JavaScript in 2018

Elijah Verdoorn

How did we get here?

- Brendan Eich, 1995
  - Netscape
  - Java, Scheme, Self
- Microsoft Competition, Java developed as a web language
  - JS : Java :: VB : C++
  - JS developed rapidly in companion to Java
- Choices made in rapid development regretted

Brief history of ECMAScript

- 1997 First Edition
- 1998 Second Edition
  - Minor Edits
- 1999 Third Edition
  - Regular Expressions
- Abandoned Fourth Edition
  - Politics
- 2009 Fifth Edition
  - “strict mode”
  - Getters, setters
  - JSON
- 2015 Sixth Edition
  - for/of loops
  - Arrow functions
  - Promises

ECMAScript Standard

- https://tc39.github.io/ecma262/
- This Standard defines the ECMAScript 2018 general-purpose programming language.
  - Key: general-purpose

Getting Started

Using the Node.JS REPL

```
$ node
> let x = 1

> X
1
```

ctrl-d to send EOF to REPL, exiting

Feature List

https://babeljs.io/learn-es2015/

Learning the 2015 standard is the fastest way to learning the newest versions

Arrows

- Arrows are a function shorthand using the => syntax. They are syntactically similar to the related feature in C#, Java 8 and CoffeeScript. They support both statement block bodies as well as expression bodies which return the value of the expression. Unlike functions, arrows share the same lexical this as their surrounding code.

Arrows

```
// Expression bodies
var odds = evens.map(v => v + 1);
var nums = evens.map((v, i) => v + i);
var pairs = evens.map(v => ({even: v, odd: v + 1}));

// Statement bodies
nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
});

// Lexical this
var bob = {
  _name: "Bob",
  _friends: [],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " knows " + f));
  }
}
```

Classes

- ES6 classes are a simple sugar over the prototype-based OO pattern. Having a single convenient declarative form makes class patterns easier to use, and encourages interoperability. Classes support prototype-based inheritance, super calls, instance and static methods and constructors.

Classes

```
class SkinnedMesh extends THREE.Mesh {
  constructor(geometry, materials) {
    super(geometry, materials);

    this.idMatrix = SkinnedMesh.defaultMatrix();
    this.bones = [];
    this.boneMatrices = [];
    //...
  }
  update(camera) {
    //...
    super.update();
  }
  get boneCount() {
    return this.bones.length;
  }
  set matrixType(matrixType) {
    this.idMatrix = SkinnedMesh[matrixType]();
  }
  static defaultMatrix() {
    return new THREE.Matrix4();
  }
}
```

Template Strings

- Template strings provide syntactic sugar for constructing strings. This is similar to string interpolation features in Perl, Python and more. Optionally, a tag can be added to allow the string construction to be customized, avoiding injection attacks or constructing higher level data structures from string contents.

Template Strings

```
// Basic literal string creation
`In JavaScript '\n' is a line-feed.`

// Multiline strings
`In JavaScript this is
 not legal.`

// String interpolation
var name = "Bob", time = "today";
`Hello ${name}, how are you ${time}?`
```

Let / Const

- Block-scoped binding constructs. let is the new var. const is single-assignment. Static restrictions prevent use before assignment.

Let / Const

```
function f() {
  {
    let x;
    {
      // okay, block scoped name
      const x = "sneaky";
      // error, const
      x = "foo";
    }
    // error, already declared in block
    let x = "inner";
  }
}
```

For…Of

```
for (var n of fibonacci) {
  // truncate the sequence at 1000
  if (n > 1000)
    break;
  console.log(n);
}
```

Modules

- Language-level support for modules for component definition. Codifies patterns from popular JavaScript module loaders (AMD, CommonJS). Runtime behaviour defined by a host-defined default loader. Implicitly async model – no code executes until requested modules are available and processed.

Modules

```
// lib/math.js
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;
```

```
// app.js
import * as math from "lib/math";
alert("2π = " + math.sum(math.pi, math.pi));
```

```
// otherApp.js
import {sum, pi} from "lib/math";
alert("2π = " + sum(pi, pi));
```

Map / Set

```
// Sets
var s = new Set();
s.add("hello").add("goodbye").add("hello");
s.size === 2;
s.has("hello") === true;

// Maps
var m = new Map();
m.set("hello", 42);
m.set(s, 34);
m.get(s) == 34;
```

Promises

Why do we need them?

- JS is synchronous, single threaded
- Event driven programming – essential for working with UIs
- Callbacks – the old solution

Event Principals:

- Main loop
- Listeners
- Event package

Promise Basics

- A promise can be:
  - fulfilled - The action relating to the promise succeeded
  - rejected - The action relating to the promise failed
  - pending - Hasn't fulfilled or rejected yet
  - settled - Has fulfilled or rejected

Promise Sample

```
function timeout(duration = 0) {
    return new Promise((resolve, reject) => {
        setTimeout(resolve, duration);
    })
}

var p = timeout(1000).then(() => {
    return timeout(2000);
}).then(() => {
    throw new Error("hmm");
}).catch(err => {
    return Promise.all([timeout(100), timeout(200)]);
})
```

Common Uses

- HTTP requests
- File access
- Database query
- Sensor readings
- Waiting for user actions
- Calculation

What's the Point?

DON'T HANG THE UI THREAD!

User experience with our app is above all else – users are willing to wait, but won't tolerate freezing or hanging.

Questions?