ECMAScript 6

Dr. Axel Rauschmayer

rauschma.de

2013-03-30

CodeFest 2013, Novosibirsk

About me

Axel Rauschmayer:

- Editor of JavaScript Weekly
- Author of 2ality.com
- Co-organizer of MunichJS

I have written about ECMAScript.next/ECMAScript 6 since early 2011.

JavaScript: it has become big

Photographer: Via Tsuji

Used for much more than it was originally created for.

ECMAScript 6

ECMAScript 6:

- Next version of JavaScript
  (current engines: ECMAScript 5).
- Be better at what JavaScript is used for now.

This talk:

- Specific goals for ECMAScript 6?
- How is ECMAScript 6 created?
- Features?

Warning

All information in this talk is preliminary, features can and will change before ECMAScript 6 is final.

Background

Glossary

- **TC39 (Ecma Technical Committee 39):** the committee evolving JavaScript.
  - Members: companies (all major browser vendors etc.).
  - Meetings attended by employees and invited experts.
- **JavaScript:** colloquially: the language; formally: one implementation
  - **ECMAScript:** the language standard
- **ECMAScript Harmony:** improvements after ECMAScript 5 (several versions)
  - **ECMAScript.next:** code name for upcoming version, subset of Harmony
  - **ECMAScript 6:** the final name of ECMAScript.next (probably)

Goals for ECMAScript 6

One of several goals [1]: make JavaScript better

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

Challenges of evolving a web language

Little control over ECMAScript versions:

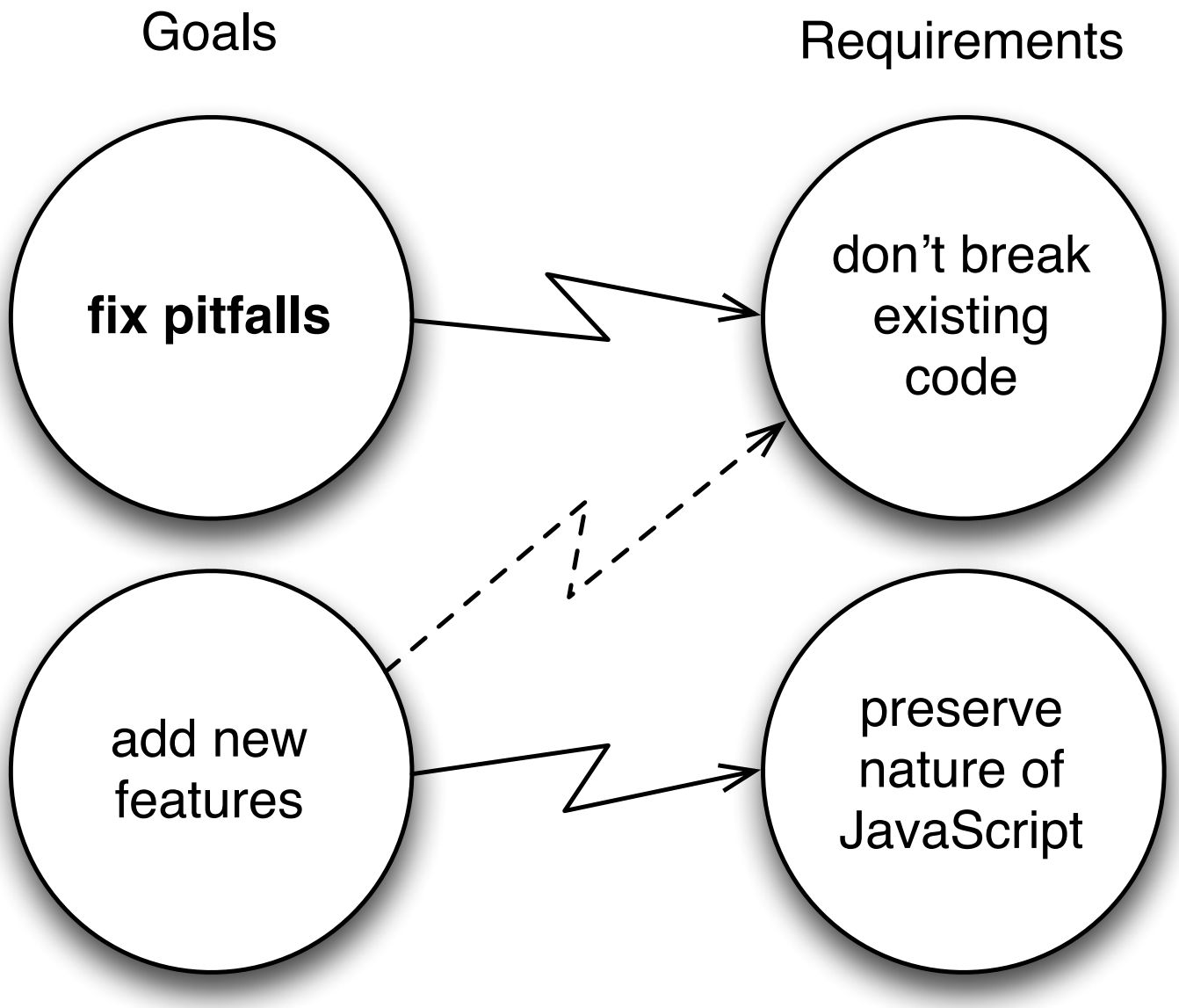
1. Browsers: encounter code in many different versions.
   - Even within a single app (third party code!).
2. Apps: encounter engines supporting a variety of versions.

