

- Day 1
- Day 2
- Day 3

- Introduction
- What's new since ES5?
- Promises
- `async/await`
- Transpilers
- Modules
- Unit tests
- The layers of testing
- Odds and ends

- Typescript
- Interfaces
- Union Types
- Classes

Hello

?



Rakesh Pai

@rakesh314

Developer

Since 2002

{errorception}

500+ customers

10,000+ concurrent errors

Node.js + MongoDB

{errorception}

the guardian

imgur



cleartrip



PAGERDUTY

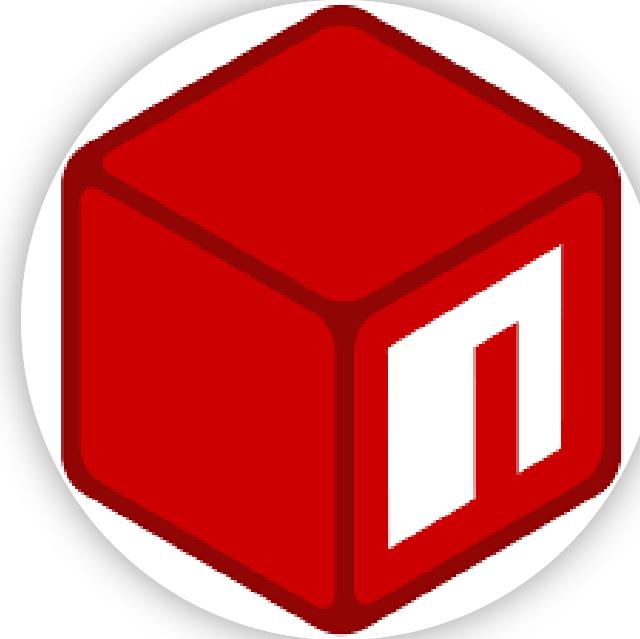




Frequent speaker
Ex - program committee



34 repos



18 packages



Name

- Role
- 2-line summary of your experience
- Expectations from this workshop



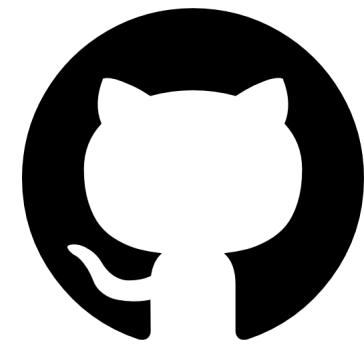
Rectangle

- Implement a geometric rectangle
- Rectangles have area and perimeter
- How can you demonstrate what your code does?

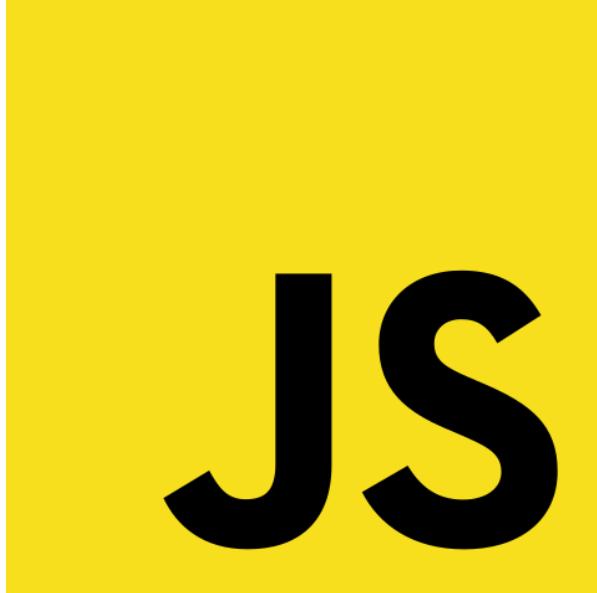


Square

- Keep rectangle behaviour as-is
- Implement a square







JS

JavaScript



ES

ECMAScript



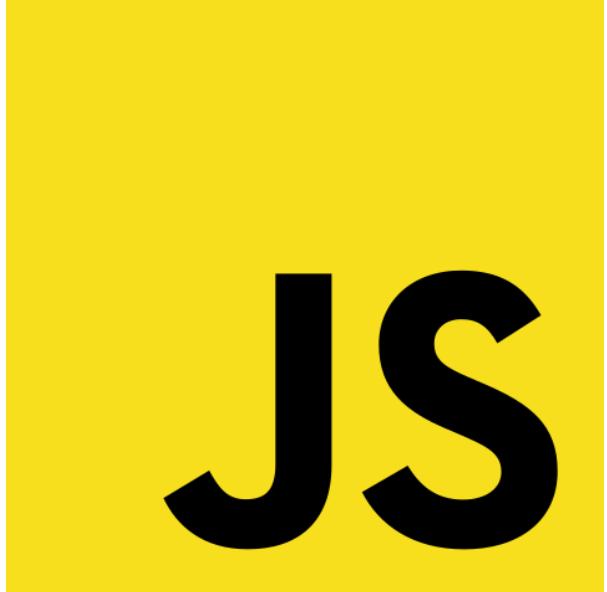
ES

- ES1 - 1997
- ES2 - 1998
- ES3 - 1999
- ES4 - abandoned
- ES5 - 2009
- ES6 - 2015
- ES7 - 2016
- ...



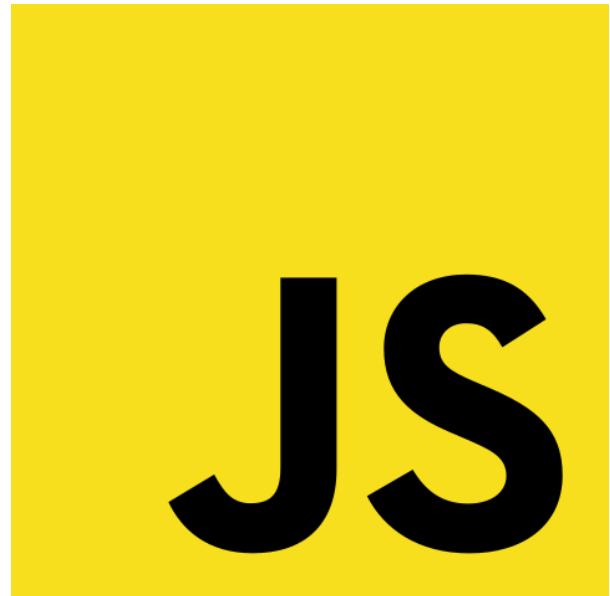
ES

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



JS

What's new?



- Variable declaration
- Template literals
- Arrow functions
- Destructuring
- Object/array rest/spread
- Function arguments rest/spread, default values

```
var x = 'outside';
function foo() {
  var x = 'inside';
  console.log(x);
}

foo(); // output: ?
console.log(x); // output: ?
```

```
function foo() {  
  if(/* some true condition */) {  
    var x = 'inside if';  
  }  
  
  // outside if:  
  console.log(x);  
}  
  
foo(); // output: ?
```

`var` is function scoped

```
function foo() {  
  if(/* some true condition */) {  
    let x = 'inside if';  
  }  
  
  // outside if:  
  console.log(x);  
}  
  
foo(); // output: ?
```

let & const are block scoped

Never use var!

- const prevents reassignment
- let allows reassignment

```
let x = 0;  
x = 100; // allowed
```

//--

```
const y = 0;  
y = 100; // error
```

Always prefer const

Use let to indicate reassignment...

```
let x = ''; // it's going to change!
```

...but you may want to rethink your approach
and use const instead



Watch out!

const doesn't mean immutable!

```
const person = { name: 'Alice' };  
person.age = 42; // This is not an error!
```

```
const arr = [1, 2];  
arr.push(3); // This is not an error!
```

Template literals

```
function greet(name) {  
  return `Hello ${name}!`;  
}  
  
console.log( greet('world') ); // output: 'Hello world!'
```

...with variable interpolation!

Template literals

```
// Expressions allowed
console.log(`Currently logged ${isLoggedIn() ? 'in' : 'out'}`); // valid
```

```
// Statements not allowed
console.log(`Currently logged ${if(...) { ... }}`); // error
```

Template literals

```
// Multiline!
const x = `

I can span
multiple
lines
with ${interpolation}
`;
```



Arrow functions

Arrow functions

```
// Without arrow functions:  
function double(num) {  
    return num * 2;  
}
```

```
// With arrow functions:  
const double = num => num * 2;
```

Arrow functions

```
// Single argument  
const double = num => num * 2;
```

```
// Multiple arguments  
const add = (a, b) => a + b;
```

Arrow functions

```
// Single line
const double = num => num * 2; // Implicit return

// Multiple lines
const addTwo = num => {
  return num + 2; // Needs explicit return
};
```

Always anonymous
Need to assign if you want reference

```
const double = num => num * 2;
```

Arrow functions

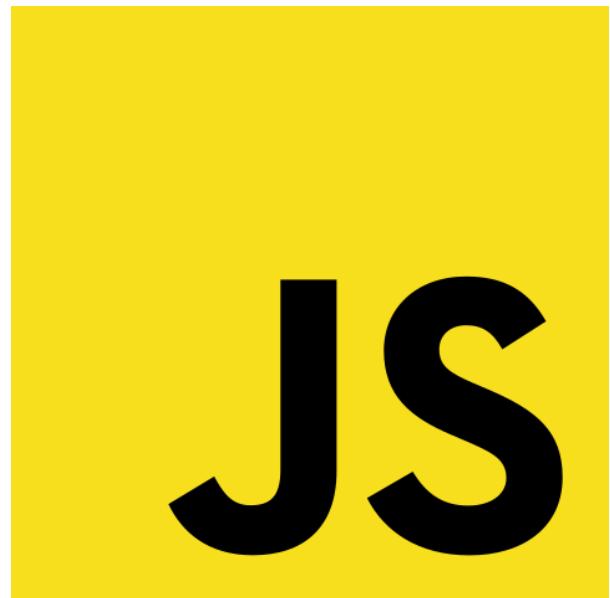
- this is lexical
- arguments is not accessible



Arrow functions

- Reduced keyword noise
- No surprises
- Clear, concise syntax

Always use arrow functions



- Variable declaration
- Template literals
- Arrow functions
- Destructuring
- Object/array rest/spread
- Function arguments rest/spread, default values

Destructuring objects

```
const person = { name: 'Alice' };
```

```
// Previously:
```

```
const name = person.name;
```

```
// With destructuring:
```

```
const { name } = person;
```

```
console.log(name); // Alice
```

Destructuring objects

```
const getUser = () => ({ name: 'Alice', age: 42 });

const { name, age } = getUser();

console.log(name); // output: 'Alice'
console.log(age); // output: 42
```

Destructuring objects

```
const greet = ({ name }) => `Hello ${name}!`;  
  
console.log( greet({ name: 'world' } ) ); // output: Hello world!
```

Destructuring objects with ...rest

```
const person = { name: 'Alice', age: 42 };

const { name, ...remaining } = person;

console.log(name); // Output: 'Alice'
console.log(remaining); // Output: { age: 42 }
```

Destructuring objects with ...rest

```
const display = ({ name, ...rest }) => {
  console.log(name); // output: 'Alice'
  console.log(rest); // output: { age: 42 }
};

const person = { name: 'Alice', age: 42 };

display(person);
```

Destructuring objects

```
const person = { name: 'Alice' };  
  
const { name: n } = person;  
  
console.log(n); // output: 'Alice'  
console.log(name); // undefined
```

Destructuring arrays

```
const arr = [1, 2, 3];

const [ first, second ] = arr;

console.log(first); // output: 1
console.log(second); // output: 2
```

Destructuring arrays

```
const getValues = () => ([ 1, 2 ]);  
  
const [ first, second ] = getValues();  
  
console.log(first); // output: 1  
console.log(second); // output: 2
```

Destructuring arrays with ...rest

```
const arr = [1, 2, 3];

const [ first, ...remainder ] = arr;

console.log(first); // output: 1
console.log(remainder); // output: [2, 3]
```

Destructuring arrays

```
const getFirst = ([ first ]) => first;  
  
console.log( getFirst([1, 2, 3]) ); // output: 1
```

Destructuring arrays

```
const x = [1, 2, 3];  
  
const [ one, , three ] = x;  
  
console.log(one); output: 1  
console.log(three); output: 3
```

Rest arguments

```
const display = (first, ...rest) => {  
  console.log(first); // output: 1  
  console.log(rest); // output: [2, 3]  
};  
  
display(1, 2, 3);
```

Rest arguments

```
const sum = (...nums) => {
  console.log(nums); // output: [1, 2, 3]
};

sum(1, 2, 3);
```

Object spread

```
const alice = { name: 'Alice' };  
  
const person = { ...alice };  
  
console.log(person); // output: { name: 'Alice' }  
console.log(person === alice); // output: false
```

Object spread

```
const alice = { name: 'Alice' };  
  
const person = { ...alice, age: 42 };  
  
console.log(person); // output: { name: 'Alice', age: 42 }
```

Array spread

```
const a1 = [1, 2];
const a2 = [3];

console.log([ ...a1, ...a2 ]); // output: [1, 2, 3]
```

Default values

```
const person = {};  
  
const { name = 'Alice' } = person;  
  
console.log(name); // output: 'Alice'
```

Default values

```
const person = { name: 'Bob' };  
  
const { name = 'Alice' } = person;  
  
console.log(name); // output: 'Bob'
```

