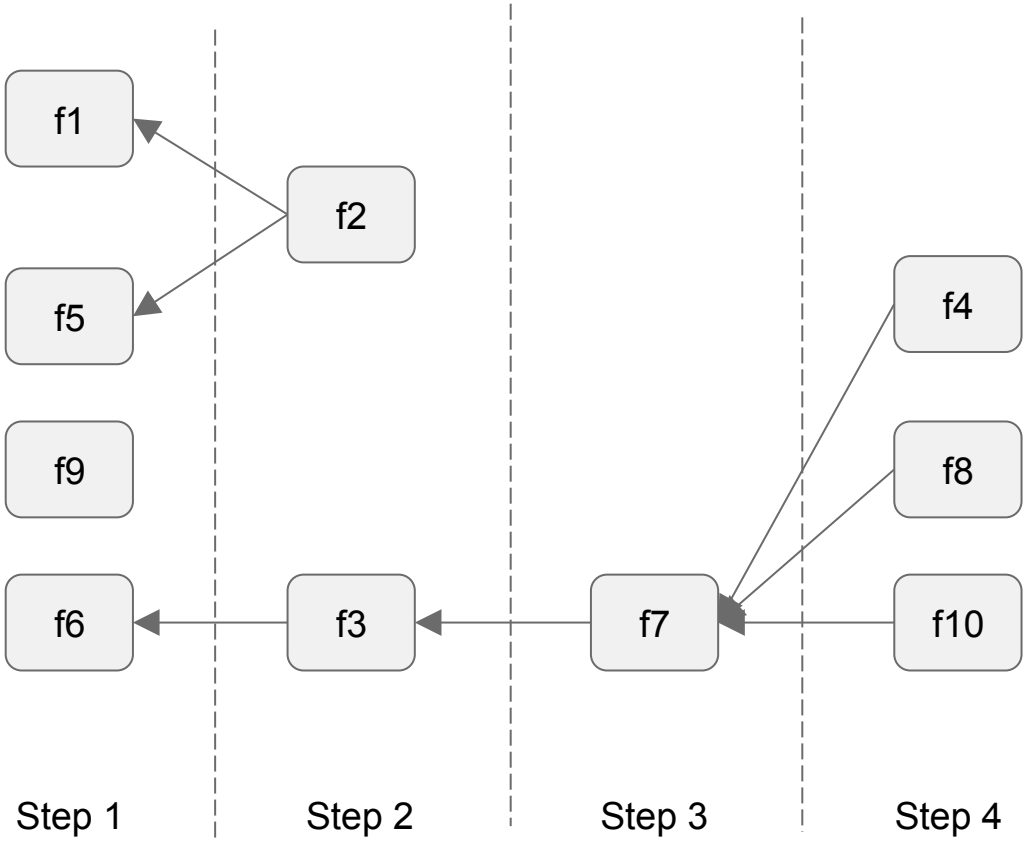
- Why execute promises in parallel? *For efficiency / avoid wasted time.*
- Is it *really* parallel? *No, not really...but you can think of it that way.*
- You still have to think about dependencies!

Dependencies

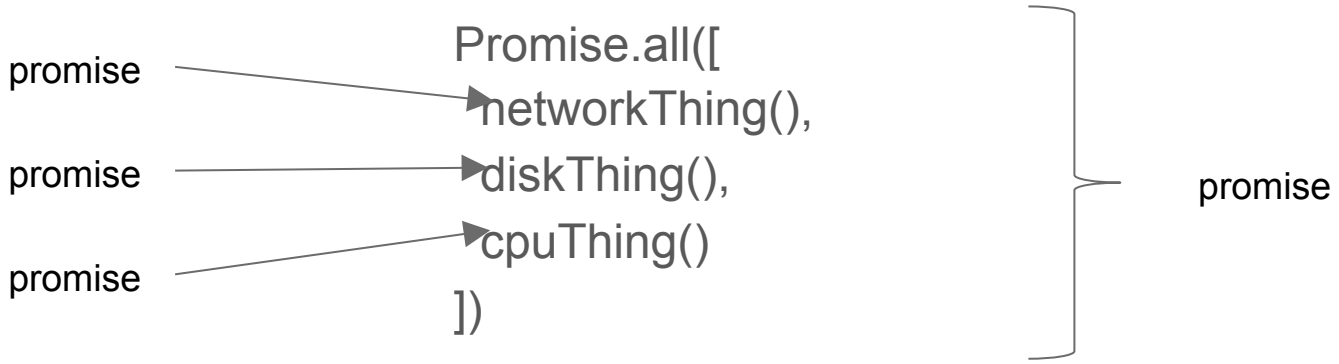
Given function f1-f10, the functions may have dependencies:



Dependencies: What Can Run When?



