Using ECMAScript 6 today

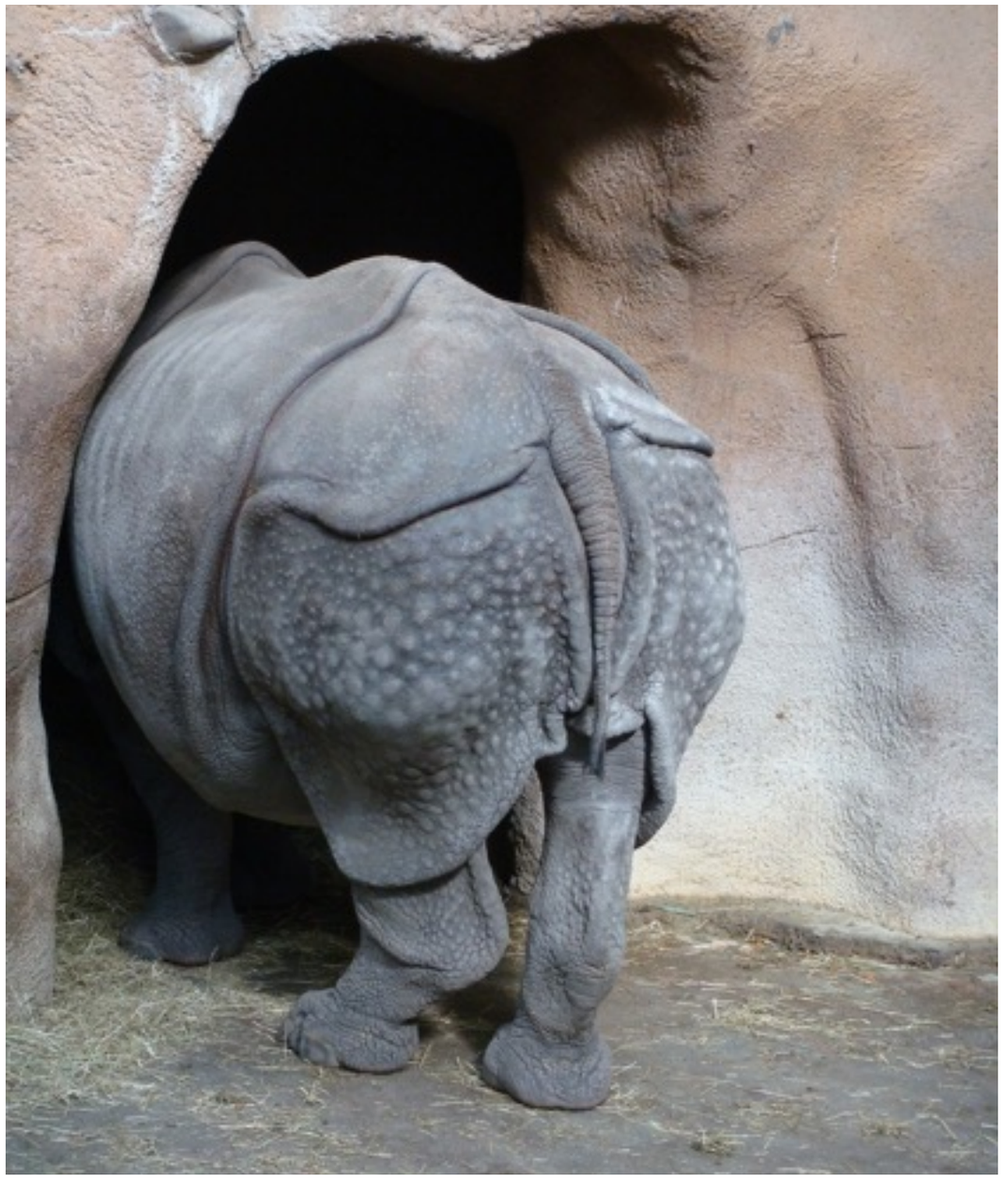
Rolling Scopes Conference, 2015-02-01

Slides: speakerdeck.com/rauschma

JavaScript is everywhere

- In browsers, servers, devices, robots, …
- Used for much more than originally created for
- How can it be improved?