Default values

```
const logName = (name = 'Alice') => {  
  console.log(name);  
};
```

```
logName(); // output 'Alice'  
logName('Bob'); // output 'Bob'
```

Default values

```
const makeRequest = ({ url, method = 'get', ...options }) => {  
  console.log(method);  
};  
  
makeRequest({ url: 'http://google.com/' }); // output: 'get'  
makeRequest({ url: 'http://google.com/', method: 'post' }); // output: 'post'
```

Property shorthands

```
const name = 'ALice';  
  
// Previously:  
const person = { name: name };  
  
// Now:  
const person = { name };
```

Property shorthands

```
const key = 'name';
const value = 'Alice';

// Previously:
const person = {};
person[key] = value;

// Now:
const person = { [key]: value };
```

Property shorthands

```
const computeKey = () => 'name';  
  
const person = { [computeKey()]: 'Alice' };
```



Lab

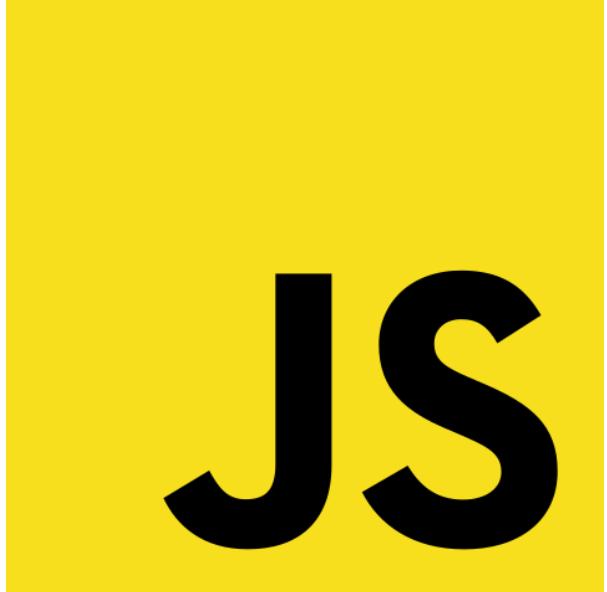
- <http://192.168.1.38:3000>
- day-1/1-es-basics
- Keep npm test running in a console
- Change index.js to use new syntax

Cheat sheet

```
// Variable declarations:  
// const > let > var  
  
// Arrow functions  
const inc = n => n + 1;  
  
// Template literals  
const greet = name => `Hello ${name}`;
```

```
// Object destructuring  
const x = { foo: 'bar', a: 'b' };  
const { foo, ...rest } = x;  
  
// Array destructuring  
const x = [1, 2, 3];  
const [ first, ...rest ] = x;
```





JS

Promises

Typical node.js async code

```
function readConfig(done) {  
  fs.readFile('/path/to/config', function(err, data) {  
    if(err) return done(err);  
  
    done(null, data);  
  }) ;  
}
```

Typical node.js async code

```
function readConfig(done) {  
  fs.readFile('/path/to/config', function(err, data) {  
    if(err) return done(err);  
  
    done(null, data);  
  }) ;  
}  
  
// With latest language features :)  
const readConfig = done => fs.readFile('/path/to/config', done);
```

Promises in real life

- An assurance that something will be done
- A promise can either be kept or broken
- When the promise is made, we can't tell if it will be kept or broken
- If the promise is kept, you usually expect something at the end
- If a promise is broken, you need to be informed about it



Provide an assurance

```
const getConfig = () => new Promise/* ... */;

getConfig(); // output: promise
```



Use the assurance

```
const getConfig = () => new Promise(/* ... */);
const getFilePath = config => /* Got config */;

getConfig().then(getFilePath);
```



Chain the assurance

```
const getConfig = () => new Promise(/* ... */);
const getFilePath = ({ filePath }) => filePath;
const readFile = path => /* Got file path */

getConfig()
  .then(getFilePath)
  .then(readFile);
```



Return assurances

```
const getConfig = () => new Promise(/* ... */);
const getFilePath = ({ filePath }) => filePath;

const getFilePathFromConfig = () =>
  getConfig().then(getFilePath);

const readFile = path => /* ... */
getFilePathFromConfig().then(readFile);
```



Chain multiple assurances

```
const getConfig = () => new Promise(/* ... */);
const getFilePath = ({ filePath }) => filePath;
const readFile = filePath => new Promise(/* ... */)

getConfig()
  .then(getFilePath)
  .then(readFile)
  .then(fileContents => /* got the file */);
```



Compare with callbacks

```
const getConfig = done => /* ... */;  
const getFilePath = ({ filePath }) => filePath;  
const readFile = (filePath, done) => /* ... */;  
  
getConfig(error, config) => {  
  if(error) throw error;  
  
  const filePath = getFilePath(config);  
  readFile(filePath, (error, fileContents) => {  
    if(error) throw error;
```



Where's the error handling?

- What are the best-practices for error handling?

Don't handle errors!

Don't handle errors!

(unless you want to provide a fallback)



Handling failure

```
const getConfig = () => new Promise(/* ... */);
const doSomething = config => /* Got config */;
const showError = error => /* Got error */;

getConfig()
  .then(doSomething)
  .catch(showError);
```



Handling failure

```
const getConfig = () => new Promise(/* ... */);
const getPath = ({ filePath }) => filePath;

const getPathFromConfig = () =>
  getConfig().then(getPath);

const readFile = path => /* ... */
  getPathFromConfig()
    .then(readFile)
    .catch(showError);
```

What should happen if you don't add a .catch?

```
(node:577) [DEP0018] DeprecationWarning: Unhandled promise rejections are  
deprecated. In the future, promise rejections that are not handled will  
terminate the Node.js process with a non-zero exit code.
```



Promises

```
const getConfig = () => new Promise(/* ... */);
const getFilePath = ({ filePath }) => filePath;
const readFile = filePath => new Promise(/* ... */)

getConfig()
  .then(getFilePath)
  .then(readFile)
  .then(fileContents => /* got the file */);
```



Creating promises

```
// Simple promise
const p = new Promise(
  resolve => setTimeout(resolve, 1000);
);

p.then(doSomething); // called after 1 second
```



Creating promises

```
// Promises with errors
const p = new Promise((resolve, reject) => {
  if /* successful */ {
    resolve(result);
  } else {
    reject(error);
  }
}) ;

p.then(doSomething); // called after 1 second
```



Callbacks to promises

```
const readFile = path =>
  new Promise((resolve, reject) => {
    fs.readFile(path, (err, data) => {
      if(err) {
        reject(err);
      } else {
        resolve(data);
      }
    })
  );
};
```



Node tries to help

```
const fs = require('fs');
const { promisify } = require('util');

const readFile = promisify(fs.readFile);
readFile(path).then(doSomething);
```



Promise gotchas

```
rejectedPromise
  .then(wontExecute)
  .catch(errorHandler)
  .then(willExecute); // This might be surprising
```



Promise gotchas

```
resolvedPromise
  .then(doSomething)      // let's say this throws
  .then(doSomethingElse) // this won't be called
  .catch(errorHandler)   // but this will be called
```



Parallel promises

```
const arrayOfPromises = [ p1, p2, p3 ];  
  
Promise.all(arrayOfPromises)  
  .then(([ r1, r2, r3 ]) => { /* ... */ }) ;
```



Lab

- day-1/2-promises
- Keep npm test running in a console
- Edit index.js to make tests pass

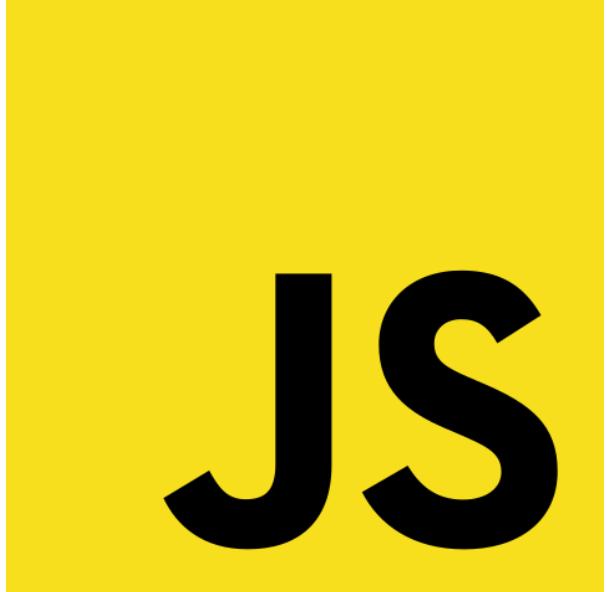
Cheat sheet

```
// Creating a promise
const p = new Promise(
  (resolve, reject) => { /* ... */ }
);

// Parallel promises
const ps = [p1, p2, p3];
const p = Promise.all(ps);
```

```
// Handling a promise
p.then(data => { /* ... */ });

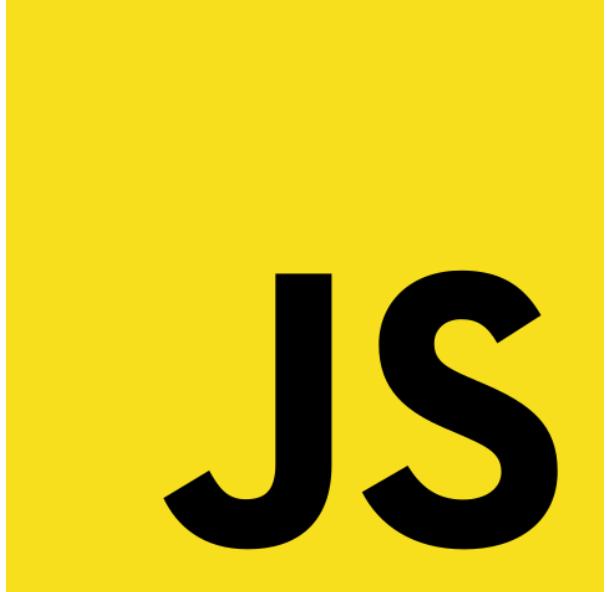
// Node's fs.readFile
readFile(path, 'utf8',
  (err, data) => {
    // ...
  }
);
```



JS

Promises

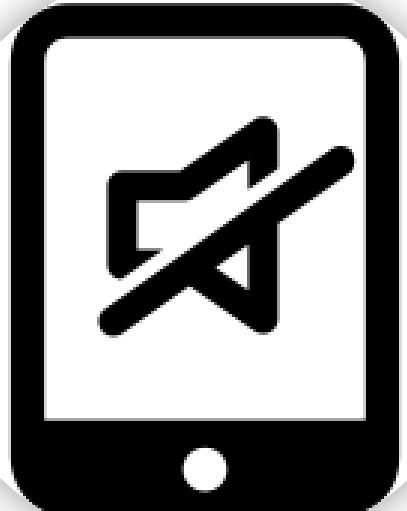




JS

async/await

You can only await in an async function



```
const asyncFunctionExample = async () => {  
    await someOtherFunction();  
};  
  
const anotherFunction = () => {  
    await someOtherFunction(); // Syntax error!  
};
```

async/await is syntax sugar
over promises

It's syntax sugar for promises

```
const readFile = path =>  
  new Promise(/* ... */);
```

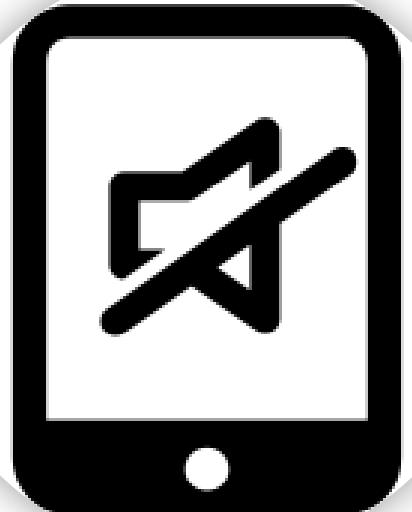
```
const printFile = path =>  
  readFile(path)  
    .then(console.log);
```

```
printFile('/path');
```

```
const readFile = path =>  
  new Promise(/* ... */);
```

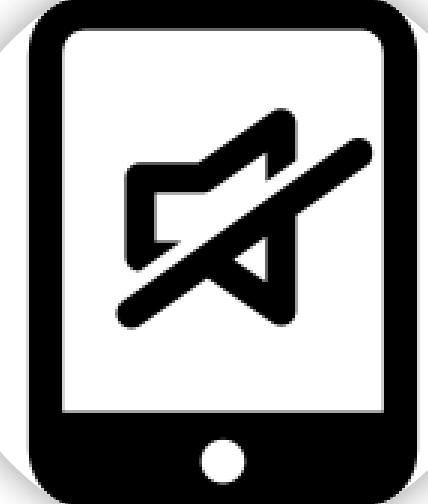
```
const printFile = async path => {  
  console.log(await readFile(path));  
};
```

```
printFile('/path');
```



Makes code look sync

```
const printFile = async () => {  
  const { filePath } = await getConfig();  
  console.log(await readFile(filePath));  
};  
  
printFile();
```

