ECMAScript 6 in theory and practice

Fluent Conference, 2015-04-20

Slides: speakerdeck.com/rauschma

JavaScript is everywhere

- In browsers (**big** apps), servers, devices, robots, …
- Does much more than what it was originally created for.
- How can we help it with that?

© Schmeegan

ECMAScript 6 (ES6): JavaScript, improved

ECMAScript 6: next version of JavaScript (current: ES5).

This presentation:

- Background (terms, goals, design process)
- Using ES6 today
- Features

Background

Goals for ECMAScript 6

Amongst other official goals: make JavaScript better

- for complex applications
- for libraries (including the DOM)
- as a target of code generators

How to upgrade a web language?

Challenges w.r.t. upgrading:

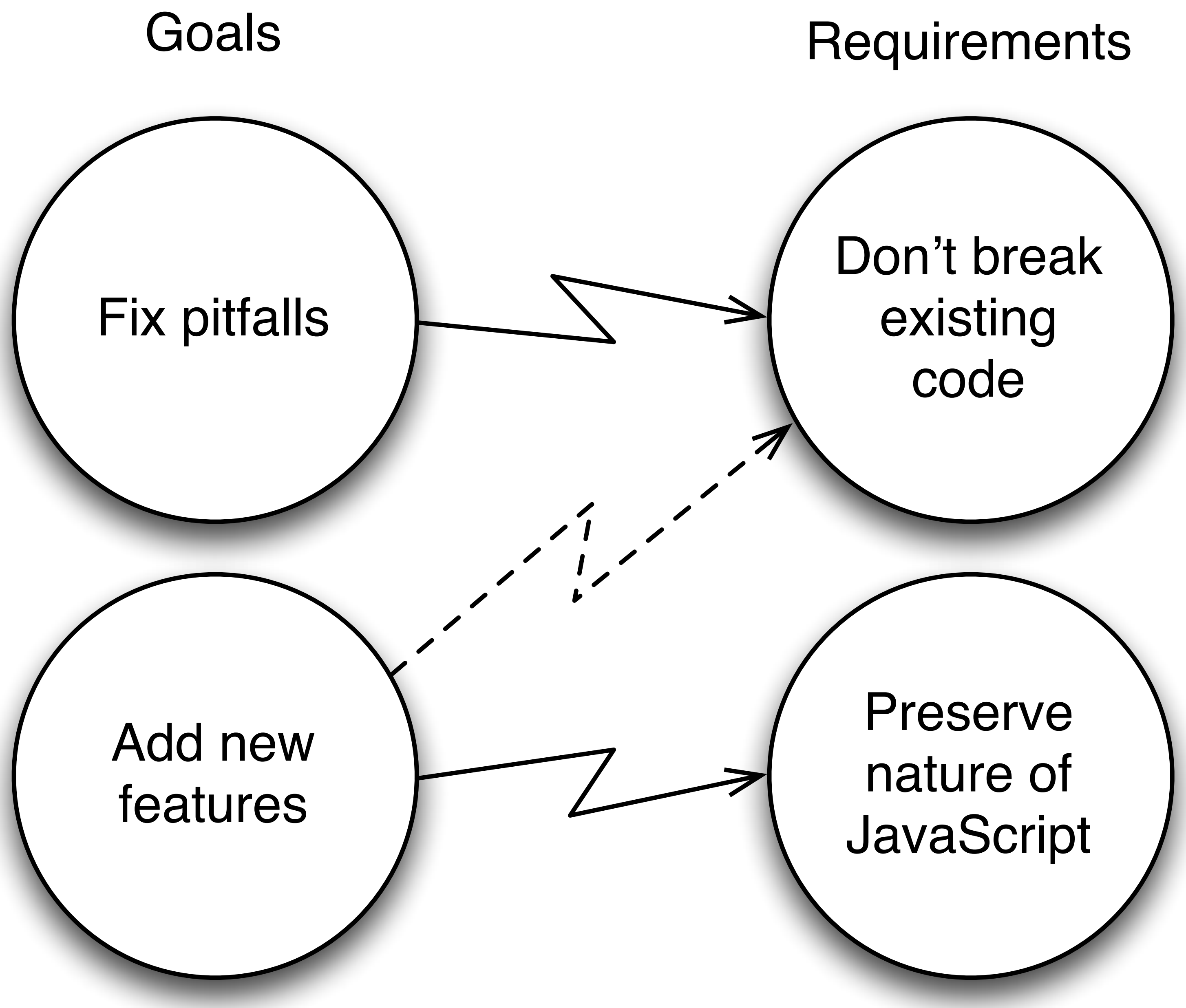
- **JavaScript engines:**
  - New versions = forced upgrades
  - Must run all existing code

  ⇒ ECMAScript 6 is a superset of ES5 (nothing is removed)

- **JavaScript code:**
  - Must run on all engines that are in use

  ⇒ wait or compile ECMAScript 6 to ES5 (details later).