Constraints for ECMAScript 6:

#1 ⇒ Must not break existing code.

#2 ⇒ Can't be used right away (not everywhere).

Challenges of evolving a web language



How features are designed

Avoid "design by committee":

- Design by "champions" (groups of 1–2 experts)
- Feedback from TC39 and the web community
- TC39 has final word on whether to include

Stages [2]:

- Strawman proposal
- TC39 is interested ⇒ proposal
- Field-testing via one or more implementations, refinements
- TC39 accepts feature ⇒ included in ECMAScript draft
- Included in final spec ⇒ Standard

Variables and scoping

Block-scoped variables

Function scope (var)

```
function order(x, y) {
    console.log(tmp);
        // undefined

    if (x > y) {
        var tmp = x;
        x = y;
        y = tmp;
    }
    return [x, y];
}
```

Block scope (let, const)

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

Destructuring: objects

Extract data (more than one value!) via patterns:

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

let { first: f, last: l } = obj;
console.log(f + ' ' + l);  // Jane Doe

let { first, last } = obj;
    // same as { first: first, last: last }
console.log(first + ' ' + last);  // Jane Doe
```

Usage: variable declarations, assignments, parameter definitions.

Destructuring: arrays

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

[x,y] = [y,x];
```

Destructuring: refutable by default

- **Refutable (default):** exception if match isn't exact.

```
{ a: x, b: y }   = { a: 3 }; // fails
```

- **Irrefutable:** always match.

```
{ a: x, ?b: y }  = { a: 3 }; // x=3, y=undefined
```

- **Default value:** use if no match or value is `undefined`

```
{ a: x, b: y=5 } = { a: 3 }; // x=3, y=5
```

Functions and parameters

Arrow functions: less to type

Compare:

```
let squares = [1, 2, 3].map(x => x * x);
```

```
let squares = [1, 2, 3].map(function (x) {return x * x});
```

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: variants

Zero parameters:

```
() => expr
() => { stmt0; stmt1; ... }
```

One parameter:

```
arg => expr
arg => { stmt0; stmt1; ... }
```

More than one parameter:

```
(arg0, arg1, ...) => expr
(arg0, arg1, ...) => { stmt0; stmt1; ... }
```

Parameter handling 1: parameter default values

Give missing parameters a default value.

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

Interaction:

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

Parameter handling 2: rest parameters

Put trailing parameters in an array.

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

Interaction:

```
> func2(0, 1, 2, 3)
[1, 2, 3]
> func2(0)
[]
> func2()
[]
```

Eliminate the need for the special variable `arguments`.

Parameter handling 3: named parameters

Idea: name parameters via an object literal.

- More descriptive:

```
moveTo({ x: 50, y: 50, speed: 0.5 })
```

- Easy to omit:

```
foo({ opt1: 'a', opt2: 'b', opt3: 'c' })
foo({ opt3: 'c' })
foo({ opt1: 'a', opt3: 'c' })
```

Parameter handling 3: named parameters

Use destructuring for named parameters `opt1` and `opt2`:

```
function func3(arg0, { opt1, opt2 }) {
    return return [opt1, opt2];
}
```

Interaction:

```
> func3(0, { opt1: 'a', opt2: 'b' })
['a', 'b']
```

Spread operator (...)