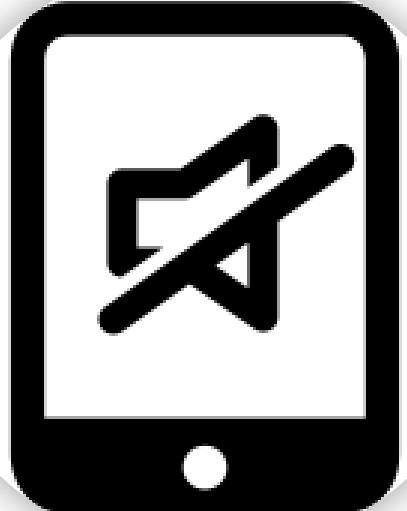


async functions always return a promise

```
const asyncIncrement = async x => x + 1;
```

```
// is equivalent to
```

```
const asyncIncrement = x => new Promise(  
  resolve => resolve(x + 1)  
);
```



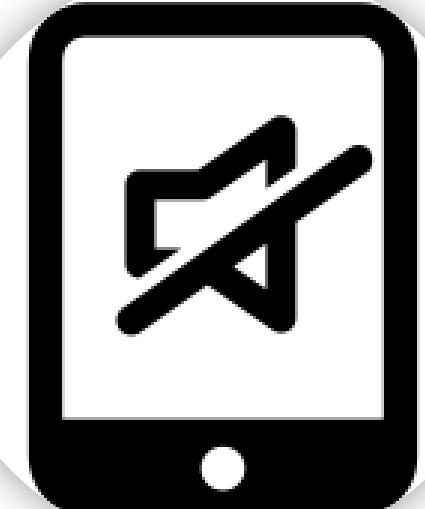
Don't need to await if returning a promise

```
const readFile = path => new Promise(/* ... */);

const getFileContents = async () => {
  const { filePath } = await getConfig();
  return readFile(filePath); // No need to await
};
```

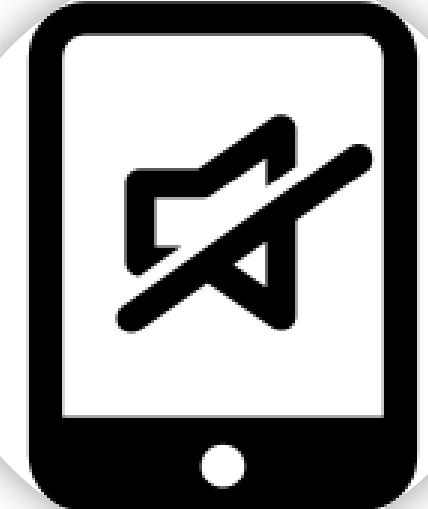


async/await
error handling



async/await error handling

```
const asyncFunction = async () => {
  try {
    console.log(await doSomething());
  } catch(e) {
    // ...
  }
}
```

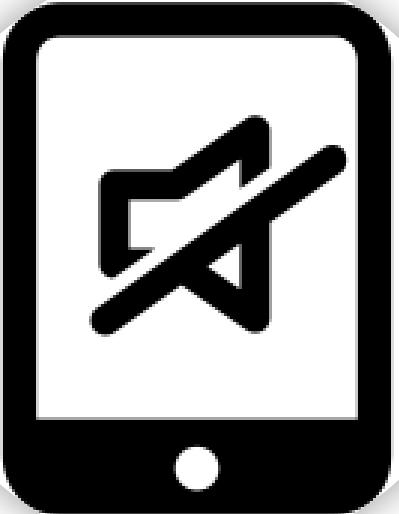


Watch out for performance traps

```
const getUserDetails = async userId => {
  const profile = await getProfile(userId);
  const comments = await getComments(userId);

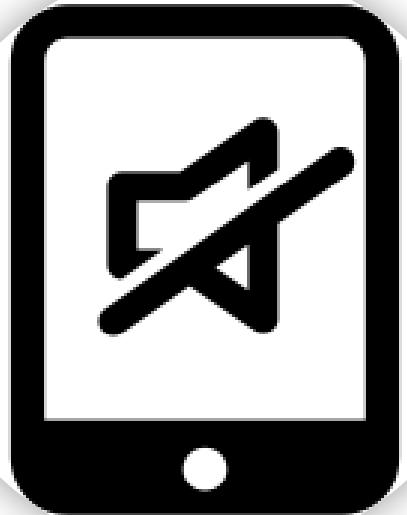
  return { profile, comments };
};
```

Making calls in parallel



```
const getUserDetails = async userId => {
  const [ profile, comments ] = await Promise.all([
    getProfile(userId),
    getComments(userId)
  ]);

  return { profile, comments };
};
```



There is no top-level await

```
const { doSomething } = require('./from/somewhere')
await doSomething(); // not allowed
```

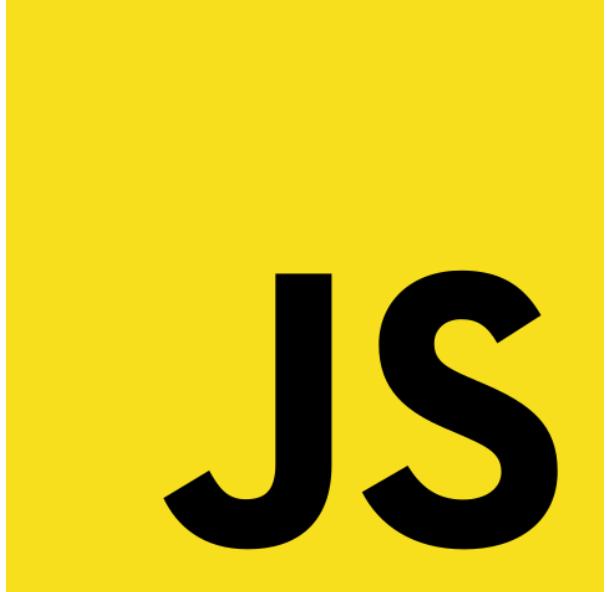


Lab

- day-1/2-promises
- Keep npm test running in a console
- Edit index.js to convert it to
async/await

Cheat sheet

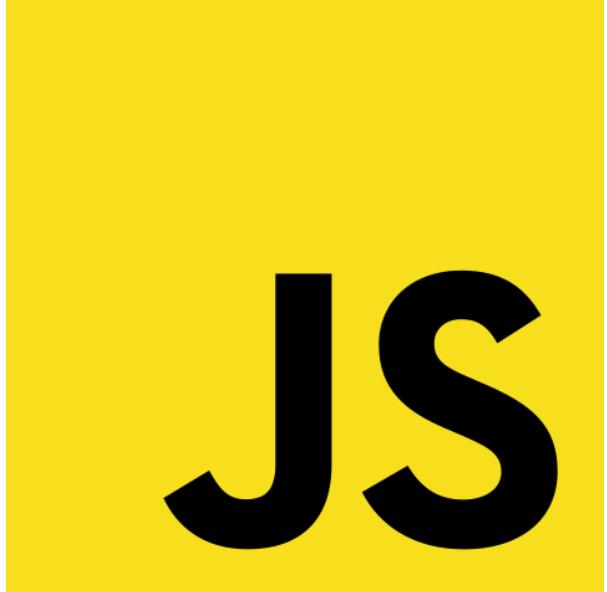
```
// Example async function
const printFile = async path => {
  console.log(await readFile(path));
};
```



JS

async/await

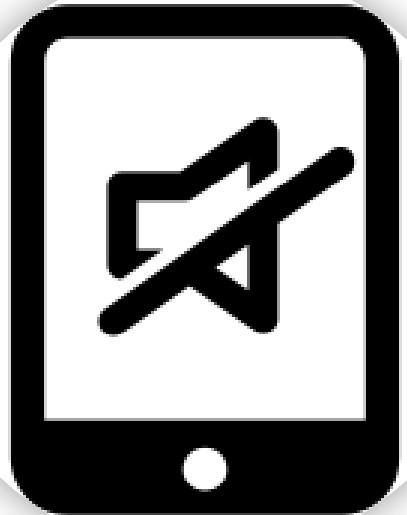




JS

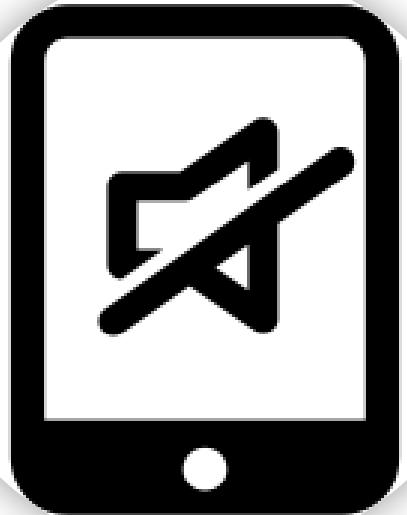
Transpilers

How do compilers work?



How do compilers work?

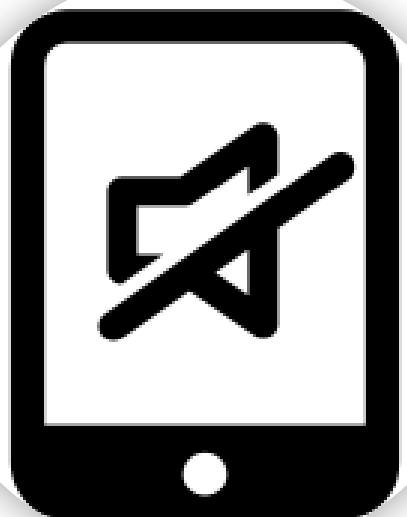
Step 1: Magic!



How do compilers work?

Step 1: Understand the code

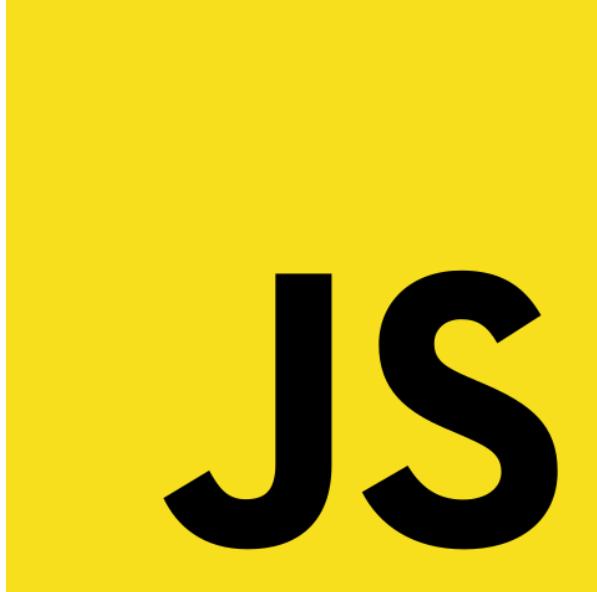
AST explorer



How do compilers work?

Step 1: Understand the code

Step 2: Convert it to something else



JS

Client-side JS

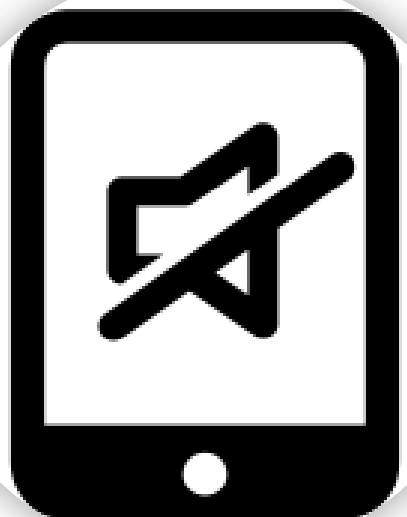
Minification

Minification

```
function foo() {  
    var i = 0;  
    function reallyLongFunctionName() { return 'Hello world!'; }  
    return reallyLongFunctionName();  
}
```

// can be converted to

```
function foo() { return 'Hello world!'; }
```



How do compilers work?

Step 1: Understand the code

Step 2: Convert it to something else

Minifiers are compilers!



CoffeeScript

2009



CoffeeScript

CoffeeScript

```
// CoffeeScript  
turnLightsOn() if lightSwitch is on
```

// gets converted to

```
// JavaScript  
if(lightSwitch === true) {  
  turnLightsOn();  
}
```



Transpiler

Source-to-source compiler

Meanwhile...

- Devs want to use new JS features
- But how do we deal with older browsers?