Goals and requirements



How ECMAScript features are designed

Avoid “design by committee”:

- Design by “champions” (1–2 experts)
- Feedback from TC39 and web development community
- Field-testing and refining via one or more implementations
- TC39 has final word on whether/when to include

ES7+: smaller, yearly scheduled releases

Overview of features

Better syntax for existing features. E.g.:

- Classes
- Modules

Better standard library. E.g.:

- New methods for strings, arrays
- Promises
- Maps, Sets

Completely new features. E.g.:

- Generators
- Proxies
- WeakMaps

Using ES6 today

Time table

ECMAScript 6 is done:

- The spec is frozen
- June 2015: formal publication
- Features are continually appearing in current engines.

ES6 tools

Transpiler (your code):

- TypeScript
- Traceur
- Babel

Traceur and Babel

Run...

- **Statically (at development time):** use build tools (Grunt, Gulp, Broccoli, etc.) to generate modules. E.g.:
    - AMD (RequireJS)
    - CommonJS (Node.js, Browserify, etc.)
- **Dynamically (at runtime):** library plus `<script type="...">`

Variables and scoping

Block-scoped variables

```
// Function scope (var)
function order(x, y) {
  if (x > y) {
    var tmp = x;
    x = y;
    y = tmp;
  }
  console.log(tmp===x);
    // true

  return [x, y];
}
```

```
// Block scope (let,const)
function order(x, y) {
  if (x > y) {
    let tmp = x;
    x = y;
    y = tmp;
  }
  console.log(tmp===x);
    // ReferenceError:
    // tmp is not defined
  return [x, y];
}
```

Destructuring

Constructing vs. extracting

Construct

```
// Single values
let jane = {};
jane.first = 'Jane';
jane.last = 'Doe';

// Multiple values
let jane = {
  first: 'Jane',
  last: 'Doe'
};
```

Extract

```
// Single values
let f = jane.first;
let l = jane.last;

// Multiple values
let ??? = jane;
```

Destructuring

Extract multiple values via patterns:

```
  let obj = { first: 'Jane', last: 'Doe' };
  let { first: f, last: l } = obj;
      // f='Jane', l='Doe'
```

Can be used for:

- variable declarations (`var`, `let`, `const`)
- assignments
- parameter definitions

Destructuring: arrays

```
let [x, y] = ['a', 'b'];
    // x='a', y='b'

let [x, y, ...rest] = ['a', 'b', 'c', 'd'];
    // x='a', y='b', rest = [ 'c', 'd' ]

[x,y] = [y,x];  // swap values

let [all, year, month, day] =
    /^(\d\d\d\d)-(\d\d)-(\d\d)$/
    .exec('2999-12-31');
```

Multiple return values

```
function findElement(arr, predicate) {
    for (let index=0; index < arr.length; index++) {
        let element = arr[index];
        if (predicate(element)) {
            return { element, index };
                   // same as { element: element, index: index }
        }
    }
    return { element: undefined, index: -1 };
}
let a = [7, 8, 6];

let {element, index} = findElement(a, x => x % 2 === 0);
    // element = 8, index = 1
let {index, element} = findElement(···); // order doesn't matter

let {element} = findElement(···);
let {index} = findElement(···);
```

Modules

Modules: named exports

```
// lib/math.js
let notExported = 'abc';
export function square(x) {
    return x * x;
}
export const MY_CONSTANT = 123;
```

```
// main1.js
import {square} from 'lib/math';
console.log(square(3));
```

```
// main2.js
import * as math from 'lib/math';
console.log(math.square(3));
```

Modules: default exports

```
//----- myFunc.js -----
export default function (...) { ... }

//----- main1.js -----
import myFunc from 'myFunc';
```

```
//----- MyClass.js -----
export default class { ... }

//----- main2.js -----
import MyClass from 'MyClass';
```

Exercises

Object literals

Method definitions

```
let obj = {
    myMethod() {
        ...
    }
};

// Equivalent:
var obj = {
    myMethod: function () {
        ...
    }
};
```