Parallel Promises: Promise.all

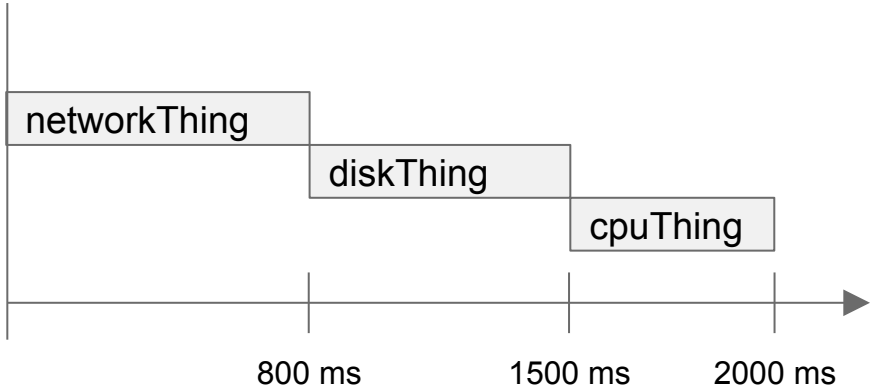


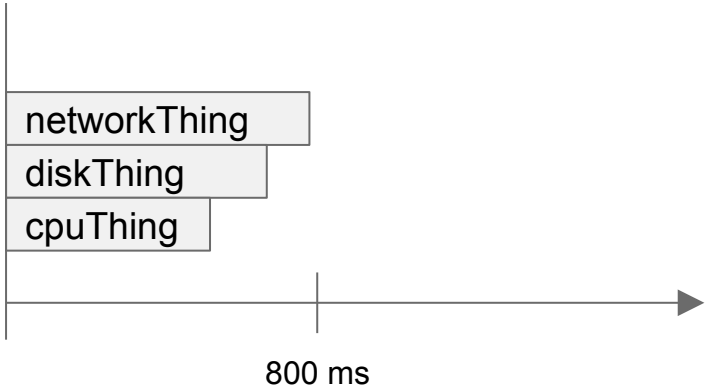
See *segment-4/example-02.js*

Serial vs. Parallel

```
Promise.resolve()
  .then(networkThing)
  .then(diskThing)
  .then(cpuThing)
```

```
Promise.all([
  networkThing(),
  diskThing(),
  cpuThing()
])
```





Promise.all: What About Return Values?

```
Promise.all([
  networkThing(),
  diskThing(),
  cpuThing()
])
  .then([networkResult, diskResult, cpuResult] => {
    console.log(networkResult)
    console.log(diskResult)
    console.log(cpuResult)
  })
```

See *segment-4/example-04.js*

Array#map and Promise.all: A Powerful Combo

returns array of promises...just what Promise.all needs!

```
Promise.all(cbFunctions.map(f =>
  new Promise((resolve, reject) => {
    f((err, data) => err ? reject(err) : resolve(data))
  })
))
```

converts callback-style function to promise-returning function

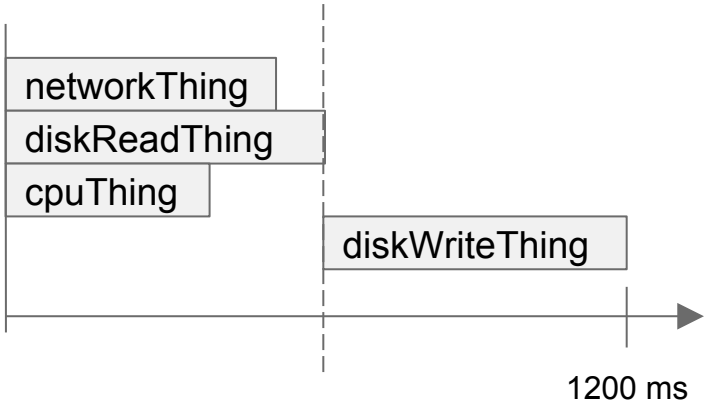
See *segment-4/example-05.js*

What about dependencies?

first this, executed in parallel

```
Promise.all([networkThing(), diskReadThing(), cpuThing()])
  .then(diskWriteThing)
```

then this!