BABEL

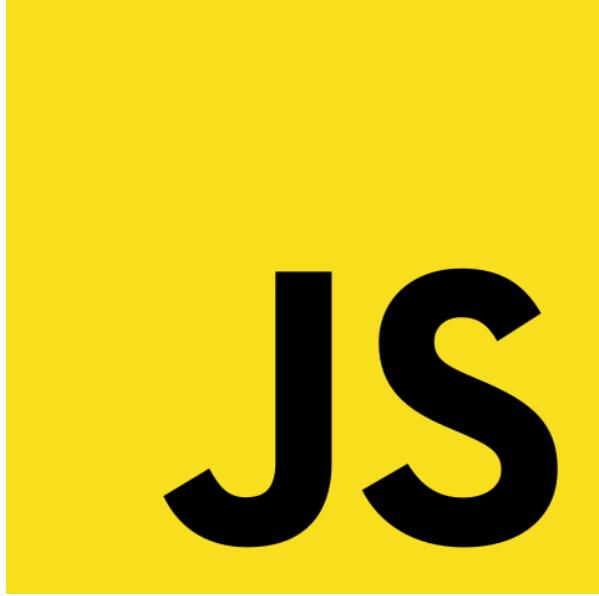
Babel REPL

Babel plugins

ESLint

ESLint plugins

Prettier

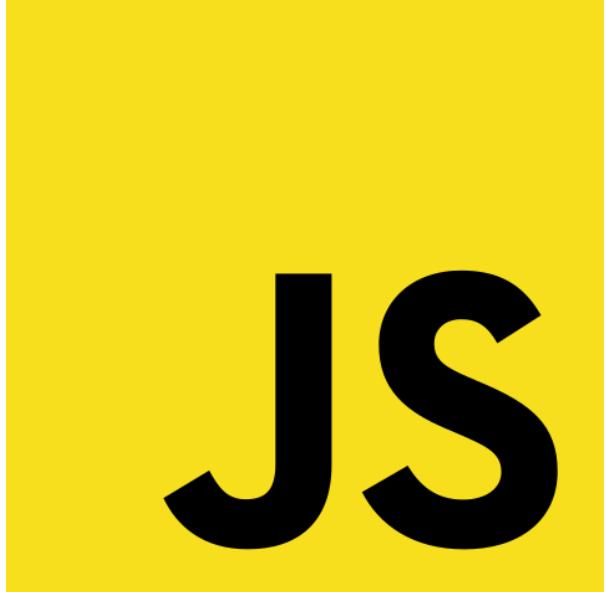
A large yellow square containing the letters "JS" in a bold, black, sans-serif font.

JS

Transpilers

- Minifiers
- Other language to JS
- Babel
- ESLint
- Prettier

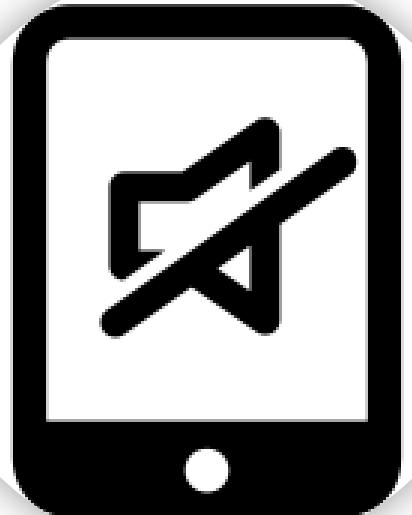


A large yellow square containing the letters "JS" in a bold, black, sans-serif font.

JS

Modules

- Browser
- Node.js
- Unforeseen environments

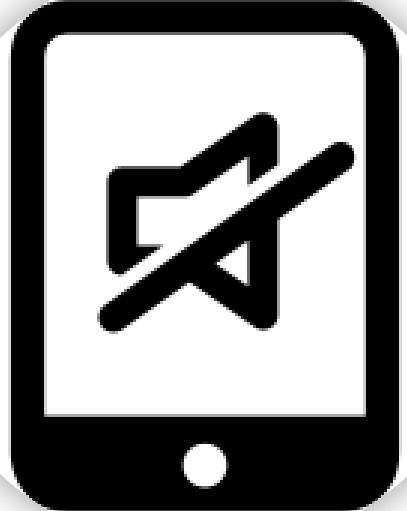


Node.js uses CommonJS

```
var fs = require('fs');
var myModule = require('./path/my-module');

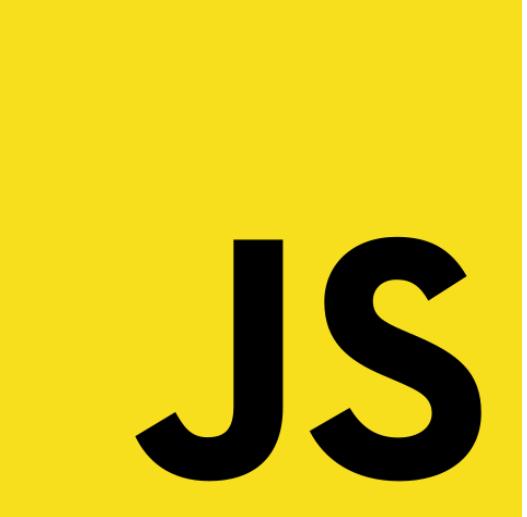
// ...

module.exports.myExport = /* ... */;
module.exports = /* ... */
```



Browsers?
AMD was somewhat popular

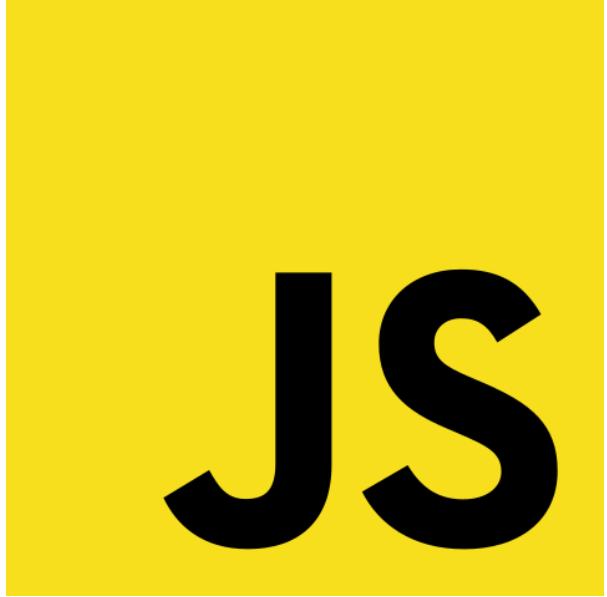
```
define('myModule', ['foo', 'bar'],
  function (foo, bar) {
    var myModule = /* ... */
      return myModule;
  }
);
```



JS

import/export

ES2015



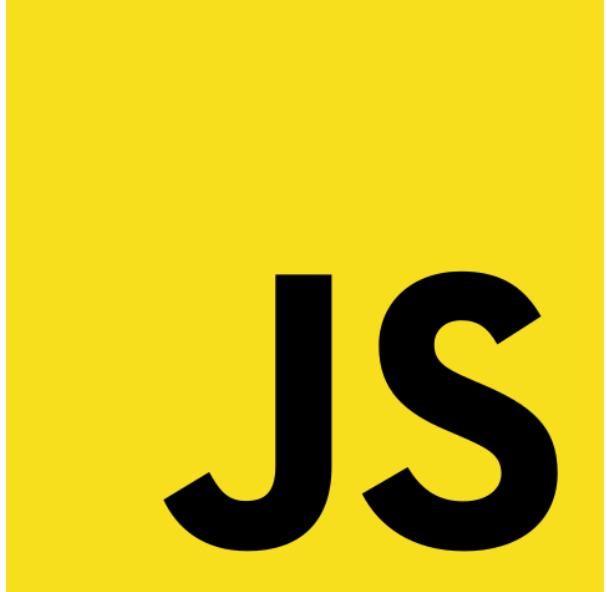
JS

Syntax

```
import fs from 'fs';
import { readFile } from 'fs';

// --

export const myFunction = () => { /*...*/ };
export default () => { /*...*/ };
```

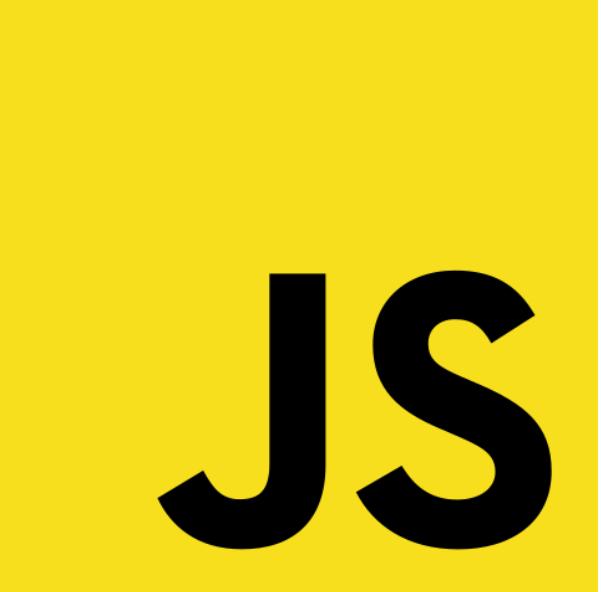
A large, bold, black "JS" logo is centered on a solid yellow background.

JS

Statically analysable

```
const moduleName = 'fs';
const fs = require(moduleName); // valid

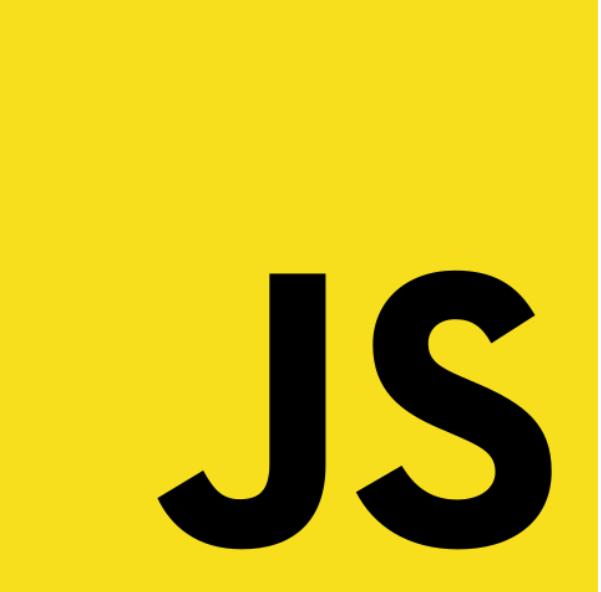
import fs from moduleName; // Syntax error!
```

A large, bold, black "JS" logo is centered on a solid yellow background.

JS

Default exports

```
export default () => {  
  // do something  
};  
  
// In another file  
import thatFunction from './path/to/module';
```

A large, bold, black "JS" logo is centered on a solid yellow background.

JS

Named exports

```
export const myFunc = () => {
  // do something
};

// In another file
import { myFunc } from './path/to/module';
```

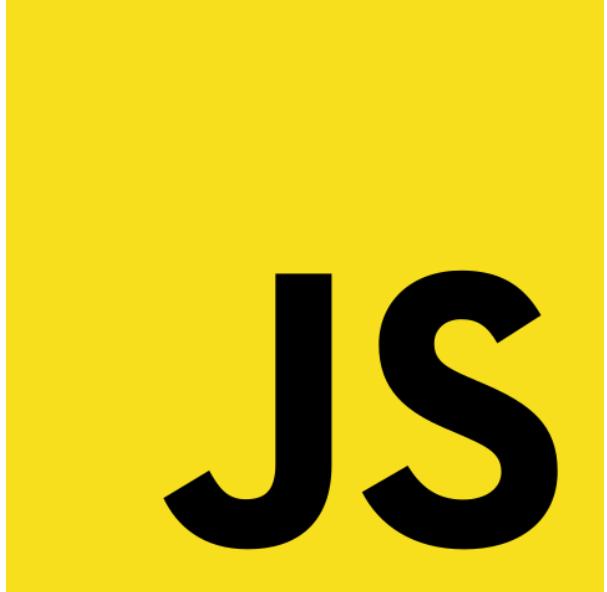
JS

Combining named and default

```
export const myFunc = () => {
  // do something
};

export default () => {
  // do something else
};

// In another file
import defaultFunc, { myFunc } from './path/to/module'
```



JS

Current status for import

- Browser support is improving
- No node support, but lots of activity

What should we do?