© Schmeegan



ECMAScript 6 (ES6): JavaScript, improved

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

This talk:

- Goals
- Design process
- Features
- When can I use it?

Background

Important ES terms

- **TC39 (Ecma Technical Committee 39):** the committee evolving JavaScript.
  - Members: companies (all major browser vendors etc.).
  - Meetings attended by employees and invited experts.
- **ECMAScript:** the official name of the language
  - Versions: ECMAScript 5 is short for “ECMAScript Language Specification, Edition 5”
- **JavaScript:**
  - colloquially: the language
  - formally: one implementation of ECMAScript
- **ECMAScript Harmony:** improvements after ECMAScript 5 (ECMAScript 6 and 7)
- **ECMAScript 2015:** New official name for ES6 (will take a while to spread)

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:

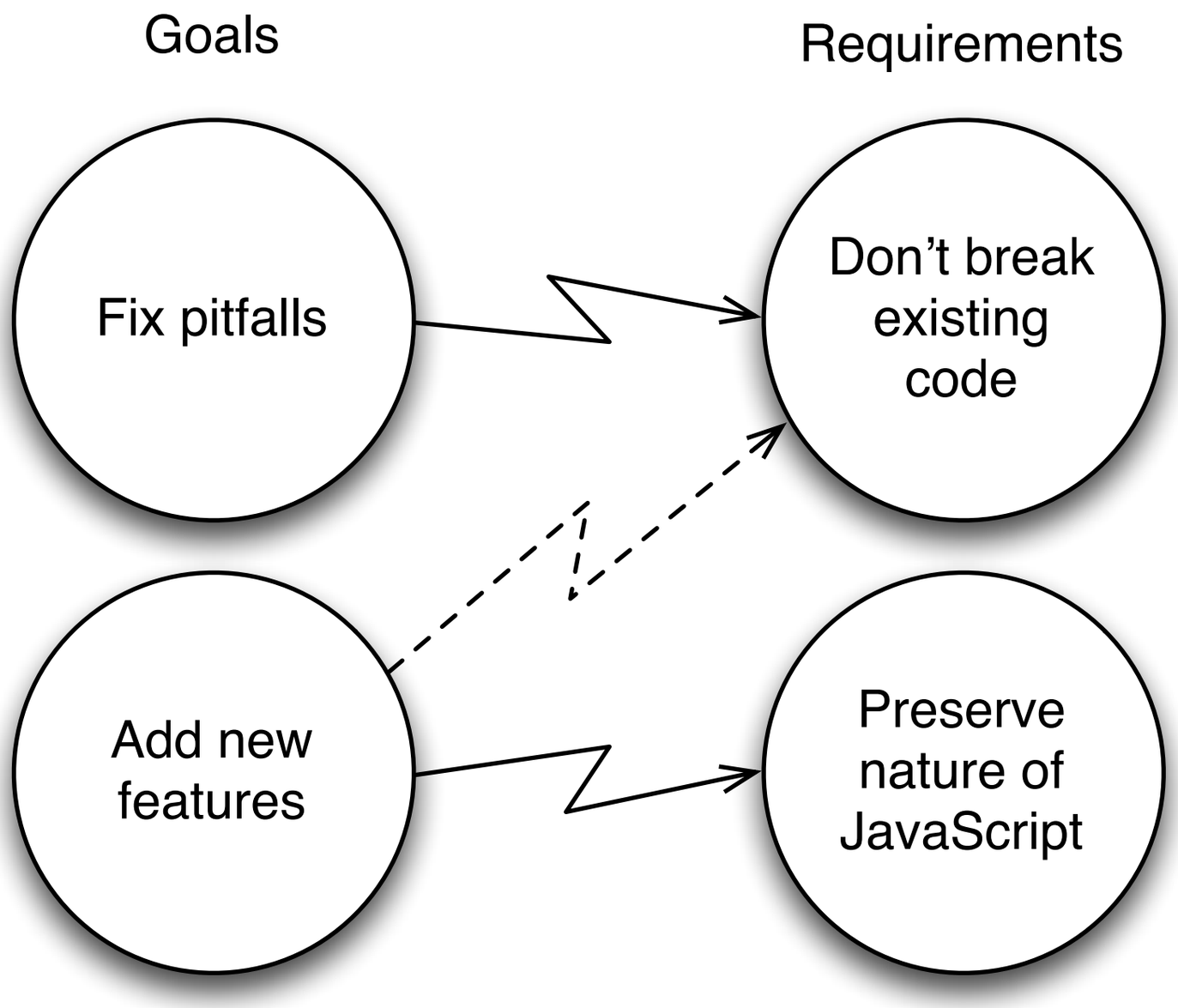
- JavaScript engines:
    - New versions = forced upgrades
    - Must run all existing code