What about generic serial execution?

(Sometimes called *waterfall* execution)

```
function serial(promises) {
  const result = promises.reduce((a, p, idx) => {
    a.last = a.last.then(p).then(r => a.results[idx] = r)
    return a
  }, { last: Promise.resolve(), results: [] })
  return result.last.then(() => result.results)
}
```

See *segment-4/example-07.js*

async/await: Async Performance, Sync Semantics

```
async function go() {
  const results = await Promise.all([
    networkThing(),
    diskRead(),
    cpuThing()
  ])
  await diskWrite(results.join('|'))
}

go()
```

See *segment-4/example-08.js*

async/await: Async Performance, Sync Semantics

```
async function go() {
  try {
    const results = await Promise.all([
      networkThing(),
      diskRead(),
      cpuThing()
    ])
    await diskWrite(results.join('|'))
  } catch(err) {
    console.log(`handling error: ${err.message}`)
  }
}
```

See *segment-4/example-10.js*

Exercises

exercise-01.js: take a promise, return a promise

exercise-02.js: take two promises, return a promise

exercise-03.js: take an array of promises, return a promise

Practical JavaScript

Ethan Brown

CommonJS & ESM

CommonJS

- "Node-style" module specification
- Dynamic resolution
- module.exports to export/expose functionality
- require to import functionality
- Not going anywhere anytime soon (?)

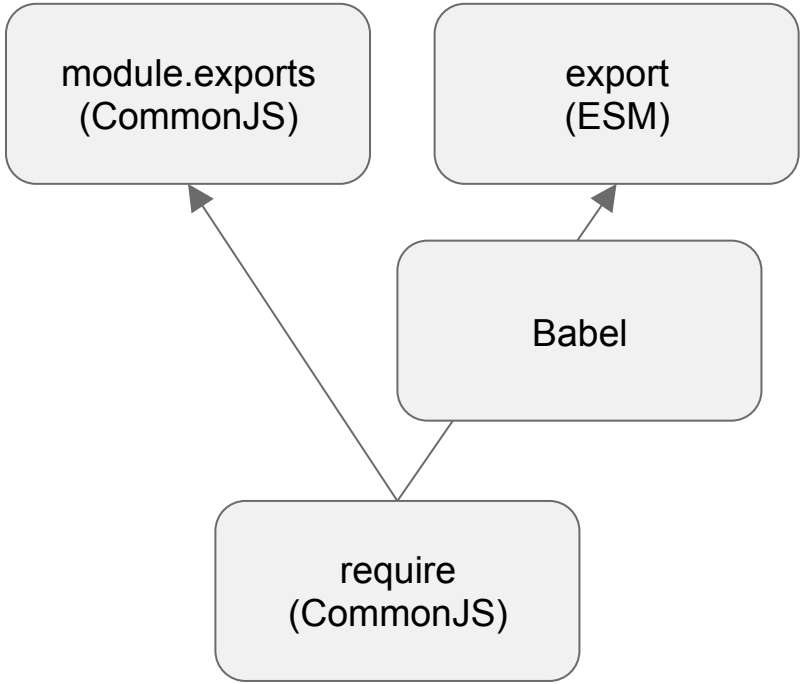
ECMAScript Modules (ESM)