Turn an array into function/method arguments:

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

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

Modularity

Object literals

```
// ECMAScript 6
let obj = {
    __proto__: someObject,  // special property
    myMethod(arg1, arg2) {  // method definition
        ...
    }
};
```

```
// ECMAScript 5
var obj = Object.create(someObject);
obj.myMethod = function (arg1, arg2) {
    ...
};
```

Object literals: property value shorthand

Shorthand: `{x,y}` is the same as `{ x: x, y: y }`.

```
function computePoint() {
    let x = computeX();
    let y = computeY();
    return { x, y };  // shorthand
}

let {x,y} = computePoint();  // shorthand
```

Symbols

Each symbol is a unique value. Use as

- enum-style values
- property keys:

```
let sym = Symbol();
console.log(typeof sym); // symbol
let obj = {
    [sym](arg) {  // computed property key
        ...
    }
};
obj[sym](123);
```

Advantages of symbols as property keys:

- No name clashes!
- Configure various aspects of the language
  $\Rightarrow$ import public symbols from a system module

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+')';
};
```

Classes: sub-type

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

```
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);
};
```

Static methods

```
class Point {
    static zero() {
        return new Point(0, 0);
    }
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}
let p = Point.zero();
```

Private properties

Hiding some properties from external access (object literals, classes):

- Still under discussion.
- But there will be *some* way of doing it.
- Possibly: a special kind of symbol.

Modules: overview

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

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

Alternatively:

```
import 'lib/math' as math;
console.log(math.square(3));
```

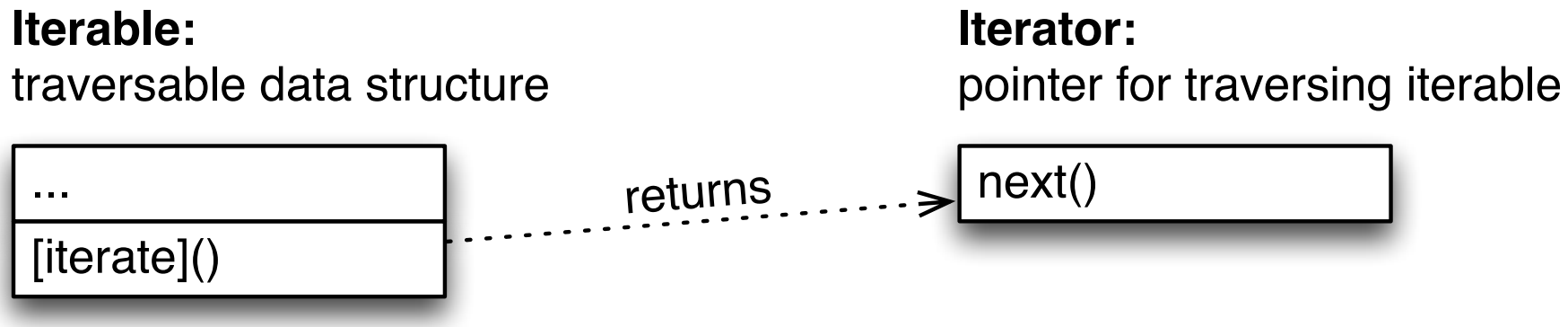
Modules: features

More features [3]:

- Rename imports
- Concatenation: put several modules in the same file
- Module IDs are configurable (default: paths relative to importing file)
- Programmatic (e.g. conditional) loading of modules via an API
- Module loading is customizable:
  - Automatic linting
  - Automatically translate files (CoffeeScript, TypeScript)
  - Use legacy modules (AMD, Node.js)

Loops and iteration

Iterables and iterators



Examples of iterables:

- Arrays
- Results produced by tool functions and methods (`keys()`, `values()`, `entries()`).

Iterators

```
import {iterate} from '@iter';  // symbol
function iterArray(arr) {
    let i = 0;
    return {  // both iterable and iterator
        [iterate]() {  // iterable
            return this;  // iterator
        },
        next() {  // iterator
            if (i < arr.length) {
                return { value: arr[i++] };
            } else {
                return { done: true };
            } } } }
for (let elem of iterArray(['a', 'b'])) {
    console.log(elem);
}
```

for-of: a better loop

- for-in loop:
  - Basically useless for arrays
  - Quirky for objects
- `Array.prototype.forEach()`: doesn't work with iterables.

for-of loop: arrays

Looping over iterables.

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

Output:

```
hello
world
```

for-of loop: objects

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

Iterate over properties:

```
import {entries} from '@iter';  // returns an iterable
for (let [key, value] of entries(obj)) {
    console.log(key + ' = ' + value);
}
```

Iterate over property names:

```
import {keys} from '@iter';  // returns an iterable
for (let name of keys(obj)) {
    console.log(name);
}
```

Template strings

Template strings: string interpolation

Invocation:

```
templateHandler`Hello ${firstName} ${lastName}!`
```

Syntactic sugar for:

```
templateHandler(['Hello ', ' ', '!'], firstName, lastName)
```

Two kinds of tokens:

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

Template strings: raw strings

No escaping, multiple lines:

```
var str = raw`This is a text
with multiple lines.