Property value shorthands

```
let x = 4;
let y = 1;
let obj = { x, y };

// Same as { x: x, y: y }
```

Computed property keys (1/2)

```
let propKey = 'hello';
let obj = {
    ['fo'+'o']: 123,
    [propKey]() {
        return 'hi';
    },
};
console.log(obj.hello()); // hi
```

Computed property keys (2/2)

```
let obj = {
    // Key is a symbol
    [Symbol.iterator]() {
        ...
    }
};
for (let x of obj) {
    console.log(x);
}
```

Parameter handling

Parameter default values

Use a default if a parameter is missing.

```
function func1(x, y='default') {
    return [x,y];
}
```

Interaction:

```
> func1(1, 2)
[1, 2]
> func1()
[undefined, 'default']
```

Rest parameters

Put trailing parameters in an array.

```
function func2(arg0, ...others) {
    return others;
}
```

Interaction:

```
> func2('a', 'b', 'c')
['b', 'c']
> func2()
[]
```

No need for `arguments`, anymore.

No needs for `arguments`

```
// ES5
function func() {
  [].forEach.call(arguments,
    function (x) {···});
  ...
}
// ES6
function func(...args) {
  for (let arg of args) {
    ...
  }
}
```

Named parameters

```
// Emulated via object literals and destructuring

// { opt1, opt2 } is same as
// { opt1: opt1, opt2: opt2 }

selectEntries({ step: 2 });
selectEntries({ end: 20, start: 3 });
selectEntries(); // enabled via `= {}` below

function selectEntries(
    {start=0, end=-1, step=1} = {}) {
        ...
};
```

Spread operator (...): function arguments

```
Math.max(...[7, 4, 11]); // 11

let arr1 = ['a', 'b'];
let arr2 = ['c', 'd'];
arr1.push(...arr2);
    // arr1 is now ['a', 'b', 'c', 'd']

// Also works in constructors!
new Date(...[1912, 11, 24]) // Christmas Eve 1912
```

Turn an array into function/method arguments:

- The inverse of rest parameters
- Mostly replaces `Function.prototype.apply()`

Spread operator (...): array elements

```
let a = [1, ...[2,3], 4]; // [1, 2, 3, 4]

// Concatenate arrays
let x = ['a', 'b'];
let y = ['c'];
let z = ['d', 'e'];

let xyz = [...x, ...y, ...z];
    // ['a', 'b', 'c', 'd', 'e']

// Convert iterable objects to arrays
let set = new Set([11, -1, 6]);
let arr = [...set]; // [11, -1, 6]
```

Arrow functions

Arrow functions: less to type

```
let arr = [1, 2, 3];
let squ;

squ = arr.map(function (a) {return a * a});
squ = arr.map(a => a * a);
```

Arrow functions: lexical `this`, no more `that=this`

```
function UiComponent {
    var that = this;
    var button = document.getElementById('myButton');
    button.addEventListener('click', function () {
        console.log('CLICK');
        that.handleClick();
    });
}
UiComponent.prototype.handleClick = function () { ... };
```

```
function UiComponent {
    let button = document.getElementById('myButton');
    button.addEventListener('click', () => {
        console.log('CLICK');
        this.handleClick();
    });
}
```

Arrow functions: versions

```
(arg1, arg2, ···) => expr
(arg1, arg2, ···) => { stmt1; stmt2; ··· }

singleArg => expr
singleArg => { stmt1; stmt2; ··· }
```

Exercises

Classes

Classes

```
class Person {
    constructor(name) {
        this.name = name;
    }
    describe() {
        return 'Person called '+this.name;
    }
}
```

```
function Person(name) {
    this.name = name;
}
Person.prototype.describe = function () {
    return 'Person called '+this.name;
};
```

Subclassing

```
class Employee extends Person {
    constructor(name, title) {
        super(name);
        this.title = title;
    }
    describe() {
        return super.describe() + ' (' + this.title + ')';
    }
}
```

```
function Employee(name, title) {
    Person.call(this, name);
    this.title = title;
}
Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;
Employee.prototype.describe = function () {
    return Person.prototype.describe.call(this)
           + ' (' + this.title + ')';
};
```

Why I recommend classes

Pros:

- Code more portable
- Tool support (IDEs, type checkers, …)
- Foundation for (longer term):
  - immutable objects
  - value objects
  - traits (similar to mixins)
- Subclassing built-ins
- Help some newcomers