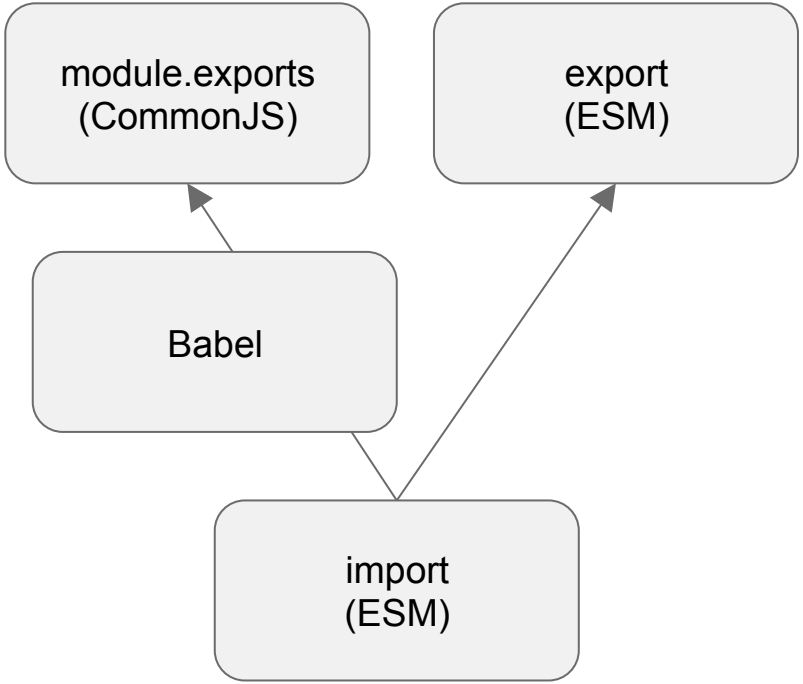
- "Official" module syntax proposed by TC-39
- Not as entrenched as CommonJS, but catching up fast
- More common in front-end use
- Static resolution
- The future

Which should I use? CommonJS or ESM?

BOTH, OF COURSE.

CommonJS & ESM: How to use Both?





CommonJS: Caching

module:

```
let counter = 0

module.exports = {
  getCount() { return counter },
  increment() { counter++ },
}
```

client:

```
const counter1 = require('./module.js')
const counter2 = require('./module.js')

counter1.increment()
counter2.increment()

console.log('counter1: ',
  counter1.getCount())
console.log('counter2: ',
  counter2.getCount())
```

See *segment-5/example-01*

CommonJS: Exporting Functions (Common)

module:

```
module.exports = function() {
  let counter = 0
  return {
    getCount() { return counter },
    increment() { counter++ },
  }
}
```

client:

```
const counter1 = require('./module.js')
const counter2 = require('./module.js')

counter1.increment()
counter2.increment()

console.log('counter1: ',
  counter1.getCount())
console.log('counter2: ',
  counter2.getCount())
```

See *segment-5/example-02*

ESM: Caching

module:

```
let counter = 0

export default {
  getCount() { return counter },
  increment() { counter++ },
}
```

client:

```
const counter1 = require('./module.js')
const counter2 = require('./module.js')

counter1.increment()
counter2.increment()

console.log('counter1: ',
  counter1.getCount())
console.log('counter2: ',
  counter2.getCount())
```

See *segment-5/example-04.*

ESM: Namespacing

module:

```
export function hello() {
  console.log('hello!')
}

export function goodbye() {
  console.log('goodbye!')
}
```

client:

```
import { hello, goodbye }
  from './module.js'

import { hello as hola, goodbye as adios }
  from './module.js'

hello()
goodbye()

hola()
adios()
```

See *segment-5/example-04*

npm: What is it?

- Package manager (but npm does not stand for "node package manager")
- Not just for JavaScript: can package anything
- Modules stored in node_modules; this is npm's domain
- Now supports org-scoped modules (hooray!)

Rolling your own npm package

```
{
  "name": "virtual-golem",
  "description": "Who doesn't want a clay automaton who will do your bidding?",
  "author": "Ethan Brown <e@zepln.com>",
  "contributors": [
    "Ethan Brown <e@zepln.com>"
  ],
  "main": "dist/Golem.js",
  "version": "1.0.0",
  "keywords": [
    "golem",
    "example"
  ],
  /*...to be continued */
```

The name that will be used when you npm install

The file require / import looks in!