Escapes are not interpreted,
\n is not a newline.`;
```

Template strings: regular expressions

ECMAScript 5 (XRegExp library):

```
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'
));

console.log(parts.year); // 2012
```

XRegExp features: named groups, ignored whitespace, comments.

Template strings: regular expressions

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
`);
```

Advantages:

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

Template strings: other use cases

- Query languages
- Text localization
- Secure templates
- etc.

Standard library

Maps

Data structure mapping from arbitrary values to arbitrary values (objects: keys must be strings).

```
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
```

Also: iteration (over keys, values, entries) and more.

Weak maps

Idea – a map with weakly held keys:

- Map objects to data.
- Don't prevent objects from being garbage-collected.
- Use cases: privately associating data with objects, private caches, ...

Constraints:

- Can't enumerate contents
- Keys are objects, values are arbitrary values

Sets

A collection of values without duplicates.

```
let set1 = new Set();
set1.add('hello');
console.log(set1.has('hello')); // true
console.log(set1.has('world')); // false

let set2 = new Set([3,2,1,3,2,3]);
console.log(set2.values());  // 1,2,3
```

Object.assign

Merge one object into another one.

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

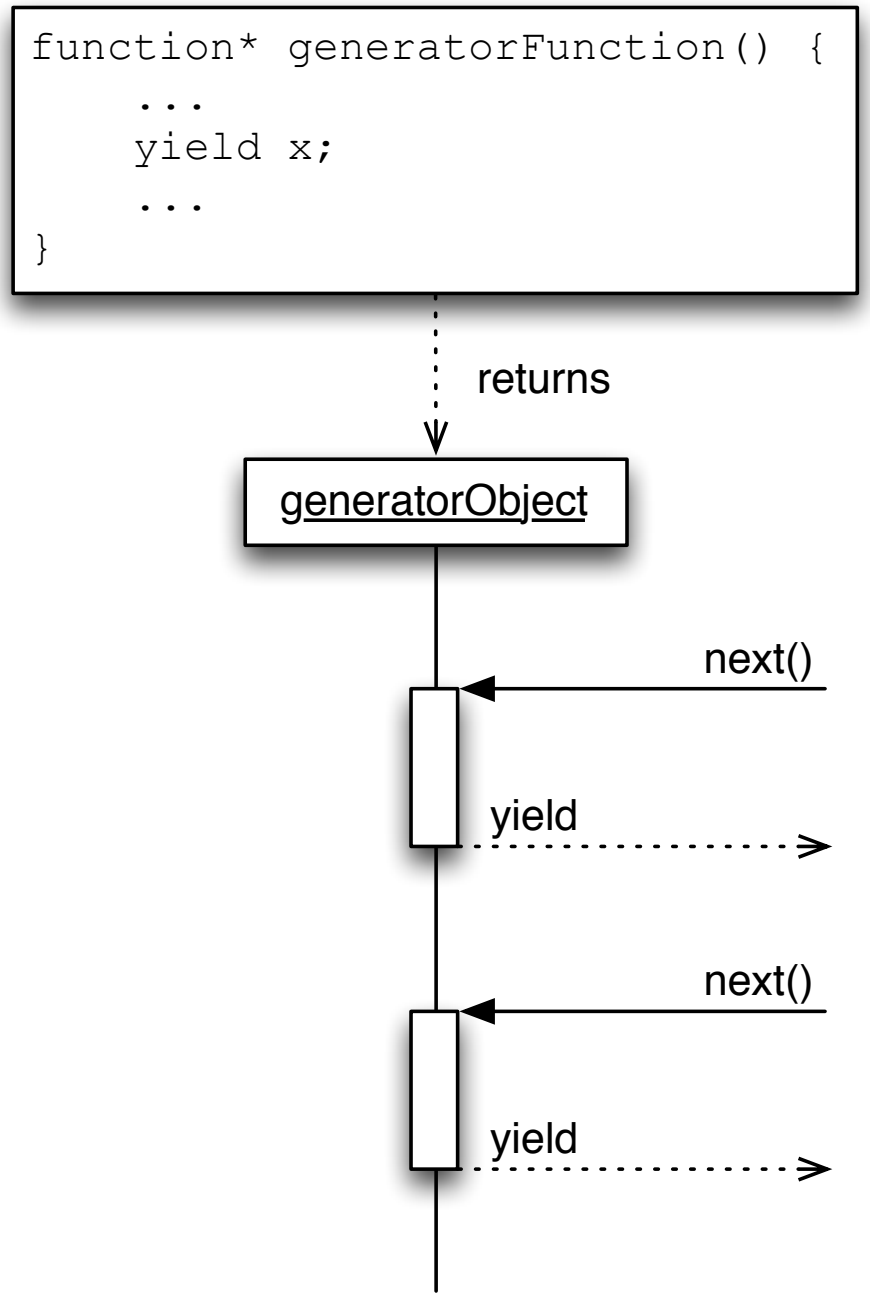
Similar to Underscore.js `_.extend()`.

