Hello

Rakesh Pai

@rakesh314

Developer

Since 2002

{errorception}

500+ customers

10,000+ concurrent errors

Node.js + MongoDB

{errorception}

Frequent speaker

Ex - program committee

34 repos

18 packages

Name

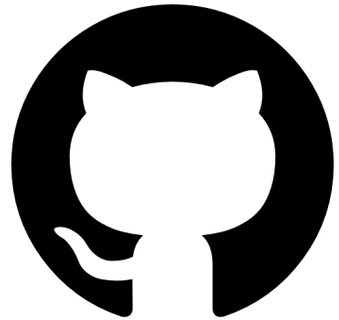
- Role
- 2-line summary of your experience
- Expectations from this workshop

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



JavaScript

ECMAScript

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

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

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;
```

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 getFilePath = ({ filePath }) => filePath;

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

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

Promises

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));
};
```

async/await

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

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
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

Transpilers

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

Modules

- Browser
- Node.js
- Unforseen 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;
  }
);
```

import/export

ES2015

Syntax

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

// --

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

Statically analysable

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

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

Default exports

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

// In another file
import thatFunction from './path/to/module';
```

Named exports

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

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

Combining named and default

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

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

Current status for `import`

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

What should we do?

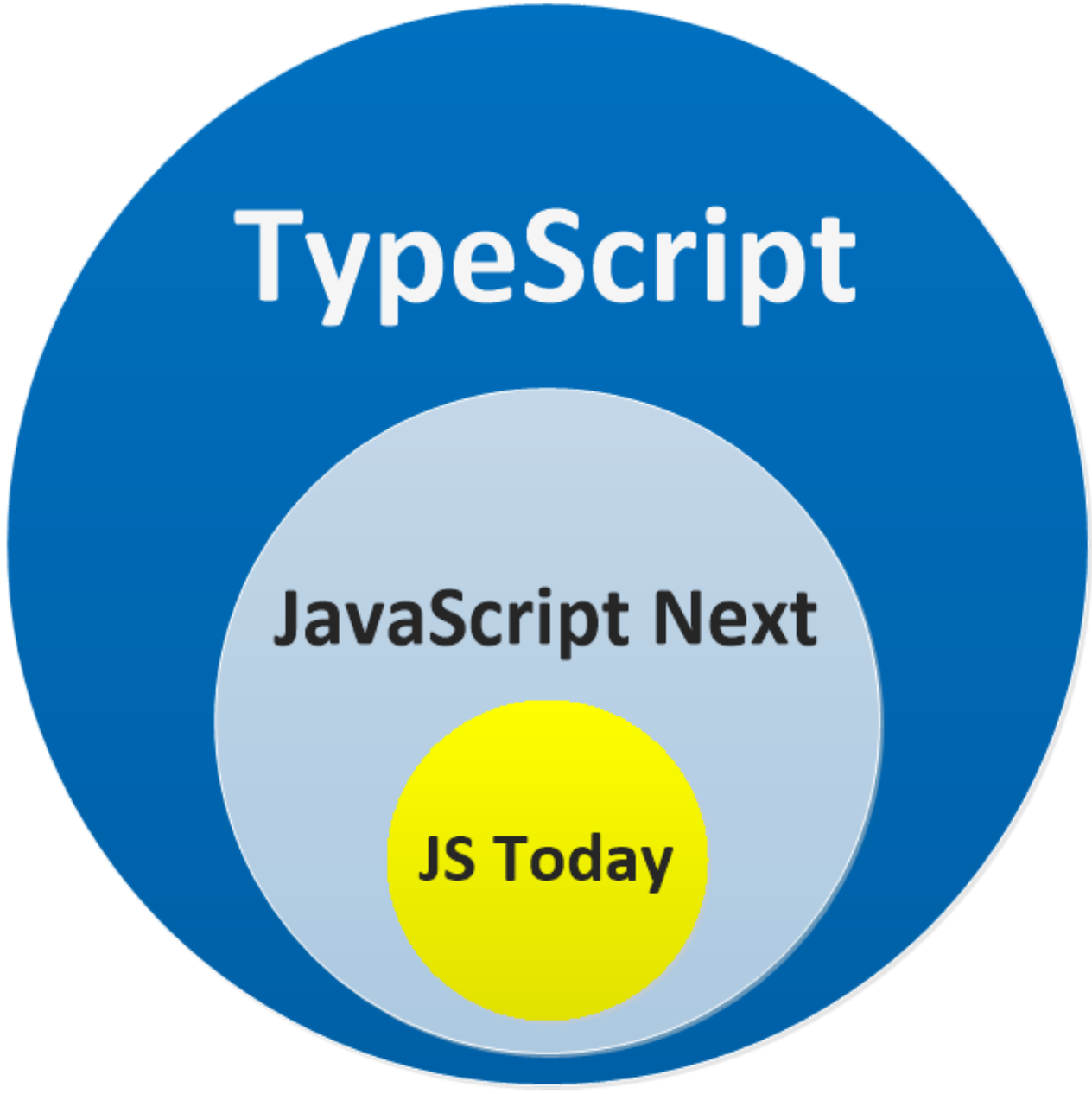
- We must follow language standards
- But we can't use the standards!
- We can use transpilers! But should we?

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 /* ... */
```

Modules

Modules

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

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

Modules

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

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