Use npm version command for version management / incrementing

Rolling your own npm package

```
  /*...continued */
  "license": "(MIT OR GPL-3.0)",
  "scripts": {
    "test": "jest",
    "dist": "babel src --out-dir dist --source-maps"
  },
  "dependencies": {
    "babel-cli": "^6.24.1",
    "babel-preset-env": "^1.5.1"
  },
  "devDependencies": {
    "jest": "^20.0.4"
  }
}
```

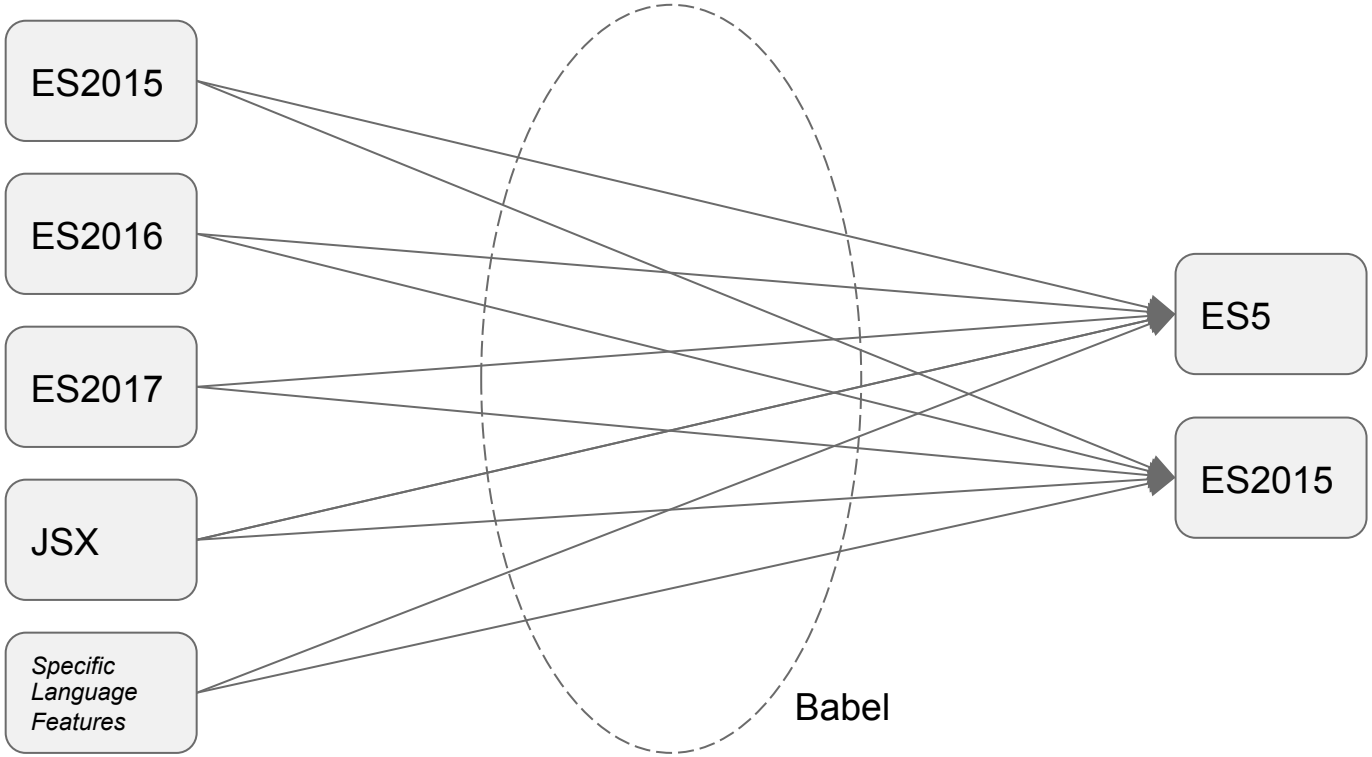
My favorite open source license combo!

Recommended!

Practical JavaScript

Ethan Brown

What is Babel?



Why/When We Need Babel

- When we are targeting older browsers
- When we want to use the latest language features
- When we want to use JSX

Using Babel: Presets & Plugins

- Plugins: specific transformations (example: "es2015-arrow-functions")
- Presets: "collections| of plugins; most often aligned with language releases (ES2015, ES2016, etc.)
- "env" preset: automatically determines best plugins based on what environments you want to target. Replaces "latest".