  ⇒ ECMAScript 6 only adds features

- 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

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

```
// Multiple 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'
```

To be used:

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

Multiple return values

```
// { x, y } is abbreviation for { x: x, y: y }

function findElement(arr, predicate) {
    for (let index=0; index < arr.length; index++) {
        let element = arr[index];
        if (predicate(element)) {
            return { element, 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(···);
```

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');
```

Parameter handling

Parameter default values

Use a default if a parameter is missing.

```
function func1(x, y=3) {
    return [x,y];
}
```

Interaction:

```
> func1(1, 2)
[1, 2]
> func1(1)
[1, 3]
> func1()
[undefined, 3]
```

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.

Spread operator (...)

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

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

Turn an array into function/method arguments:

- The inverse of rest parameters
- Mostly replaces `Function.prototype.apply()`
- Also works in constructors

Named parameters

```
// Emulate named parameters
// via object literals and destructuring

function func(arg0, { opt1, opt2 }) {
    return [opt1, opt2];
}
// { opt1, opt2 } is abbreviation for
// { opt1: opt1, opt2: opt2 }

func(0, { opt1: 'a', opt2: 'b' });
    // ['a', 'b']
```

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; ··· }
```

Object literals

Method definitions

```
let obj = {
    myMethod() {
        ...
    }
};
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 = 'foo';
let obj1 = {
    [propKey]: true,
    ['b'+'ar']: 123
};

let obj2 = {
    ['h'+'ello']() {
        return 'hi';
    }
};
console.log(obj.hello()); // hi
```

Computed property keys (2/2)

```
let obj = {
    // Generator method whose key is a symbol
    * [Symbol.iterator]() {
        yield 'hello';
        yield 'world';
    }
};
for (let x of obj) {
    console.log(x);
}
// Output:
// hello
// world
```

Classes

Classes

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() {
        return '('+this.x+', '+this.y+')';
    }
}
```

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}
Point.prototype.toString = function () {
    return '('+this.x+', '+this.y+')';
};
```

Subclassing

```
class ColorPoint extends Point {
    constructor(x, y, color) {
        super(x, y);
        this.color = color;
    }
    toString() {
        return this.color+' '+super.toString();
    }
}
```

```
function ColorPoint(x, y, color) {
    Point.call(this, x, y);
    this.color = color;
}
ColorPoint.prototype = Object.create(Point.prototype);
ColorPoint.prototype.constructor = ColorPoint;
ColorPoint.prototype.toString = function () {
    return this.color+' '+Point.prototype.toString.call(this);
};
```

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';
```

Template strings

Untagged template strings

```
// 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 template strings = function calls

Usage:

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

Syntactic sugar for:

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

Two kinds of tokens:

- Literal sections (static): `'Hello'`
- Substitutions (dynamic): `first`

Template strings: XRegExp

XRegExp library – ignored whitespace, named groups, comments

```
// 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
```

Advantages:

- Raw characters: no need to escape backslash and quote
- Multi-line: no need to concatenate strings with newlines at the end

Template strings: XRegExp

```
// 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'
));
```

Template strings: other use cases

Great for building embedded DSLs:

- Query languages
- Text localization
- Templating
- etc.