- We must follow language standards
- But we can't use the standards!
- We can use transpilers! **But should we?**

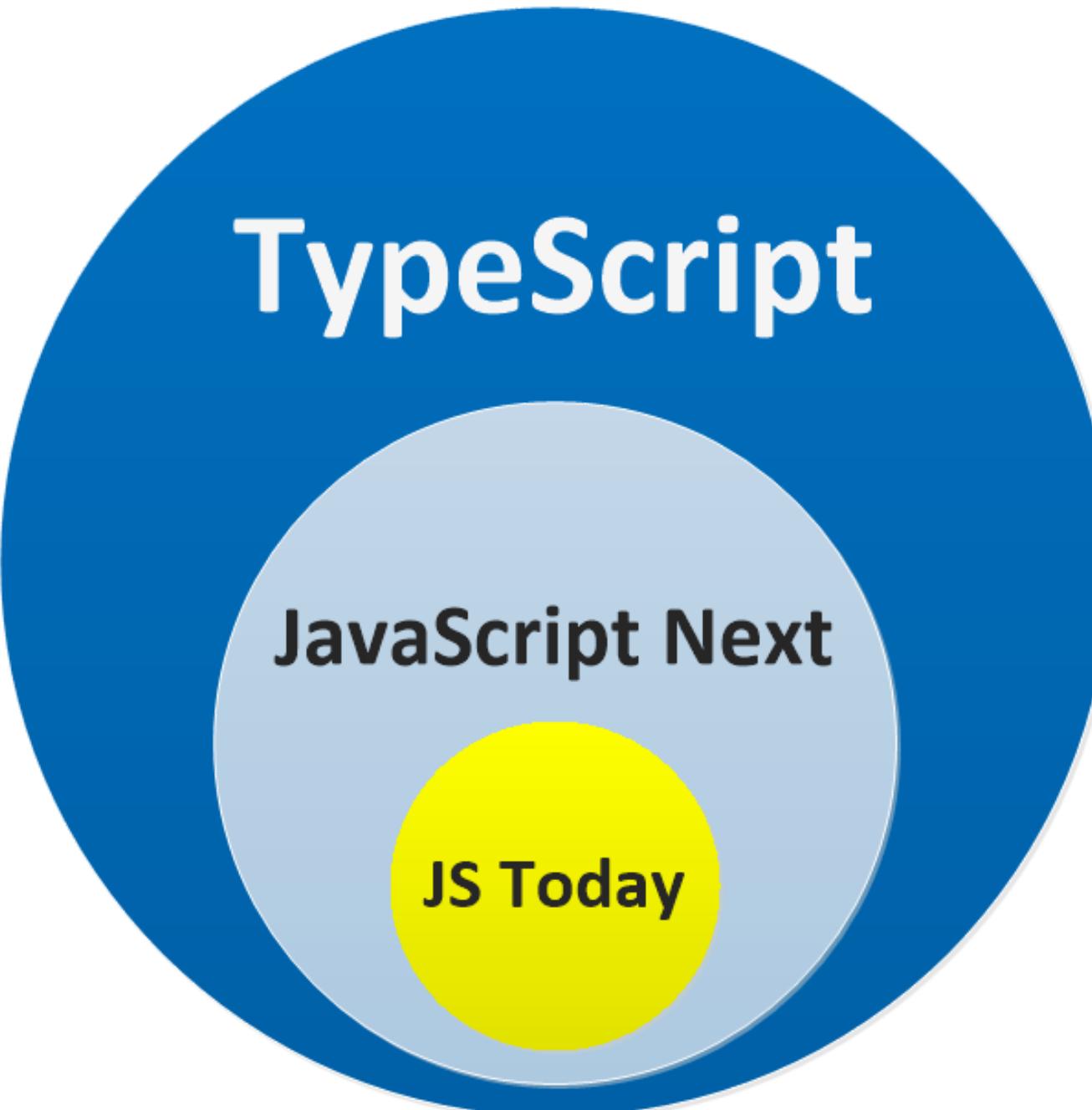
Should we wait for import support?

- If browser JS, don't wait, use a transpiler.
- Node.js is not so clear. Most people are waiting.

However...



TypeScript



TypeScript

JavaScript Next

JS Today

TypeScript supports import



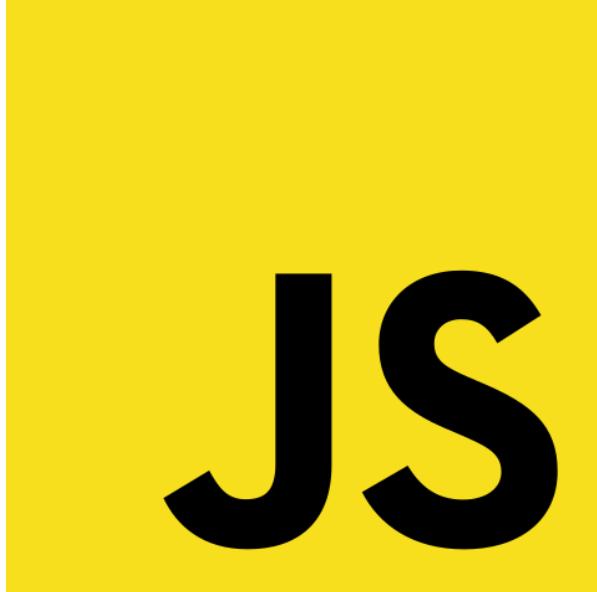
Lab

- day-1/3-import
- Keep npm test running in a console
- Convert src/*.ts to use import syntax

Cheat sheet

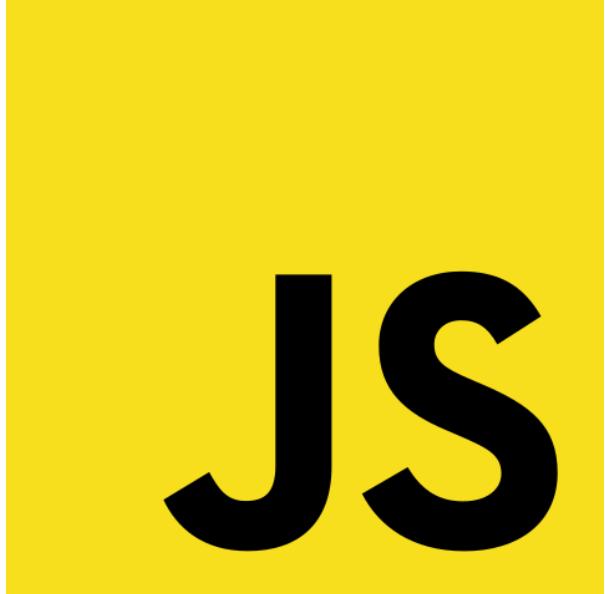
```
import { anExport } from './path';
import defaultThing from './path2';
```

```
export const anExport = /* ... */
export default /* ... */
```



JS

Modules

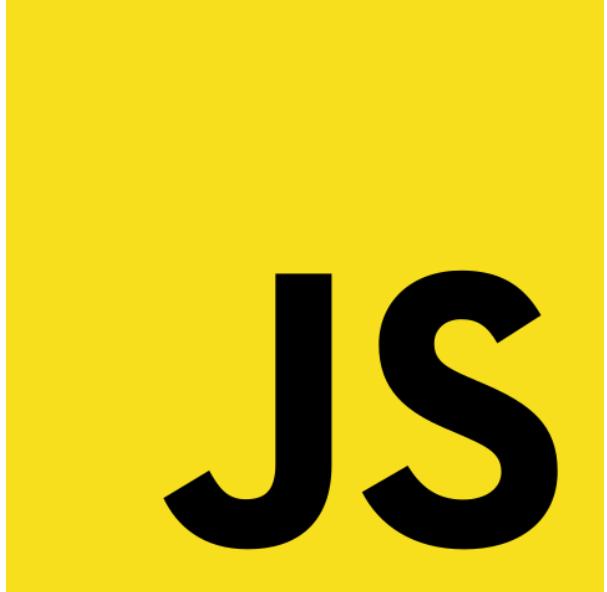
A large, bold, black "JS" logo is centered on a solid yellow background.

JS

Modules

```
// Instead of
import { add, subtract, ... } from 'maths-fns';

// you can do
import * as maths from 'maths-fns';
```



JS

Modules

```
import './path/to/file';
```





Unit tests

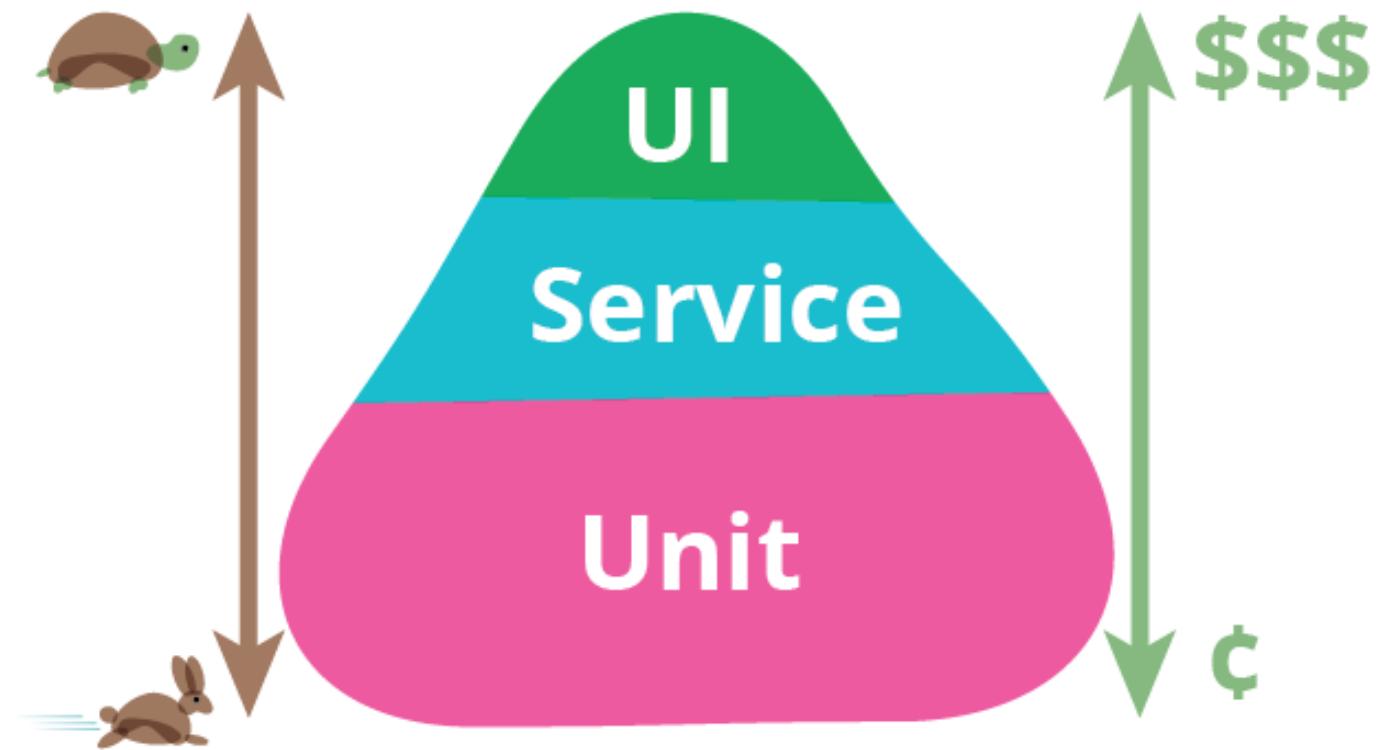
- You're already doing it
- Just make it reusable



Why automated tests?

- Increases confidence in the code
- Fearless refactoring
- Catch bugs early
- Deliver better software faster

The test pyramid





What should you test?

- Everything you write!
- In JS, this usually means every export should have tests
- You might consider exporting things just for testing
- You might change the design to make it more testable



What is a unit?

- The smallest testable part
- Only your code, not the network, or other external things
- In JS, a unit is usually a function

There are other kinds of tests

We'll look at them later



Unit testing frameworks

- There are several: Mocha, Qunit, Jasmine
- We're using Jest by Facebook
- Jest comes with batteries included



Writing a test

- *filename.spec.js*
- or __tests__/*.js



Writing a test

```
it('should add two numbers', () => {  
  // perform test here  
});
```



Writing a test

```
it('should add two numbers', () => {  
  expect( add(1, 2) ).toEqual(3);  
}) ;
```



Writing a test

```
it('should calculate interest correctly', () => {
  // setup
  const interest = calculateInterest(1000, 10);

  // assertions
  expect(interest).toEqual(100);
}) ;
```



Lab

- day-1/4-unit-tests
- Find further instructions in README .md
- Write code and tests



Try this

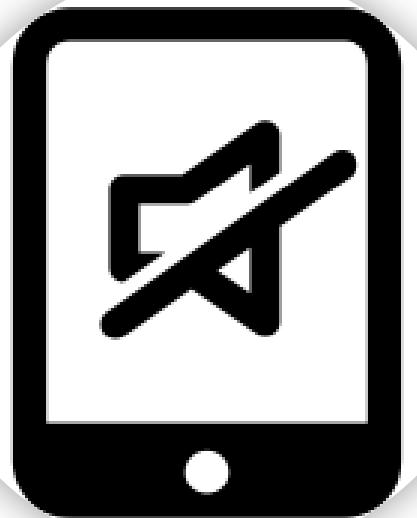
- Try writing the code first, then the test
- Try writing the test first, then the code