Using Babel: The .babelrc File

- Determines which plugins/presets are being used
- Uses JSON5!
- Install Babel + presets first (example: "npm install -D babel-cli babel-presets-env")

.babelrc: short & sweet:

```
{ presets: ["env"] }
```

Using Babel: babel-polyfill

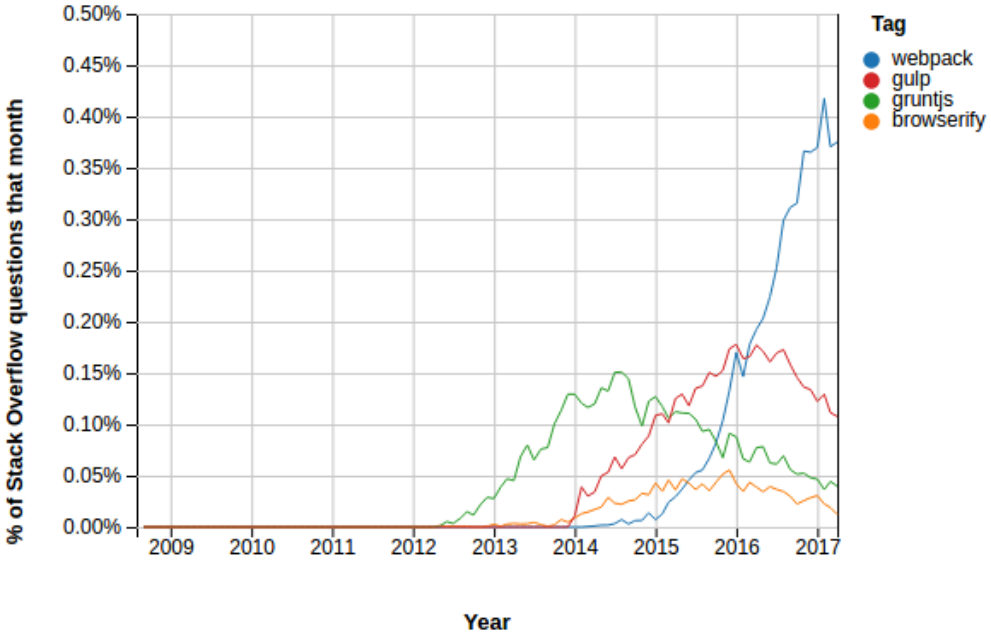
- Needed when using promises and targeting ES5 (among other things)
- Install: `npm install babel-polyfill`
- Add to the top of your JavaScript files: `require('babel-polyfill')`

Why Webpack?

- Helps with dependency management
- Improves performance
- Fast becoming ubiquitous
- Highly configurable
- Offers unique approaches to handling non-JavaScript assets (CSS, images, etc.)
- De facto choice for React apps
- The usual front-end needs: minifying, reducing HTTP requests

Webpack: Alternatives

- Browserify
- Grunt
- Gulp



Source: Stack Overflow Trends

Using Webpack: webpack.config.js

- First, install webpack: npm install -D webpack
- To run webpack, add "build": "webpack" to "scripts" section of package.json and run "npm build"
- (Or, run ./node_moduules/.bin/webpack)

```
const path = require('path')

module.exports = {
  entry: './path/to/my/entry/file.js'
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  }
}
```

Using Webpack: devtool, bundle hashes

```
const path = require('path')

module.exports = {
  devtool: 'eval-source-map',
  entry: './path/to/my/entry/file.js'
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.[chunkhash].js',
  }
}
```

Important for debugging

Useful for cachebusting

Webpack: module rules

- Install babel core/loader: npm install -D babel-core babel-loader

```
module.exports = {
  //…
  module: {
    rules: [
      {
        test: /\.jsx?$/,
        loader: 'babel-loader',
      },
    ],
  },
}
```

What type of files to process

What processor (loader) to use

Using Webpack: SCSS/SASS/CSS

- Install sass-loader: npm install -D sass-loader node-sass

```
module.exports = {
  //…
  module: {
    rules: [
      {
        test: /\.s?css$/,
        use: ['style-loader', 'css-loader', 'sass-loader']
      },
    ],
  },
}
```

Loaders applied right to left

Summary

- Babel: lets you use new language features (or dialects) while maintaining compatibility
- Webpack: high-performance, ubiquitous choice for web development
- Babel & Webpack work together
- Webpack reduces HTTP requests (improving performance) by generating (large) bundles...browser caching is key!
- You probably want at least two bundles: a bundle for your app and a bundle for vendor code