Cons:

- Syntax quite different from semantics
  - Based on constructors, not prototype chains (directly)

Template literals and tagged templates

Template literals

```
// String interpolation
if (x > MAX) {
    throw new Error(
        `At most ${MAX} allowed: ${x}!`
        // 'At most '+MAX+' allowed: '+x+'!'
    );
}

// Multiple lines
let str = `this is
a text with
multiple lines`;
```

Tagged templates = function calls

Usage:

```
tagFunction`Hello ${first} ${last}!`
```

Syntactic sugar for:

```
tagFunction(['Hello ', ' ', '!'], first, last)
```

Two kinds of tokens:

- Template strings (static): `'Hello '`
- Substitutions (dynamic): `first`

Template literals/tagged templates vs. templates

Different (despite names and appearances being similar):

- Web templates (data): HTML with blanks to be filled in

- Template literals (code): multi-line string literals plus interpolation

- Tagged templates (code): function calls

Tagged templates: XRegExp

XRegExp library: ignored whitespace, named groups, comments

```
// ECMAScript 5
var str = '/2012/10/Page.html';
var parts = str.match(XRegExp(
    '^ # match at start of string only \n' +
    '/ (?<year> [^/]+ ) # capture top dir as year \n' +
    '/ (?<month> [^/]+ ) # capture subdir as month \n' +
    '/ (?<title> [^/]+ ) # file name base \n' +
    '\\.html? # file name extension: .htm or .html \n' +
    '$ # end of string',
    'x'
));
```

Problems:

- Escaping: backslash of string literals vs. backslash of regular expression
- One string literal per line

Tagged templates: XRegExp

```
// ECMAScript 6
let str = '/2012/10/Page.html';
let parts = str.match(XRegExp.rx`
    ^ # match at start of string only
    / (?<year> [^/]+ ) # capture top dir as year
    / (?<month> [^/]+ ) # capture subdir as
month
    / (?<title> [^/]+ ) # file name base
    \.html? # file name extension: .htm or .html
    $ # end of string
`);
console.log(parts.year); // 2012
```

Maps and Sets

Maps

Dictionaries from arbitrary values to arbitrary values.

```
let map = new Map();
let obj = {};

map.set(obj, 123);
console.log(map.get(obj)); // 123
console.log(map.has(obj)); // true

map.delete(obj);
console.log(map.has(obj)); // false

for (let [key,value] of map) {
    console.log(key, value);
}
```

Sets

A collection of values without duplicates.

```
let set = new Set();
set.add('hello');
console.log(set.has('hello')); // true
console.log(set.has('world')); // false

// Sets are iterable

let unique = [...new Set([3,2,1,3,2,3])];
             // [3,2,1]

for (let elem of set) {
    console.log(elem);
}
```

WeakMaps for private data

```
let _counter = new WeakMap();
let _action = new WeakMap();
class Countdown {
    constructor(counter, action) {
        _counter.set(this, counter);
        _action.set(this, action);
    }
    dec() {
        let counter = _counter.get(this);
        if (counter < 1) return;
        counter--;
        _counter.set(this, counter);
        if (counter === 0) {
            _action.get(this)();
        }
    }
}
```

Iteration and loops

Iterable data sources

- Arrays
- Strings
- Maps
- Sets
- `arguments`
- DOM data structures (work in progress)

Not: plain objects

Iterating constructs

- Destructuring via array pattern
- `for-of` loop
- `Array.from()`
- Spread operator (`...`) in arrays and function calls
- Constructor argument of Maps and Sets
- `Promise.all()`, `Promise.race()`
- `yield*`

`for-of`: a better loop

Replaces:

- `for-in`
- `Array.prototype.forEach()`

Works for: iterables

- Convert array-like objects via `Array.from()`.