Standard library

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

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

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

`Object.assign()`

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

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

```
let obj = { foo: 123 };
Object.assign(obj, { bar: true });
  // obj is now { foo: 123, bar: true }
```

Object.assign()

```
class Point {
    constructor(x, y) {
        Object.assign(this, {x, y});
    }
}
```

Object.assign()

```
const DEFAULTS = {
    logLevel: 0,
    outputFormat: 'html'
};
function processContent(options) {
    options = Object.assign({}, DEFAULTS, options);
    ...
}
```

Object.assign()

```
Object.assign(SomeClass.prototype, {
    someMethod(arg1, arg2) { ··· },
    anotherMethod() { ··· },
});

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

Object.assign()

```
function clone(orig) {
    return Object.assign({}, orig);
}
```

New string methods

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

Symbols

Symbols

A new kind of primitive value – unique IDs:

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

Symbols: enum-style values

```
const red = Symbol();
const green = Symbol();
const blue = Symbol();

function handleColor(color) {
    switch(color) {
        case red:
            ...
        case green:
            ...
        case blue:
            ...
    }
}
```

Symbols: property keys

```
let specialMethod = Symbol();
let obj = {
    // computed property key
    [specialMethod]: function (arg) {
        ...
    }
};
obj[specialMethod](123);

// Shorter -- method definition syntax:
let obj = {
    [specialMethod](arg) {
        ...
    }
};
```

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.

Loops and iteration

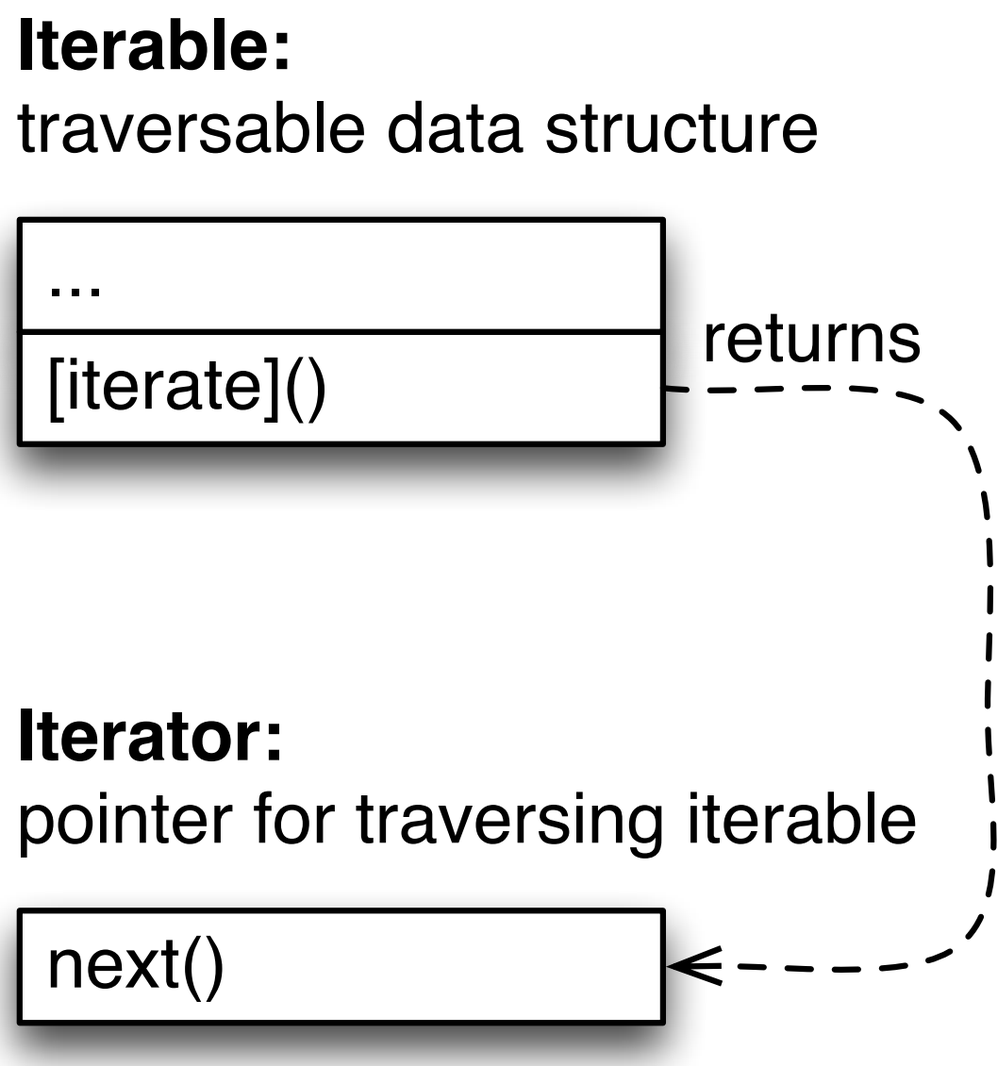
Iterables and iterators

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);
}
```