Cheat sheet

- Write a module to add, subtract, multiply and divide two numbers
- If any input is not a number, it should return null.
- You can use multiple it blocks and multiple assertions if you want
- Try writing tests first

```
it('should calculate interest', () => {
    // setup
    const interest = calcInterest(1000, 10);

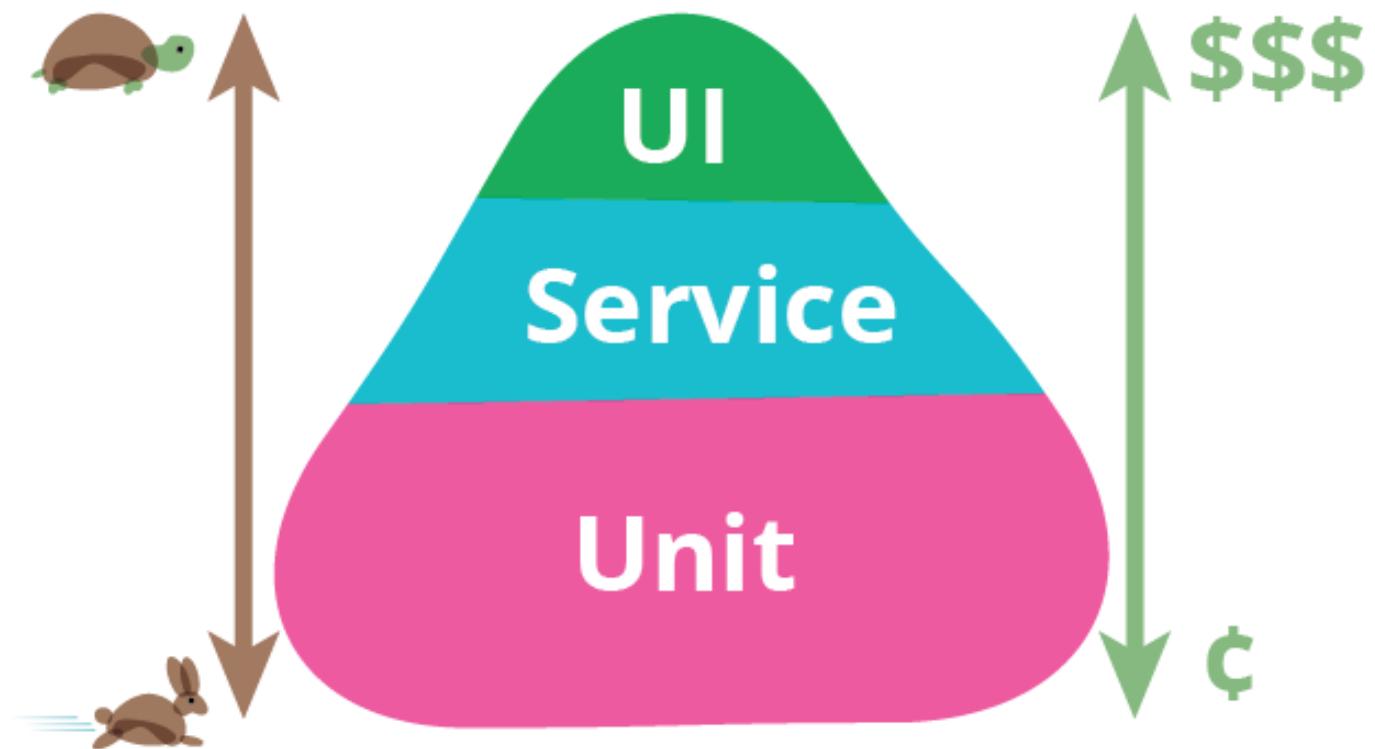
    // assertions
    expect(interest).toEqual(100);
}) ;
```



Unit tests



The layers of testing



Things we need to test

- Does our code do what's expected?
- Does our code interact correctly with the DB or API?
- Does the system as-a-whole behave correctly?
- Does the UI behave correctly?
- Does this match what our users want?



Async tests



Jest supports returning a promise

```
it('should test something', () => {
  return addAsync(1, 2)
    .then(value => expect(value).toEqual(3));
}) ;
```

But a function that returns a promise is...
an `async` function!



Jest with async/await

```
it('should test something', async () => {
  const value = await addAsync(1, 2);
  expect(value).toEqual(3);
});
```



Refactor lab 2's tests



How do we test something with an external dependency?

fs? DB? API?



How do we test something with an external dependency?

getUserById



How do we test something with an external dependency?

Start the database first?



Problems with using a DB

- The test machine needs to have a running DB
- Tests lose repeatability because of DB persistence
- Tests run slower



How about using a throwaway in-memory DB for the tests?

mockgoose



- A real DB is 'real', but brings its problems.
- A fake DB solves many problems, but may not give confidence.



Local DJ

- Get GPS update when location changes
- Call service with location to find song
- Play the song through a media player
- Queue up the song if something's playing
- Never repeat a song immediately
- Some locations won't have songs



What is code?

Questions

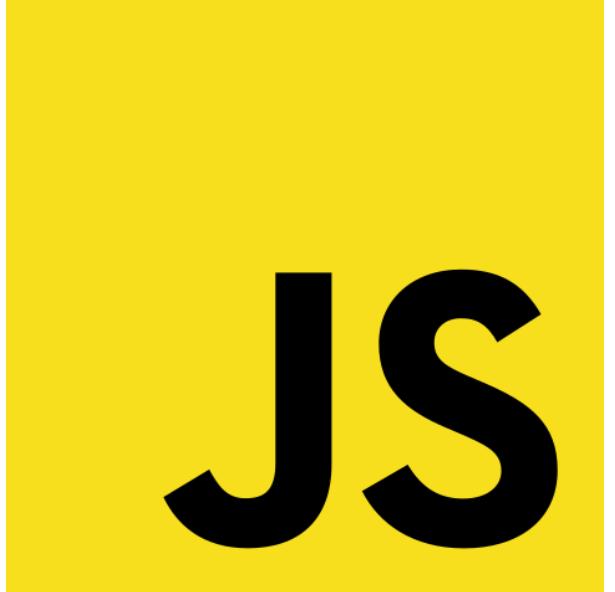
Please write down your top takeaway of the day.



TypeScript



Why types?

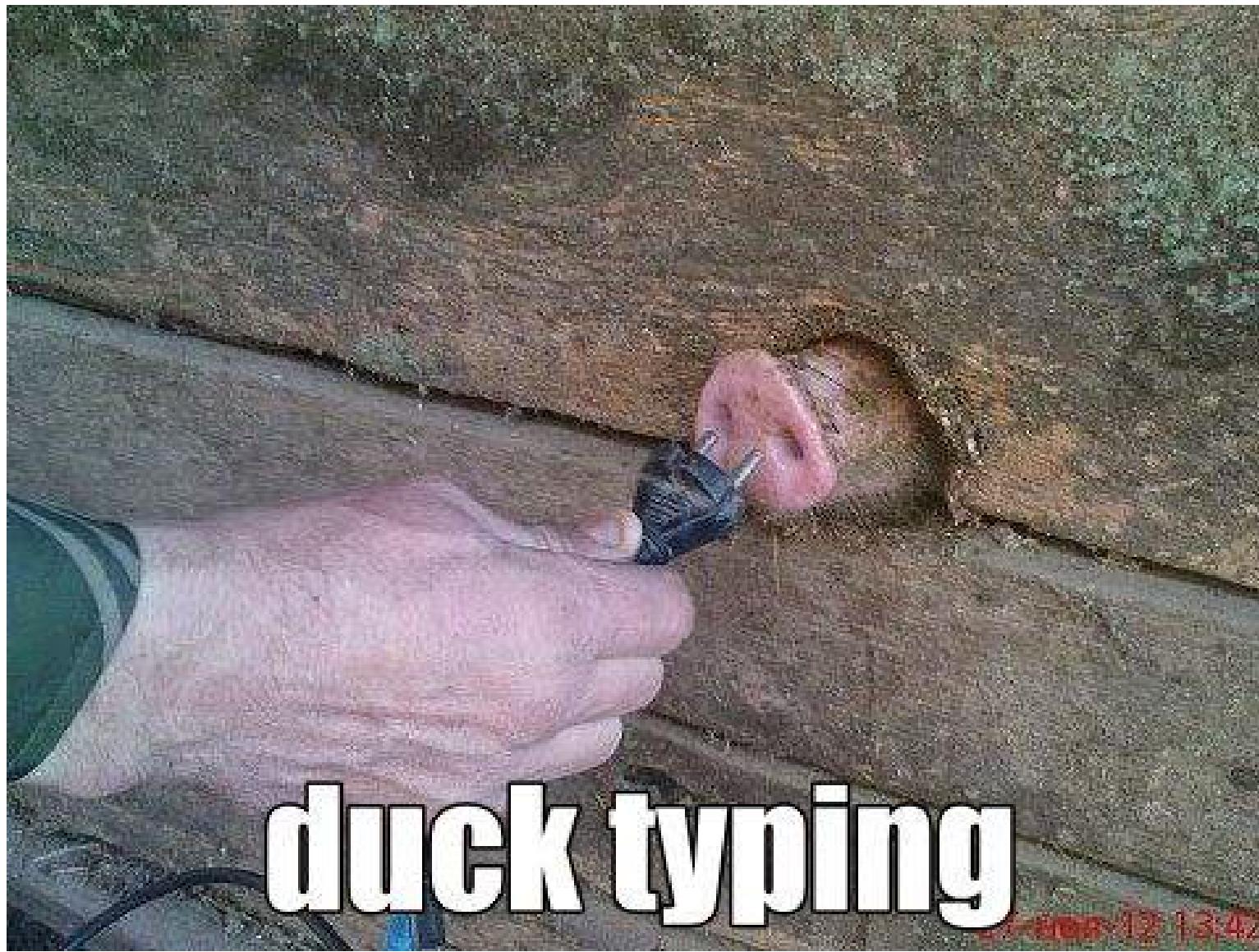
A large, bold, black "JS" logo is centered on a solid yellow background.

JS

JS already has types!

```
const a = "Hello world";
console.log(typeof a); // output: string

const b = 1; // number
const c = true; // boolean
const d = {} ; // object
```



duck typing

13:42



What value does TypeScript add?



Converting a JS file to TypeScript

Rename .js to .ts!



How it works

- You write JS + Types:
- The compiler checks if your types match correctly
- The compiler throws away type information, converts it to JS



How it works

- Types are only checked at build time
- No runtime type-checking!



const vs. let/var

```
const greeting = "Hello"; // Definitely always string

let greeting2 = "Hello"; // String right now
let greeting2 = 1; // Should this be allowed?
```



const vs. let/var

```
let greeting1;
greeting1 = "Hello";
greeting1 = 1; // valid TS
```

```
let greeting2 = "Hello";
let greeting2 = 1; // Error!
```



any

```
let greeting1;
```

```
// same as
```

```
let greeting1: any; // 'any' type
```



string

```
let greeting1 = "Hello";
```

// same as

```
let greeting1: string = "Hello";
```



number

```
let greeting1 = 1;
```

// same as

```
let greeting1: number = 1;
```



Omit types where inference can do its job

```
let a = "Hello";
// same as
let a: string = "Hello";

let a;
// same as
let a: any;
```



Arrays

```
const a = [1, 2, 3];
```

```
// same as
```

```
const a: number[] = [1, 2, 3];
```

```
// or
```

```
const Array<number> = [1, 2, 3];
```



Functions

```
const increment = (x: number) => x + 1;
```

// same as

```
const increment = (x: number): number => x + 1;
```

// same as

```
const increment: (x:number) => number = x => x + 1;
```



Extra arguments

```
const increment: (x: number): number = x => x + 1;  
  
increment(1, 2); // error
```



Too few arguments

```
const increment: (x:number) : number = x => x + 1;  
  
increment(); // error
```



Optional arguments

```
const increment = (x:number, by?:number) => {
    if(by === undefined) return x + 1;
    return x + by;
};

increment(1); // valid. by = 1
increment(1, 10); // valid. by = 10
increment(1, 2, 3); // error
```



Default arguments

```
const increment: (x:number, by?:number) => number =  
  (x, by = 1) => x + by;  
  
increment(10); // same as increment(10, 1);
```



Rest parameters

```
const sum = (...nums: number[]): number => {
  /* ... */
};

sum(1, 2, 3);
```



Array destructuring

```
const [ x, y ] = [ 1, 2 ];
```

```
// same as
```

```
const [ x, y ] : [number, number] = [ 1, 2 ];
```



Object destructuring

```
const { x, y } = { x: 1, y: 2 };
```

```
// same as
```

```
const { x, y }: { x: number, y: number } =  
  { x: 1, y: 2 };
```



Object destructuring

```
const { x: renamed, y } = { x: 1, y: 2 };

// same as

const { x: renamed, y }: { x: number, y: number }
  = { x: 1, y: 2 };
```



null and undefined

```
const u: undefined = undefined;  
const n: null = null;
```

Subtype of all other types by default,
but this can be changed



Lab

- day-2/1-ts-basics
- Ensure tests pass with `npm test`.
- Modify `tsconfig.json` as specified in `README.md`
- Add type signatures to make tests pass

Cheat sheet

```
// Type not always needed  
const x = "Hello";  
const y = 1;
```

```
// Destructuring  
const [a: number, b: number] = [ 1, 2 ];
```

```
// Arrays  
const num: number[] = [ 1, 2, 3 ];
```

```
// Functions
```



TypeScript



never type

```
const infiniteLoop = () : never => {
    while(true) { }
};
```



void type

```
const log = (thing): void => {  
    console.log(thing);  
};
```





Interfaces

Define the shape of your objects



Interfaces

```
interface NamedThing {  
    name: string;  
}  
  
const printName = (x: NamedThing) =>  
    console.log(x.name);  
  
const person = { name: 'Alice', age: 42 };  
printName(person);
```



Optional parameters

```
interface Person {  
    name: string,  
    age?: number  
};  
  
const printPerson = (person: Person) => {  
    if(person.age) {  
        return `${person.name} is ${person.age} yrs old.`;  
    } else {  
        return `This is ${person.name}.`;  
    }  
}  
  
printPerson({ name: 'Alice' })
```



Excess property checks

```
interface Person {  
    name?: string,  
}  
  
const printPerson = (person: Person) => // ...  
  
printPerson({ named: 'Alice' }); // Error!
```



Use interfaces to avoid shape checks

```
interface Person {  
    name: string,  
}  
  
const printPerson = (person: Person) => {  
    // The following line isn't required  
    if(typeof person.name !== 'string') throw ...;  
  
    ...  
}
```



Type Aliases

Similar to interfaces



Type Aliases

```
type NamedThing = {  
    name: string  
};  
  
const printName = (x: NamedThing) =>  
    console.log(x.name);  
  
const person = { name: 'Alice', age: 42 };  
printName(person);
```



Type Aliases with primitives

```
type Name = string;
```

```
type Person = {  
    name: Name  
};
```



Structural Typing (TS)
vs.
Nominal Typing (Java)



Structural Typing

```
type Person = {  
    name: string  
};  
  
let person: Person;  
person = { name: 'Alice' };
```



Nesting types

```
type Person = {  
    name: string  
};  
  
type Ride = {  
    commuters: Person[]  
};
```



Lab

- day-1/2-ts-objects
- Verify that tests pass.
- Change tsconfig.json as shown in README.md
- Change src/index.ts to use types.