`for-of` loop: arrays

```
let arr = ['hello', 'world'];
for (let elem of arr) {
    console.log(elem);
}
// Output:
// hello
// world
```

`for-of` loop: arrays

```
let arr = ['a', 'b', 'c'];
for (let [index, elem] of arr.entries()) {
    console.log(`${index}. ${elem}`);
}
// Output
// 0. a
// 1. b
// 2. c
```

`for-of` loop: other iterables

```
let set = new Set(['hello', 'world']);
for (let elem of set) {
    console.log(elem);
}
// Output:
// hello
// world
```

Exercises

Symbols

Symbols

A new kind of primitive value – unique IDs:

```
> let sym = Symbol();
> typeof sym
'symbol'
```

Symbols: enum-style values

```
const COLOR_RED    = Symbol();
const COLOR_ORANGE = Symbol();
...

function getComplement(color) {
    switch (color) {
        case COLOR_RED:
            return COLOR_GREEN;
        case COLOR_ORANGE:
            return COLOR_BLUE;
        ...
        default:
            throw new Exception('Unknown color: '+color);
    }
}
```

Symbols: property keys

```
let specialMethod = Symbol();

obj[specialMethod] = function (arg) {
    ...
};
obj[specialMethod](123);
```

Symbols: property keys

- **Important** advantage: No name clashes!
    - Separate levels of method keys:
      app vs. framework
- Configure objects for ECMAScript and frameworks:
    - Introduce publicly known symbols.
    - Example: property key `Symbol.iterator` makes objects iterable.

Iteration API

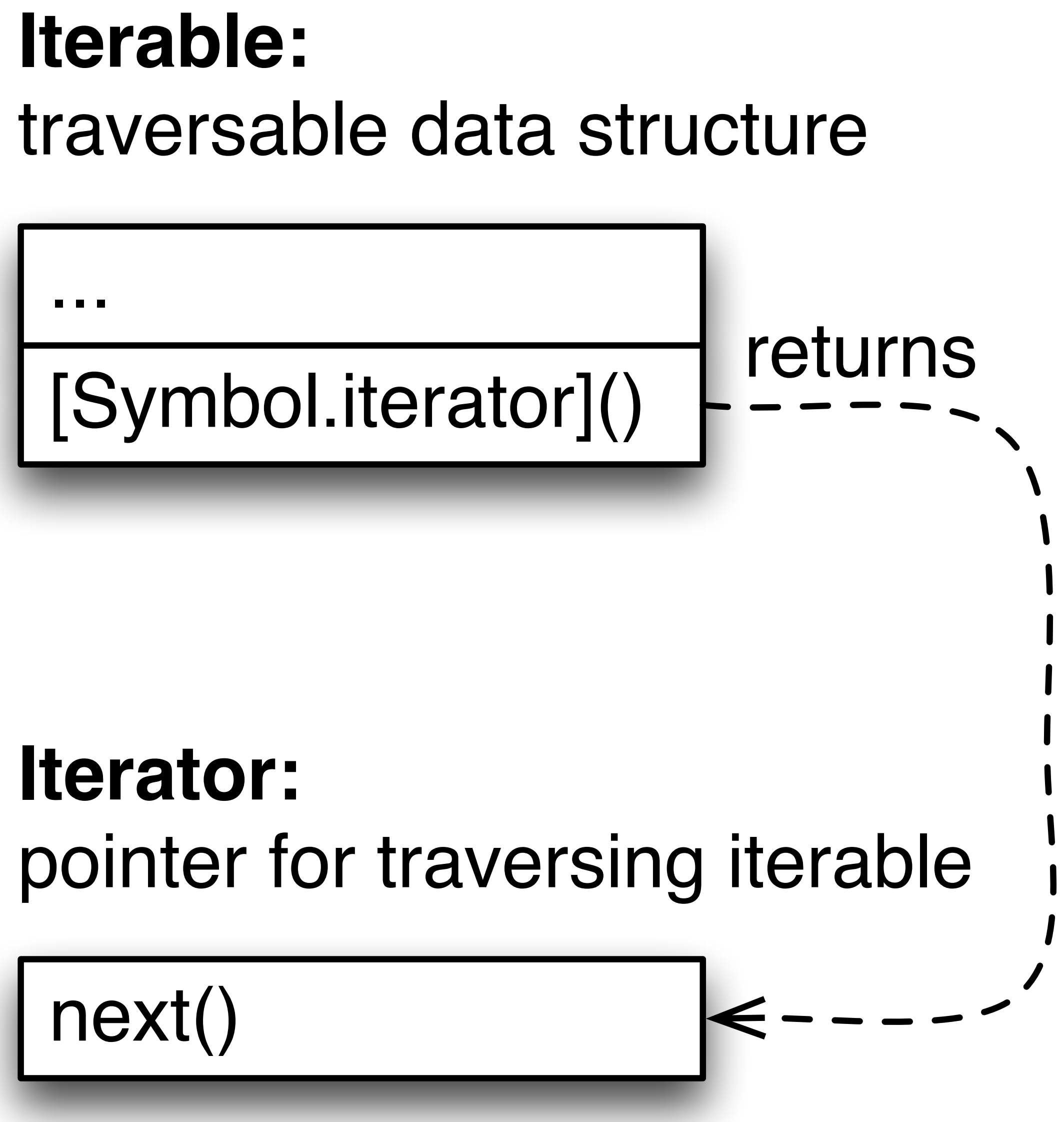
Iterables and iterators

Iteration protocol:

- **Iterable:** a data structure whose elements can be traversed
- **Iterator:** the pointer used for traversal

Examples of iterables:

- Arrays
- Sets
- `arguments`
- All array-like DOM objects (eventually)



Iterables and iterators

```
function iterateOver(...values) {
  let index = 0;
  let iterable = {
    [Symbol.iterator]() {
      let iterator = {
        next() {
          if (index < values.length) {
            return { value: values[index++] };
          } else {
            return { done: true };
          }
        }
      }
      return iterator;
    }
  }
  return iterable;
}
for (let x of iterateOver('eeny', 'meeny', 'miny')) {
  console.log(x);
}
```

Generators

Generators

```
// Suspend via `yield` (“resumable return”):
function* generatorFunction() {
    yield 0;
    yield 1;
    yield 2;
}

// Start and resume via `next()`:
let genObj = generatorFunction();
genObj.next();  // { value: 0, done: false }
genObj.next();  // { value: 1, done: false }
genObj.next();  // ( value: 2, done: false }
genObj.next();  // ( value: undefined, done: true }
```

Generators: implementing an iterator

```
function iterateOver(...vs) {
  let index = 0;
  let iterable = {
    [Symbol.iterator]() {
      let iterator = {
        next() {
          if (index < vs.length) {
            return {value: vs[index++]};
          } else {
            return {done: true};
          }
        }
      }
      return iterator;
    }
  }
  return iterable;
}
```

```
function iterateOver(...vs) {
  let iterable = {
    * [Symbol.iterator]() {
      for(let v of vs) {
        yield v;
      }
    }
  }
  return iterable;
}
```

Generators: implementing an iterator

```
function* objectEntries(obj) {
   for (let key of Object.keys(obj)) {
       yield [key, obj[key]];
    }
}
let myObj = { foo: 3, bar: 7 };
for (let [key, value] of objectEntries(myObj)) {
    console.log(key, value);
}
// Output:
// foo 3
// bar 7
```

An iterator for a tree (1/2)

```
class BinaryTree {
    constructor(value, left=null, right=null) {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    /** Prefix iteration */
    * [Symbol.iterator]() {
        yield this.value;
        if (this.left) {
            yield* this.left;
        }
        if (this.right) {
            yield* this.right;
        }
    }
}
```

An iterator for a tree (2/2)

```
let tree = new BinaryTree('a',
    new BinaryTree('b',
        new BinaryTree('c'),
        new BinaryTree('d')),
    new BinaryTree('e'));

for (let x of tree) {
    console.log(x);
}
// Output:
// a
// b
// c
// d
// e
```

Asynchronous programming (1/2)

```
co(function* () {
    try {
        let [croftStr, bondStr] = yield Promise.all([
            getFile('http://localhost:8000/croft.json'),
            getFile('http://localhost:8000/bond.json'),
        ]);
        let croftJson = JSON.parse(croftStr);
        let bondJson = JSON.parse(bondStr);

        console.log(croftJson);
        console.log(bondJson);
    } catch (e) {
        console.log('Failure to read: ' + e);
    }
});
```

Asynchronous programming (2/2)

```
function getFile(url) {
    return fetch(url)
        .then(request => request.text());
}
```

Promises

First example: Node.js

```
fs.readFile('config.json',
    function (error, text) {
        if (error) {
            console.error('Error');
        } else {
            try {
                var obj = JSON.parse(text);
                console.log(JSON.stringify(obj, null, 4));
            } catch (e) {
                console.error('Invalid JSON in file');
            }
        }
    });
```

First example: Promises

```
readFilePromisified('config.json')
.then(function (text) {
    var obj = JSON.parse(text);
    console.log(JSON.stringify(obj, null, 4));
})
.catch(function (reason) {
    // File read error or JSON SyntaxError
    console.error('Error', reason);
});
```

The basics

```
var promise = new Promise(
    function (resolve, reject) {
        resolve(value); // success
        reject(error); // failure
    });

promise.then(
    function (value) { /* success */ },
    function (error) { /* failure */ }
);
```