`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 = ['hello', 'world'];
for (let [index, elem] of arr.entries()) {
    console.log(index, elem);
}
// Output:
// 0 hello
// 1 world
```

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* iterEntries(obj) {
    let keys = Object.keys(obj);
    for (let i=0; i < keys.length; i++) {
        let key = keys[i];
        yield [key, obj[key]];
    }
}
let myObj = { foo: 3, bar: 7 };
for (let [key, value] of iterEntries(myObj)) {
    console.log(key, value);
}
// Output:
// foo 3
// bar 7
```

Generators: asynchronous programming

Using the Q promise library:

```
Q.spawn(function* () {
    try {
        let [foo, bar] = yield Promise.all(
                [read('foo.json'), read('bar.json')] );
        render(foo);
        render(bar);
    } catch (e) {
        console.log('Failure to read: ' + e);
    }
});

// Wait for asynchronous calls via `yield`
// (internally based on promises)
```

When?

Various other features

Also part of ECMAScript 6:

- Promises
- Proxies (meta-programming)
- Better support for Unicode (strings, regular expressions)
- Tail call optimization

Time table

ECMAScript 6 is basically done:

- Its feature set is frozen.
- It is mostly being refined now.
- Features are continually appearing in current engines [4].

Time table:

- End of 2014: specification is finished (except fixing last bugs)
- March 2015: publication process starts
- June 2015: formal publication

ES6 today: compile ES6 to ES5

Three important ones:

- TypeScript: ECMAScript 6 plus (optional) type annotations
- Traceur
- 6to5

Traceur and 6to5

Run...

- Statically: via build tools (Grunt, Gulp, Broccoli, etc.)
- Dynamically (include library, use `<script type="module">`)

Generate three kinds of modules: TODO

- ES6 module loader API (via shim on ES5)
- AMD (RequireJS)
- CommonJS (Node.js, browserify, webpack)

ES6 today: module systems

- Browserify: transforms for Traceur and 6to5
- webpack: loaders for Traceur and 6to5
- ES6 Module Loader Polyfill: based on ES6, for Node.js and browsers. Complemented by:
    - SystemJS: additionally loads AMD and CommonJS
    - jspm: package manager for SystemJS

TODO: links

ES6 today: shims

ES6 standard library, backported to ES5:

- es6-shim: most of the ES6 standard library
- Core.js: polyfill for ES5/ES6 standard library. Used by 6to5.

Shims for ES6 promises:

- RSVP.js: superset of ES6 API
  - es6-promise: only ES6 API (subset of RSVP.js)
- Q.Promise: is compatible with ES6

Conclusion

Take-aways: ECMAScript 6

- Many features are already in engines [4]
- Can be used today, by compiling to ECMAScript 5
- Biggest impact on community (currently: fragmented):
    - Classes
    - Modules

© Scott the Hobo

Thank you!

Early draft of upcoming book: “Using ECMAScript 6 today”
www.2ality.com/2014/08/es6-today.html