Various other additions to the standard library

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

And more!

Generators



Generators: suspend and resume a function

- Shallow coroutines [4]: only function body is suspended.
- Uses: iterators, simpler asynchronous programming.

Generators: example

Suspend via `yield` ("resumable return"):

```
function* generatorFunction() {
    yield 0;
    yield 1;
    yield 2;
}
```

Start and resume via `next()`:

```
let genObj = generatorFunction();
console.log(genObj.next());  // 0
console.log(genObj.next());  // 1
console.log(genObj.next());  // 2
```

Generators: implementing an iterator

An iterator for nested arrays:

```
function* iterTree(tree) {
    if (Array.isArray(tree)) {
        // inner node
        for(let i=0; i < tree.length; i++) {
            yield* iterTree(tree[i]); // recursion
        }
    } else {
        // leaf
        yield tree;
    }
}
```

Difficult to write without recursion.

Generators: asynchronous programming

Using the task.js library:

```
spawn(function* () {
    try {
        var [foo, bar] = yield join(
            read("foo.json"), read("bar.json")
        ).timeout(1000);
        render(foo);
        render(bar);
    } catch (e) {
        console.log("read failed: " + e);
    }
});
```

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

Proxies

Proxies

Observe operations applied to object `proxy`, via handler `h`:

```
let target = {};
let proxy = Proxy(target, h);
```

Each of the following operations triggers a method invocation on `h`:

```
proxy['foo']              → h.get(target, 'foo', p)
proxy['foo'] = 123        → h.set(target, 'foo', 123, p)
'foo' in proxy            → h.has(target, 'foo')
for (key in proxy) {...}  → h.enumerate(target)
```

Proxies in the prototype chain

```
let child = Object.create(proxy);
```

Operations on `child` can still trigger handler invocations (if the search for properties reaches `proxy`):

```
child['foo']                → h.get(target, 'foo', ch)
child['foo'] = 123          → h.set(target, 'foo', 123, ch)
'foo' in child              → h.has(target, 'foo')
for (key in child) {...}    → h.enumerate(target)
```

Proxy: example

```
let handler = {
    get(target, name, receiver) {
        return (...args) => {
            console.log('Missing method '+name
                        + ', arguments: '+args);
        }
    }
};
let proxy = Proxy({}, handler);
```

Using the handler:

```
> let obj = Object.create(proxy);
> obj.foo(1, 2)
Missing method foo, arguments: 1, 2
undefined
```

Use cases for proxies

Typical meta-programming tasks:

- Sending all method invocations to a remote object
- Implementing data access objects for a database
- Data binding
- Logging

Conclusion

Time table

ECMAScript specification:

- November 2013: final review of draft
- July 2014: editorially complete
- December 2014: Ecma approval

Features are already appearing in JavaScript engines [5]!

Thank you!

References

1. ECMAScript Harmony wiki
2. “The Harmony Process” by David Herman
3. “ES6 Modules” by Yehuda Katz
4. “Why coroutines won’t work on the web” by David Herman
5. “ECMAScript 6 compatibility table” by kangax [features already in JavaScript engines]

Resources

- ECMAScript 6 specification drafts by Allen Wirfs-Brock
- ECMAScript mailing list: es-discuss
- TC39 meeting notes by Rick Waldron
- "A guide to 2ality's posts on ECMAScript 6" by Axel Rauschmayer
- Continuum, an ECMAScript 6 virtual machine written in ECMAScript 3.

(Links are embedded in this slide.)

Bonus slides

Function-scoping: pitfall

```
function foo() {
    var x = 3;
    if (x >= 0) {
        var tmp;
        console.log(tmp);  // undefined
        tmp = 'abc';
    }
    if (x > 0) {
        var tmp;
        console.log(tmp);  // abc
    }
}
```

Comprehensions

Array comprehensions produce an array:

```
let numbers = [1,2,3];
let squares = [for (x of numbers) x*x];
```

Generator comprehensions produce a generator object (an iterator):

```
let squares = (for (x of numbers) x*x);
```