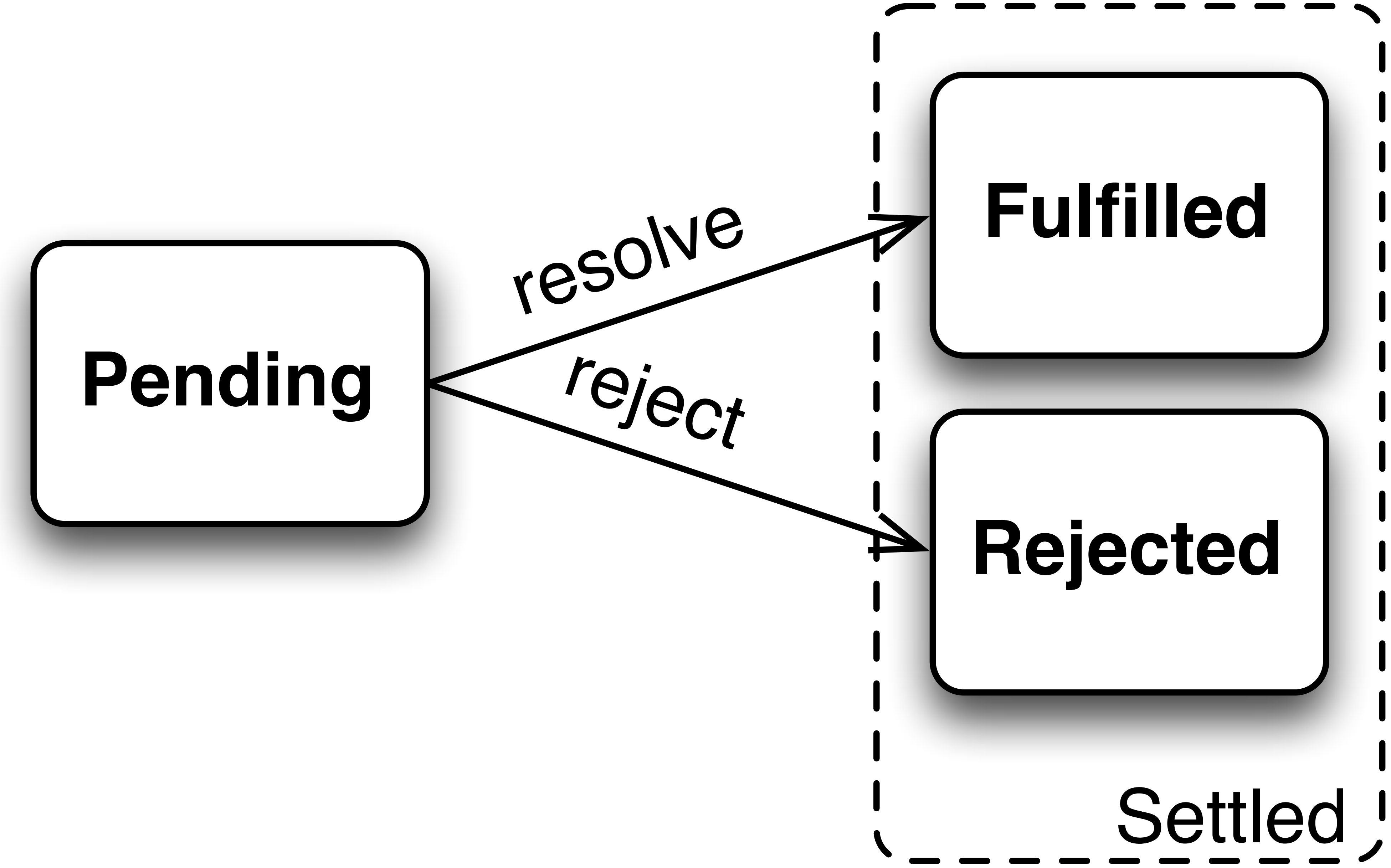
The basics

```
// Equivalent:

promise.catch(
    function (error) { /* failure */ }
);

promise.then(
    null,
    function (error) { /* failure */ }
);
```

States of promises (simplified)