Cheat sheet

Add types to make object shapes explicit

```
type Person = {  
    name: string,  
    age?: number  
};
```



Interfaces Types



You can specify function interfaces

```
interface PersonsName {  
  (p: Person): string  
}
```

// or

```
type PersonsName = (p: Person): string
```

The TS logo consists of the letters 'T' and 'S' in white, bold, sans-serif font, positioned on a solid blue square background.

TS

You can specify readonly properties

```
interface Person {  
  readonly name: string  
}
```

// or

```
type Person = {  
  readonly name: string  
}
```





Union Types



Union Types

```
const getName = personOrName => {
  if(typeof personOrName === 'string') {
    return personOrName;
  } else {
    return person.name;
  }
};

getName('Alice'); // output: 'Alice'
getName({ name: 'Alice' }); // output: 'Alice'
```



Union Types

```
const getName = (personOrName: any) => {
  if(typeof personOrName === 'string') {
    return personOrName;
  } else {
    return person.name;
  }
};

getName(true); // should be invalid, but isn't
```



Union Types

```
const getName = (personOrName: Person | string) =>
  if(typeof personOrName === 'string') {
    return personOrName;
  } else {
    return person.name;
  }
};

getName(true); // Compile error!
```



String Literal Types



String Literal Types

```
type LoadingState = 'loading' | 'success' | 'failure'

type Request = {
    state: LoadingState,
    ...
};

const request: Request = { state: 'loading', ... };
```



Number Literal Types



Number Literal Types

```
type DieRolls = 1 | 2 | 3 | 4 | 5 | 6;

const rollDie = (): DieRolls => {
    // ...
}
```



Discriminated Unions

Tagged Unions
Algebraic Data Types



Discriminated Unions

```
type Square = { type: 'square', size: number };
type Circle = { type: 'circle', radius: number };
type Rectangle = {
    type: 'rectangle', width: number, height: number
};

type Shape = Square | Circle | Rectangle;
```



Discriminated Unions

```
const area = (shape: Shape): number => {
  switch(shape.type) {
    'square': return shape.size * shape.size;
    'circle': return Math.PI * shape.radius ** 2;
    'rectangle': return shape.width * shape.height;
    default: const n: never = shape;
  }
};
```



Exhaustiveness checking

```
type Shape = Square | Rectangle | Circle | Triangle

const area = (shape: Shape): number => {
    switch(shape.type) {
        'square': return shape.size * shape.size;
        'circle': return Math.PI * shape.radius ** 2;
        'rectangle': return shape.width * shape.height;
        default: const n: never = shape; // Compile error
    }
};
```

The logo consists of the letters 'T' and 'S' in white, bold font, positioned on a solid blue square background.

TS

Discriminated Unions are everywhere

```
type PaymentMethod = CreditCard | PayPal | Cash;

const processPayment = (method: PaymentMethod) => {
  switch(method.type) {
    case 'credit-card': return `CC: ${method.cardNo}`;
    case 'paypal': return `PayPal: ${method.email}`;
    case 'cash': return 'Cash';
    case default: const n: never = method;
  }
};
```



Discriminated Unions are everywhere

```
type AuthMethod = Cookies | Headers;  
  
type Packet = Ping | Odometry | GPSLocation;  
  
type ProductTypes = Shirts | Shoes | Wallets;
```



Lab

- day-1/3-ts-unions
- Verify that tests pass.
- Change tsconfig.json as shown in README.md
- Change src/index.ts to use union types.

Cheat sheet

- Modify `src/index.ts` to use union types
- Bonus: Add a PayPal payment type

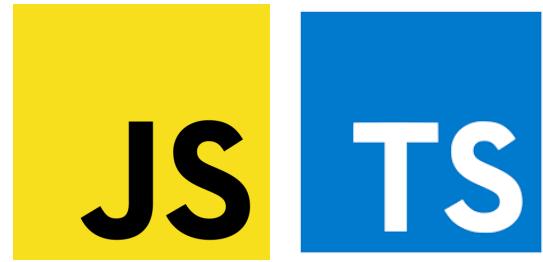
```
type Square = { kind: 'square', size: number };
type Circle = { kind: 'circle', radius: number };

type Shape = Square | Circle;
```



Union Types





Classes

Actually, I invented the term Object-Oriented, and I can tell you I did not have C++ in mind.

– Alan Kay

The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana... and the entire jungle.

— Joe Armstrong, creator, Erlang

What is Object Oriented Programming?

What can OOP do that others can't?

Objects are (usually) containers for mutable state
State is bad!

We'll talk more about state in the next section

Design Patterns

Design Patterns are ways to overcome limitations
in a programming language

Design Patterns from Java, C++, etc. don't apply
to languages like JavaScript

OO pattern/principle

- Single Responsibility Principle
- Open/Closed principle
- Dependency Inversion Principle
- Interface Segregation Principle
- Factory pattern
- Strategy pattern
- Decorator pattern
- Visitor pattern

FP pattern/principle

- Functions
- Functions
- Functions, also
- Functions
- Yes, functions
- Oh my, functions again!
- Functions
- Functions ☺

Scott Wlaschin

JavaScript has first-class functions

This makes it more powerful than classical OO languages

Most OO patterns are redundant in JS

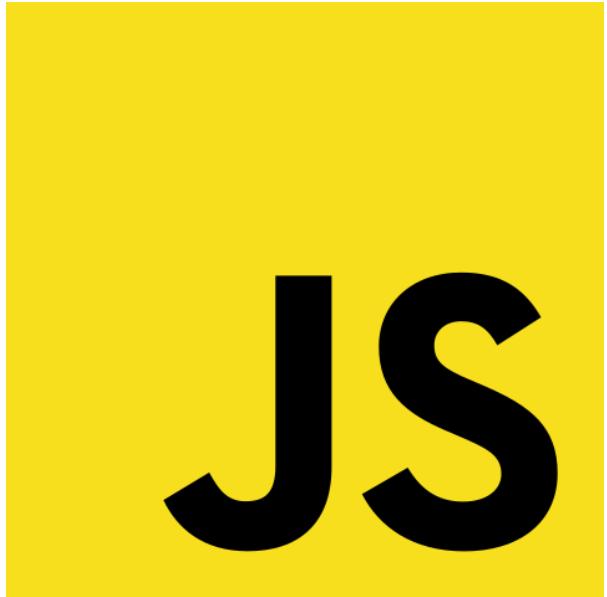
The complexity of OO

- Classes
- Inheritance
- Polymorphism
- Public/private/protected members
- Getters and setters
- `this`

Temporal Complexity

OO in JS is unfamiliar

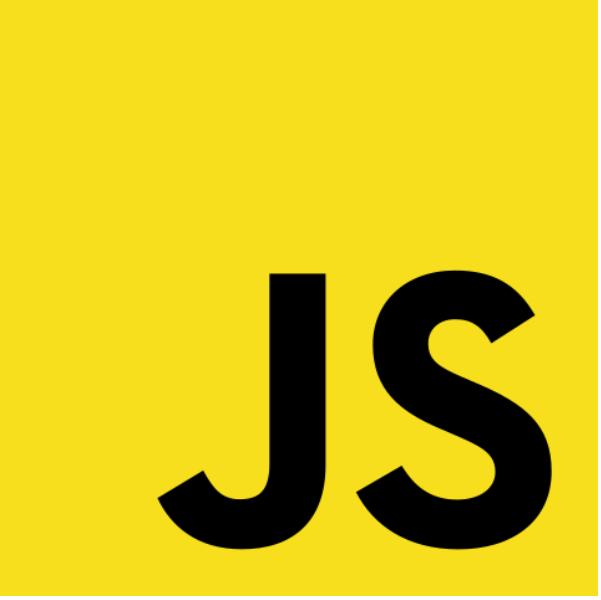
- Prototypal inheritance
- Dynamically scoped this

A large, bold, black "JS" logo on a yellow background.

JS

OO in JS

```
let Greeter = (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
}());
```

A large yellow square containing the letters "JS" in a bold, black, sans-serif font.

JS

class syntax sugar

```
class Greeter {  
  constructor(message) {  
    this.greeting = message;  
  }  
  greet() {  
    return "Hello, " + this.greeting;  
  }  
}
```



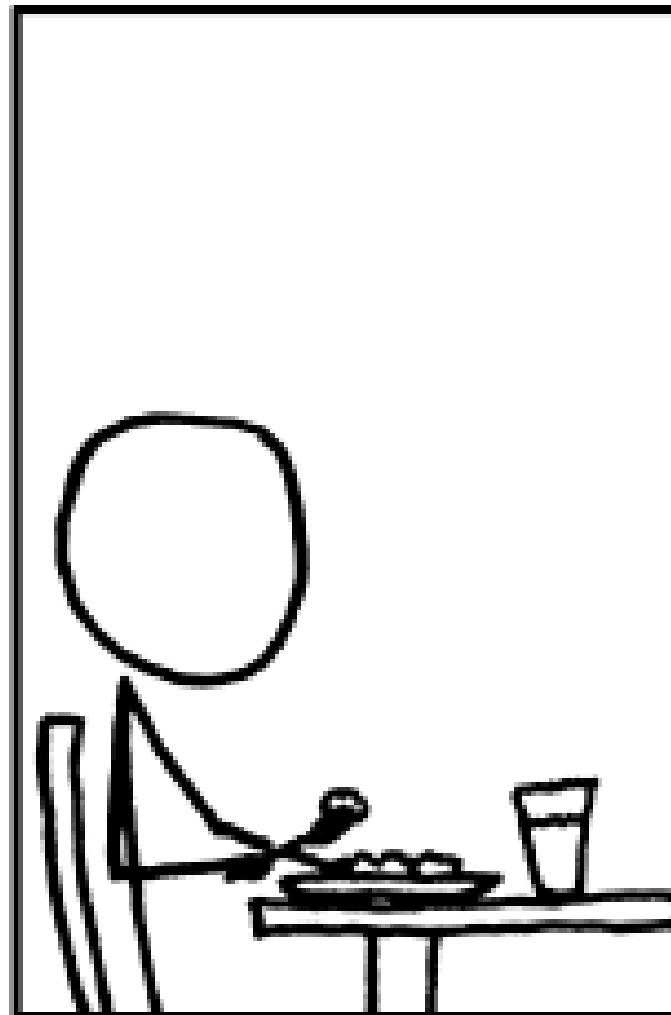
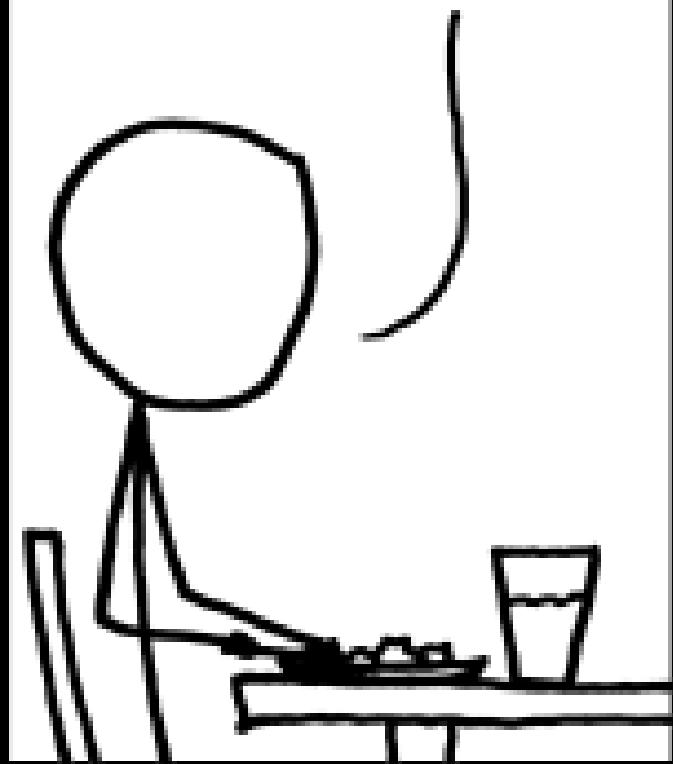
class with types

```
class Greeter {  
    greeting: string;  
    constructor(message: string) {  
        this.greeting = message;  
    }  
    greet() {  
        return "Hello, " + this.greeting;  
    }  
}
```

AbstractAnnotationConfigDispatcherServletInitializer
springframework.web.servlet

This doesn't make your code better!

CAN YOU PASS
THE SALT?



I SAID-

I KNOW! I'M DEVELOPING
A SYSTEM TO PASS YOU
ARBITRARY CONDIMENTS.

IT'S BEEN 20
MINUTES!

)
IT'LL SAVE TIME
IN THE LONG RUN!



Complexity is your enemy. Any fool can make something complicated. It is hard to keep things simple.

