# Introduction to ES2015

# JavaScript Versioning

## JavaScript Versioning

- Behind JavaScript, is EMCAScript
    - The standard is called ECMA-262
- Up to now, JavaScript was always based on a number
    - Now it is a year-based schema

## JavaScript: Beyond

- Proposals follow the TC39 process
    - Stage 0: Strawman
    - Stage 1: Proposal
    - Stage 2: Draft
    - Stage 3: Candidate
    - Stage 4: Finished
- Other related stuff (e.g. WebAssembly)

# Dealing with new Versions

## Working with future JS, today

- Ignore it
- Polyfills and Shims
- Transpilation

# Variables in JavaScript

# Variables in JavaScript

## What's wrong with `var`?

- Function Scoping vs. Block Scoping
- No **Temporal Dead Zone**

# `let`

- Block Scoped
- Can be re-assigned

```
let something = true;
```

## `const`

- Block Scoped
- Can't be re-assigned
- It has an **immutable binding**

```
const favNumber = 42;
```

# Functions in JavaScript

## Enter: Arrow Functions

- More concise
- The option of implicit returns
- They are always anonymous though

# Arrow Functions

```
const sayHi = () => {
  console.log("Hello");
};

sayHi();
```

# Arrow Functions

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

add(4, 5);
```

# Arrow Functions

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

add(4, 5);
```

# Default Function Arguments

```
function sayHello(name = "World") {
  return "Hello " + name;
}
```

# Template Strings

# Template Strings

- Three ways to make strings
    - Single quotes
    - Double quotes
    - Backticks
- Template strings || Interpolation

# Template Strings

```
const brand = "Wurlitzer";
const keys = 64;
const message = `My ${brand} keyboard has ${keys} or ${keys / 8} octaves`;
```

# Destructuring

# Destructuring

```
const details = ["Groucho", "Marx", "Duck Soup"];
const [first, last, bestMovie] = details;

const explorer = {
  first: "Jacques",
  last: "Cousteau"
};

const { first, last } = explorer;
```

# Enhanced Object Literals

# Enhanced Object Literals

```
const firstName = "Jacques";
const lastName = "Cousteau";

const jacques = {
  firstName,
  lastName,
  saying: "Hello World",
  speak(message) {
    console.log(message);
  }
};
```

# Asynchronous Programming with JavaScript

# Dealing with Asynchronicity

There are lots of different ways to deal with asynchronicity

- Callbacks
- **Promises**
- Now, async/await

# Promises

## What are promises?

- Promises represent eventual results of an asynchronous operation
- It's an object that may produce a single piece of data at some point in the future
    - Either a resolved value
    - Or a rejection (an error that tells us why it wasn't resolved)

## States & Fates

- Promises have three mutually exclusive potential states:
    - Fulfilled: The action relating to the promise succeeded
    - Rejected: The action relating to the promise failed
    - Pending: Hasn't fulfilled or rejected yet
- We say that a promise is "settled" if it isn't pending

## States & Fates

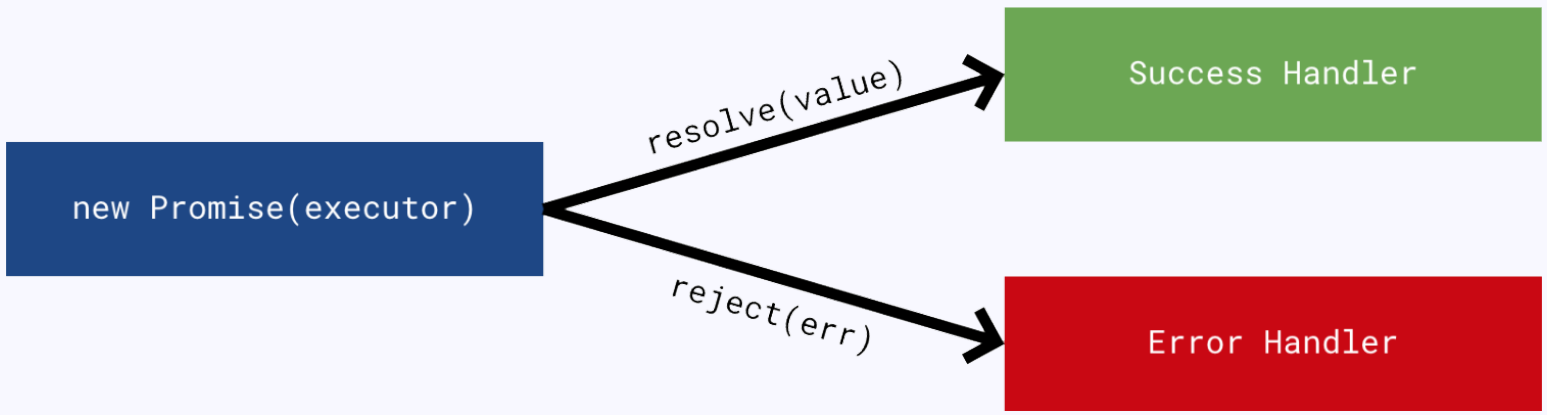
- Promises have two mutually exclusive potential fates:
  - Resolved: Finished (or locked into a thenable or another promise)
  - Unresolved: If trying to resolve or reject will make an impact