Error handling

```
retrieveFileName()
.catch(function () {
    // Something went wrong, use default
    return 'Untitled.txt';
})
.then(function (fileName) {
    ...
});
```

```
function httpGet(url) {
    return new Promise(
        function (resolve, reject) {
            var request = new XMLHttpRequest();
            request.onreadystatechange = function () {
                if (this.status === 200) {
                    resolve(this.response); // Success
                } else {
                   reject(new Error(this.statusText));
                }
            }
            request.onerror = function () {
                reject(new Error(
                    'Error: '+this.statusText));
            };
            request.open('GET', url);
            request.send();
        }); }
```

.map()

```
var fileUrls = [
    'http://example.com/file1.txt',
    'http://example.com/file2.txt'
];
var promisedTexts = fileUrls.map(httpGet);

Promise.all(promisedTexts)
.then(function (texts) {
    texts.forEach(function (text) {
        console.log(text);
    });
})
.catch(function (reason) {
    // Receives first rejection among the promises
});
```

Advantages of promises

- Cleaner signature
- Chaining results, while avoiding nesting
- Better error handling (chaining, catching exceptions, etc.)
- Easier to compose (fork-join, map, etc.)

Tail call optimization

Tail calls

- Tail call: a function call that is the last action inside a function.

- Can be implemented as goto (vs. gosub which needs call stack space).

Tail calls: examples

```
function f(x) {
    g(x); // no tail call
    h(x); // tail call
}

function f(x) {
    return g(x); // tail call
}
function f(x) {
    return 1+g(x); // no tail call
}
function f(x) {
    if (x > 0) {
        return g(x); // tail call
    }
    return h(x); // tail call
}
```

Tail recursion

```
/** Not tail recursive */
function fac(x) {
    if (x <= 0) {
        return 1;
    } else {
        return x * fac(x-1);
    }
}
/** Tail recursive */
function fac(n) {
    return facRec(n, 1);
    function facRec(i, acc) {
        if (i <= 1) {
            return acc;
        } else {
            return facRec(i-1, i*acc);
        }
    }
}
```

Loops via recursion

```
function forEach(arr, callback, start = 0) {
  if (0 <= start && start < arr.length) {
    callback(arr[start]);
    forEach(arr, callback, start+1); // tail call
  }
}
```

More standard library

`Object.assign()`

`Object.assign(target, source_1, source_2, ···)`

- Merge `source_1` into `target`
- Merge `source_2` into `target`
- Etc.
- Return `target`

Warning:

- sets properties (may invoke setters),
- does not define them (always new)

ES6 versus lodash

```
let obj = { foo: 123 };

// ECMAScript 6
Object.assign(obj, { bar: true });
    // obj is now { foo: 123, bar: true }

// lodash/Underscore.js
_.extend(obj, { bar: true })
    // also: _.assign(···)
```

Object.assign()

```
// Provide default values if properties are missing
const DEFAULTS = {
    logLevel: 0,
    outputFormat: 'html'
};
function processContent(options) {
    options = Object.assign({}, DEFAULTS, options);
    ...
}
```

Object.assign()

```
// A quick and somewhat dirty way to
// add methods to a prototype object
Object.assign(SomeClass.prototype, {
    someMethod(arg1, arg2) { ··· },
    anotherMethod() { ··· },
});

// Without Object.assign():
SomeClass.prototype.someMethod =
    function (arg1, arg2) { ··· };
SomeClass.prototype.anotherMethod =
    function () { ··· };
```

New string methods

```
> 'abc'.repeat(3)
'abcabcabc'
> 'abc'.startsWith('ab')
true
> 'abc'.endsWith('bc')
true
> 'foobar'.includes('oo')
true
```

New array methods

```
> [13, 7, 8].find(x => x % 2 === 0)
8
> [1, 3, 5].find(x => x % 2 === 0)
undefined

> [13, 7, 8].findIndex(x => x % 2 === 0)
2
> [1, 3, 5].findIndex(x => x % 2 === 0)
-1
```

Conclusion

Various other features

Also part of ECMAScript 6:

- Proxies (meta-programming)
- Better support for Unicode (strings, regular expressions)

Transpilation

Things that can’t be transpiled at all:

- Proxies
- Subclassable built-in constructors (`Error`, `Array`, …)
- Tail call optimization

Things that are difficult to transpile:

- Symbols (via objects)
- Generators (transformed to state machines)
- WeakMaps, WeakSets (keys stored in values)

Take-aways: ECMAScript 6

- Many features are already in engines
- Can be used today, by compiling to ECMAScript 5
- Biggest impact on community (currently: fragmented):
  - Classes
  - Modules

© Scott the Hobo

Thank you!

Free online book by Axel: "Exploring ES6"