— Richard Branson

Avoid classes

Too much complexity, too little gain





Functional Programming

Functions in maths

$$f(x) = x + 1$$

Result only depends on its argument

Pure functions

- Result only depends on arguments
- Same results for the same arguments
- Doesn't cause anything else to change



Pure functions

```
const add = (x, y) => x + y;
```

```
String.prototype.toUpperCase
```

```
Math.floor
```



Side effects



Side effects

```
let currentUserId = 3; // state

const setCurrentUserId = userId => {
  currentUserId = userId; // side-effect
  return id;
};

setCurrentUserId(4);
```



State is a slice of time

In the presence of side effects, a program's behaviour may depend on history; that is, the order of evaluation matters.

Understanding and debugging a function with side effects requires knowledge about the context and its possible histories.

– *Side effect, Wikipedia*

Mutable state is very complex



Side effects

```
const addToArray = (arr, item) => {
    arr.push(item); // Mutated argument!
    return arr;
};
```



Side effects

```
// Other examples of side effects
console.log('Hello world');
Math.random();
Date.now();
fetch(url);
```

It is not possible to avoid side-effects

But we can at least externalize it

Make the impure pure



Make the impure pure

```
const currentId = 0;  
const getNextId = () => {  
    currentId++;  
    return currentId;  
};  
  
// becomes  
  
const getNextId = currentId => currentId + 1;  
getNextId(0);
```



Make the impure pure

```
const addToArray = (arr, item) => {  
    arr.push(item);  
    return arr;  
};
```

// becomes

```
const addToArray = (arr, item) => ([...arr, item]
```



Make the impure pure

```
const addObject = (obj, key, value) => {  
  obj[key] = value;  
  return obj;  
};
```

// becomes

```
const addObject = (obj, key, value) =>  
  ({ ...obj, [key]: value }) ;
```

Why pure functions?



Cacheability

```
const add2 = x => x + 2;  
  
add2(4); // output: 6  
add2(4); // output: still 6  
add2(4); // output: it isn't going to change
```



Testable

Testing is easy, since setup is minimal.



Easy to reason about

No external dependencies

Referential transparency



Referential Transparency

```
const increment = x => x + 1;  
  
const afterZero = increment(0);
```



Parallelisation



Functional Programming





First Class Functions



Functions are
types



Functions are types

```
const x = () => {};  
  
console.log(typeof x); // output: 'function'
```



Functions are types

```
const foo = {  
    bar: () => {}  
};
```

```
const foo = [  
    () => {},  
    () => {}  
];
```



Functions are types

```
fs.readFile('/path/to/file', (err, data) => {  
    // ...  
} );
```



Functions are types

```
new Promise( (resolve, reject) => {  
    resolve();  
} );
```



Currying





Currying

```
const prop = key => obj => obj[key];  
  
const name = prop('name');  
  
console.log(name({ name: 'Alice' })); // output: 'Alice'
```



Currying

```
const prop = key => obj => obj[key];  
const name = prop('name');  
console.log(name({ name: 'Alice' })); // output: 'Alice'
```

```
const prop = key => {  
  return obj => obj[key];  
};
```



Higher-order functions



Thinking in lists

or, "*never write a for loop again*"



List

```
// Every array iteration looks like this

for(var i = 0; i < arr.length; i++) {
    // do something with arr[i]
}
```



Array.map



Array.map

```
const arr = [1, 2, 3];

const doubled = arr.map(
  x => x * 2
);

console.log(doubled); // output: [2, 4, 6]
```



Array.map

```
const arr = [1, 2, 3];
const double = x => x * 2;

const doubled = arr.map(double);

console.log(doubled); // output: [2, 4, 6]
```



Array.map

```
const arr = [
  { name: 'Alice' },
  { name: 'Bob' }
];

const prop = key => obj => obj[key];

const name = prop('name');

console.log(arr.map(name)); // output: 'Alice', 'Bo
```



Array.map

```
const readFilePart = new Promise(/* ... */);

const promises = [1, 2, 3].map(readFilePart);

const results = await Promise.all(promises);
```



Array.map

```
const arr = [1, 2, 3];
const doubled = arr.map(x => x * 2);
```

// vs

```
const arr = [1, 2, 3];
const doubled = [];
for(let i = 0; i < arr.length; i++) {
  doubled.push(arr[i] * 2);
}
```



Array.filter



Array.filter

```
const arr = [1, 2, 3, 4];  
  
const isEven = x => x % 2 === 0;  
  
arr.filter(isEven); // output: [2, 4]
```



Array.filter

```
const arr = [1, 2, 3, 4];  
  
arr.filter(  
  x => x > 2  
);  
  
// output: [3, 4]
```



Array.filter

```
const people = [
  { name: 'Alice', age: 42 },
  { name: 'Bob', age: 21 }
];

const under30 = ({ age }) => age < 30;

console.log(
  people.filter(under30)
);
```



Combining .map and .filter

```
const people = [  
  { name: 'Alice', age: 42 },  
  { name: 'Bob', age: 21 }  
];  
  
people  
.filter(isOver30)  
.map(prop('name'));  
  
// output: [ 'Alice' ]
```



Array.some



Array.some

```
const ages = [10, 20, 30, 40];
const olderThan30 = x => x > 30;

ages.some(olderThan30); // true
```



Array.some

```
const colours = [  
  'red', 'green', 'blue'  
];  
  
const isYellow = x => x === 'yellow';  
  
colours.some(isYellow); // output: false
```



Array.every



Array.every

```
const ages = [  
    10, 20, 30, 40  
];  
  
const under50 = x => x < 50;  
const isAdult = x => x >= 21;  
  
ages.every(under50); // output: true  
ages.every(isAdult); // output false
```



Array.reduce

```
const add = (a, b) => a + b;  
  
[1, 2, 3].reduce(add); // output: 6
```



Array.reduce

```
const add = (a, b) => a + b;  
  
[1, 2, 3].reduce(add); // output: 6
```



Array.reduce

```
const add = (a, b) => a + b;

[1, 2, 3].reduce(
  (accumulator, value) => add(accumulator, value),
  0
);
```



Array.reduce

```
const fruits = [
  'banana', 'cherry', 'orange', 'apple',
  'cherry', 'orange', 'apple', 'banana',
  'cherry', 'orange', 'fig'
];
```

```
const counts = fruits.reduce(
  (totals, fruit) => ({
    ...totals,
    [fruit]: (totals[fruit] || 0) + 1
  })
);
```



Lab

- day-1/4-fp
- Change `src/index.ts` to use array helpers.

Cheat sheet

```
[1, 2, 3].map(x => x * 2); // [2, 4, 6]
[1, 2, 3].filter(x => x < 3); // [1, 2]
[1, 2, 3].some(x => x > 5); // false
[1, 2, 3].every(x => x > 1); // false
[1, 2, 3].reduce((a, b) => a + b, 0); // 6
```



Array helpers

```
[1, 2, 3].forEach(x => {  
    // ...  
});  
  
// for example...  
[1, 2, 3].forEach(console.log);
```





Clean code

Programming is the art of telling another human what one wants the computer to do.

— Donald Knuth



Variable names

```
// Don't:  
const arr = [ ... ];  
const x = ...;  
const ages = arr.map(x => x.age);  
  
// Do:  
const people = [ ... ];  
const ages = people.map(person => person.age); // or destruct
```



Variable names

```
// Don't:  
const nameString = ...;  
const theUsers = ...;
```

```
// Do:  
const name = ...;  
const users = ...;
```



Variable names

```
// Don't:  
const fname = ...;  
const lname = ...;  
const cntr = 0;  
  
// Do:  
const firstName = ...;  
const lastName = ...;  
const counter = 0;
```



Variable names

```
// Bad:
```

```
const inv = e => ...
```

```
// Good:
```

```
const invite = email => ...
```

```
// Better:
```

```
const inviteUser = email => ...
```



Variable names

```
// Don't:  
const getMatchingUser = (fields, include, fromDate, toDate) =  
    // ...  
};  
  
getMatchingUser(  
    ['firstName', 'lastName'],  
    ['invitedUsers'],  
    '2019-01-01',  
    '2019-01-27'  
);
```



Variable names

```
// Don't:  
const getMatchingUser = ({ fields, include, fromDate, toDate }) =  
  // ...  
};  
  
getMatchingUser({  
  fields: ['firstName', 'lastName'],  
  include: ['invitedUsers'],  
  fromDate: '2019-01-01',  
  toDate: '2019-01-27'  
});
```



Variable names

```
// Don't:  
getUserInfo();  
getClientData();  
getCustomerRecord();
```

```
// Do:  
getUser();
```



Variable names

```
// Don't:  
setTimeout(doSomething, 86400000);  
  
// Do:  
const millisecondsInADay = 86400000;  
setTimeout(doSomething, millisecondsInADay);
```



Variable names

```
// Don't:  
  
const Car = {  
    carMake: 'Honda',  
    carModel: 'Accord',  
    carColor: 'Blue'  
};  
  
// Do:  
  
const car = {  
    make: 'Honda',
```



Variable names

```
// Don't:  
const addToDate = (date, month) => ...  
addToDate(date, 1);  
  
// Do:  
const addMonthToDate = (date, month) => ...  
addMonthToDate(date, 1);
```



Long functions

Good maximum length for a function: 4
- 5 lines



Long functions

```
// Don't:  
const createFile = (name, temp) => {  
    if (temp) {  
        fs.create(`./temp/${name}`);  
    } else {  
        fs.create(name);  
    }  
  
    createFile('readme.md', true);
```



Long functions

```
// Do:  
const writeFile = name => fs.create(name);  
  
const createTempFile = name => writeFile(`./temp/${name}`)
```



Conditionals



Conditionals

```
// Don't:  
if(state === 'loading' && list.length === 0) {  
    // ...  
}  
  
// Do:  
const shouldUseFallback = (state, list) =>  
    state === 'loading' && list.length === 0;  
  
if(shouldUseFallback(state, list)) {  
    // ...
```



Conditionals

```
// Don't:  
const jsonIsNotAvailable = ...;  
if(!jsonIsNotAvailable) { ... };
```

```
// Do:  
const jsonIsAvailable = ...;  
if(jsonIsAvailable) { ... };
```



Conditionals

```
// Don't:  
const jerseyColour = team => {  
    if(team === 'India') return 'blue';  
    if(team === 'Zimbabwe') return 'red';  
    if(team === 'Pakistan') return 'green';  
}
```



Lab

- day-3/2-clean-code
- Change `src/index.ts` to improve code quality.



Conditionals

```
// Do:  
const jerseyColourByTeam = {  
    India: 'blue',  
    Zimbabwe: 'red',  
    Pakistan: 'green'  
};  
  
const jerseyColour = team => jerseyColourByTeam[team];
```



More clean
code



Avoid
comments



Avoid comments

```
// Don't:  
// Increments a number by 1  
const inc = x => x + 1;  
  
// Do:  
const increment = num => num + 1;
```



Avoid comments

```
// Don't:  
// Calculates taxes  
const calcT = ...  
  
// Do:  
const calculateTaxes = ...
```



Avoid comments

```
// Don't:  
// This function calculates how many  
// people are in the list are adults  
const aCnt = ...;
```

```
// Do:  
const adultCount = ...;
```



Avoid comments



If you have to use comments, use it to explain “*why*” you’re doing something.

But prefer renaming things instead.



Dead code



Dead code

```
// Don't  
const old GetUser = ...  
const new GetUser = ...
```

```
// Do:  
const GetUser = ...
```



Dead code

```
// Don't
/*
Not used anymore
const someFunction = ...
*/
// Do: Just delete it
```



Primitive
obsession



Primitive obsession

```
// Don't:  
if(users.indexOf(user) != -1) { /* ... */ }
```

```
// Do:  
if(users.includes(user)) { /* ... */ }
```



Primitive obsession

```
// Don't:  
for(var i = 0; i < arr.length; i++) {  
    // use arr[i]  
}  
  
// Do:  
arr.map(...) // or .filter or .reduce
```



Late returns

```
// Don't:  
const someFn = users => {  
  if(users) {  
    if(users.length) {  
      ...  
    }  
  }  
}
```



Late returns

```
// Do:  
const someFn = users => {  
    if (!users) return;  
    if (!users.length) return;  
  
    // ...  
}
```



Lab

- day-3/3-clean-code
- Change `src/index.ts` to improve code quality.



Clean code



The 12-factor
app



The 12-factor app

The 12 things an app should do to be deployment-ready.

<https://12factor.net/>

- Codebase
 - Dependencies
 - Config
 - Backing services
 - Build, release, run
 - Processes
-
- Port binding
 - Concurrency
 - Disposability
 - Dev/prod parity
 - Logs
 - Admin processes



Codebase

The app should be in a single repository



Dependencies

All required dependencies should be explicitly declared



Config

Configuration parameters should be in the environment



Config

```
TWILIO_KEY=ca61bac4... npm start
```

In node:

```
const twilioKey = process.env.TWILIO_KEY
```



Backing services

DB, S3, etc, should be accessed by URL



Backing services

DB, S3, etc, should be accessed by URL



Build, release, run



Processes

Apps should be stateless processes



Expose services via port binding



Concurrency

Scale out via the process model



Disposability

Optimize for frequent startup/shutdown



Dev/prod parity

Keep all the environments similar



Logs

Write logs to stdout



Admin processes

Script admin tasks for adhoc runs