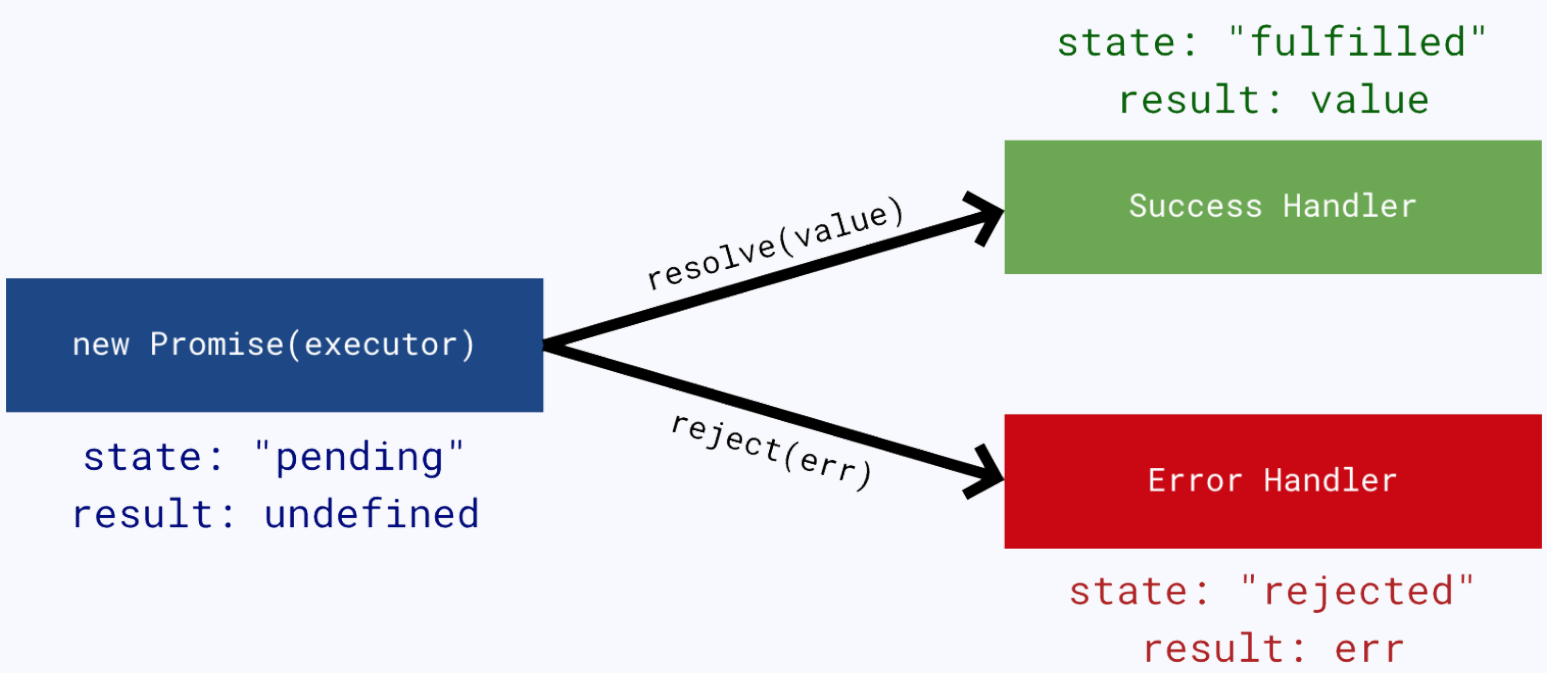
## A little like event listeners

- A promise can only succeed or fail once
    - If a promise has succeeded or failed and you later add a success/failure callback
- The correct callback will be called (even though the event happened earlier)

## Why use promises?

- They help us write readable code
- They help us deal with the complexities of asynchronous programming
- They help us avoid "callback hell" || "the pyramid of doom"
- They are the backbone of some of the newer features coming out with JavaScript
- Lots of libraries/frameworks/packages use promises
    - We will have to use them all of the time


new Promise(executor)
resolve(value)
reject(err)
Success Handler
Error Handler


state: "fulfilled"
result: value
Success Handler
resolve(value)
new Promise(executor)
reject(err)
Error Handler
state: "pending"
result: undefined
state: "rejected"
result: err

# Creating Promises

```
var promise = new Promise(function(resolve, reject) {
  if (true) {
    resolve("Will go to the .then");
  } else {
    reject("Will go to the .catch");
  }
});
```

The **executor** callback function automatically receives a:

- **resolve** function
- **reject** function

# Creating Promises

```
var promise = new Promise(function(resolve, reject) {
  if (true) {
    resolve("Will go to the .then");
  } else {
    reject("Will go to the .catch");
  }
});
```

The **executor** callback function automatically receives a:

- **resolve** function
- **reject** function

# Consuming Promises

```
var promise = new Promise(function(resolve, reject) {
  if (true) {
    resolve("Will go to the .then");
  } else {
    reject("Will go to the .catch");
  }
});

promise.then(function(data) {
  console.log(data);
});
```

# Chaining Promises

```
function getNumbers() {
  return new Promise(function(resolve, reject) {
    resolve([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
  });
}

function filterToEvenNumbers(nums) {
  /* ... */
}

function multiplyByFive(nums) {
  /* ... */
}

getNumbers()
  .then(filterToEvenNumbers)
  .then(multiplyByFive);
```

# Handling Errors

```
var promise = new Promise(function(resolve, reject) {
  if (false) {
    resolve("Will go to the .then");
  } else {
    reject("Will go to the .catch");
  }
});

function successHandler() {
  /* ... */
}

function errorHandler() {
  /* ... */
}

promise.then(successHandler).catch(errorHandler);
```

Data provided to **reject** is passed to `.catch`

# In-class Exercise / Homework

Turn an event into a promise!

I want to be able to write something along these lines:

```
onClick("h1").then(/* ... */);

onClick("p").then(/* ... */);
```

## Resources

- MDN: Promises and MDN: Using Promises
- Google Web Fundamentals: Promises
- JavaScript.info: Promises
- Scotch: JavaScript Promises
- David Walsh: Promises
- You Don't Know JS: Promises
- Exploring JS: Promises
- Eric Elliot: Promises
- Domenic: The Point of Promises