Callable entities in ECMAScript 6

Dr. Axel Rauschmayer

2ality.com

2014-03-12

Fluent 2014

Now: functions do triple duty

- Normal functions (non-method functions)
- Methods
- Constructors

Problems

1. Confusing: `function` for methods, etc.
2. Risk of using a function incorrectly. E.g.:
   - Calling a method as a normal function
   - Calling a constructor as a normal function
3. No lexical `this`:
   - normal function inside a method or a constructor: `this` is shadowed

Function expression → arrow function

```
function GuiComponent() {  // constructor
    var that = this;
    var domNode = ...;
    domNode.addEventListener('click', function () {
        console.log('CLICK');
        that.handleClick();  // `this` is shadowed
    });
}
```

```
function GuiComponent() {  // constructor
    var domNode = ...;
    domNode.addEventListener('click', () => {
        console.log('CLICK');
        this.handleClick();  // `this` not shadowed
    });
}
```

Function declaration → const + arrow function

```
function foo(arg1, arg2) {
    ...
}
```

```
const foo = (arg1, arg2) => {
    ...
};
```

- Loss: hoisting, function name
- Many people already avoid function declarations

IIFE → block + let

```
(function () {  // open IIFE
    var tmp = ...;
    ...
}());  // close IIFE
```

```
{  // open block
    let tmp = ...;
    ...
}  // close block
```

Function in object literal → concise method

```
var obj = {
    myMethod: function (arg1, arg2) {
        ...
    }
};
```

```
let obj = {
    myMethod(arg1, arg2) {
        ...
    }
};
```

Constructor → class

```
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); }
```

```
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();
    } }
```

Generators

Generator function:

```
function *foo(arg1, arg2) {
    ...
}
```

Generator method:

```
let obj = {
    *bar() { ... }
};
```

Unfortunate mix of new and old:

- Function declaration syntax
- Method definition syntax

Possible solution: generator arrow functions

Hypothetical syntax:

```
const generatorFunction = (arg1, arg2) =>* {
    ...
};
```

Alas, rejected for ECMAScript 6.

Thin arrow is not needed

Adding methods to an object

ECMAScript 5:

```
MyClass.prototype.foo = function (arg1, arg2) {
    ...
};
```

CoffeeScript

```
MyClass.prototype.foo = (arg1, arg2) -> {
    ...
```

ECMAScript 6:

```
Object.assign(MyClass.prototype, {
    foo(arg1, arg2) {
        ...
    }
});
```

You can often avoid `this`

```
$('ul.tabs li').on('click',
    function () {
        var tab = $(this);
        highlightTab(tab);
        ...
    });
```

```
$('ul.tabs li').on('click',
    event => {
        var tab = $(event.target);
        highlightTab(tab);
        ...
    });
```

API design must change

```
beforeEach(function () {
    this.addMatchers({
        toBeInRange: function (start, end) {
            ...
        }
    });
});
```

```
beforeEach(api => {
    api.addMatchers({
        toBeInRange(start, end) {
            ...
        }
    });
});
```

Have the problems been fixed?

1. Clear separation of concerns w.r.t. functions in ECMAScript 6
2. Incorrect uses:
   - Prevented: can't call a class as a function
   - No help with: calling extracted methods
3. Shadowing this in normal function – prevented via arrow functions.

Iterators

Iterables and iterators

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

Examples of iterables:

- Arrays
- Sets
- Results produced by tool functions
  Examples (built-in, for objects): `keys()`, `values()`, `entries()`
- All array-like DOM objects (eventually)

Iterables and iterators

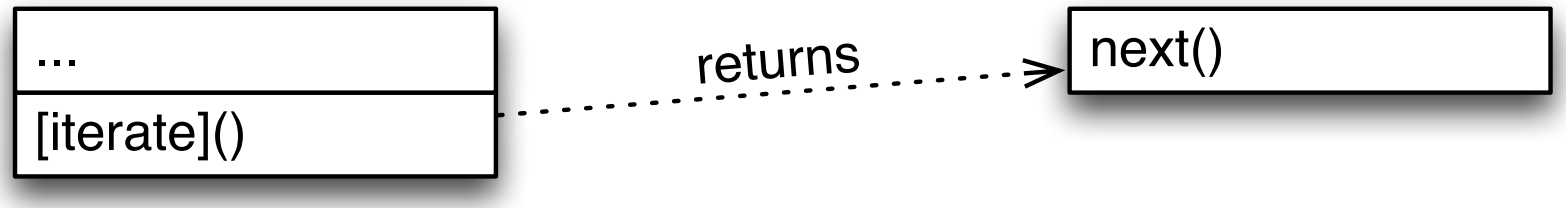
```
import {iterate} from '@iter';  // symbol

let iterable = ['a', 'b'];
let iterator = iterable[iterate]();

iterator.next(); // { value: 'a' }
iterator.next(); // { value: 'b' }
iterator.next(); // { done: true }
```

Iterable:
traversable data structure

Iterator:
pointer for traversing iterable



Example: iterator (1/2)

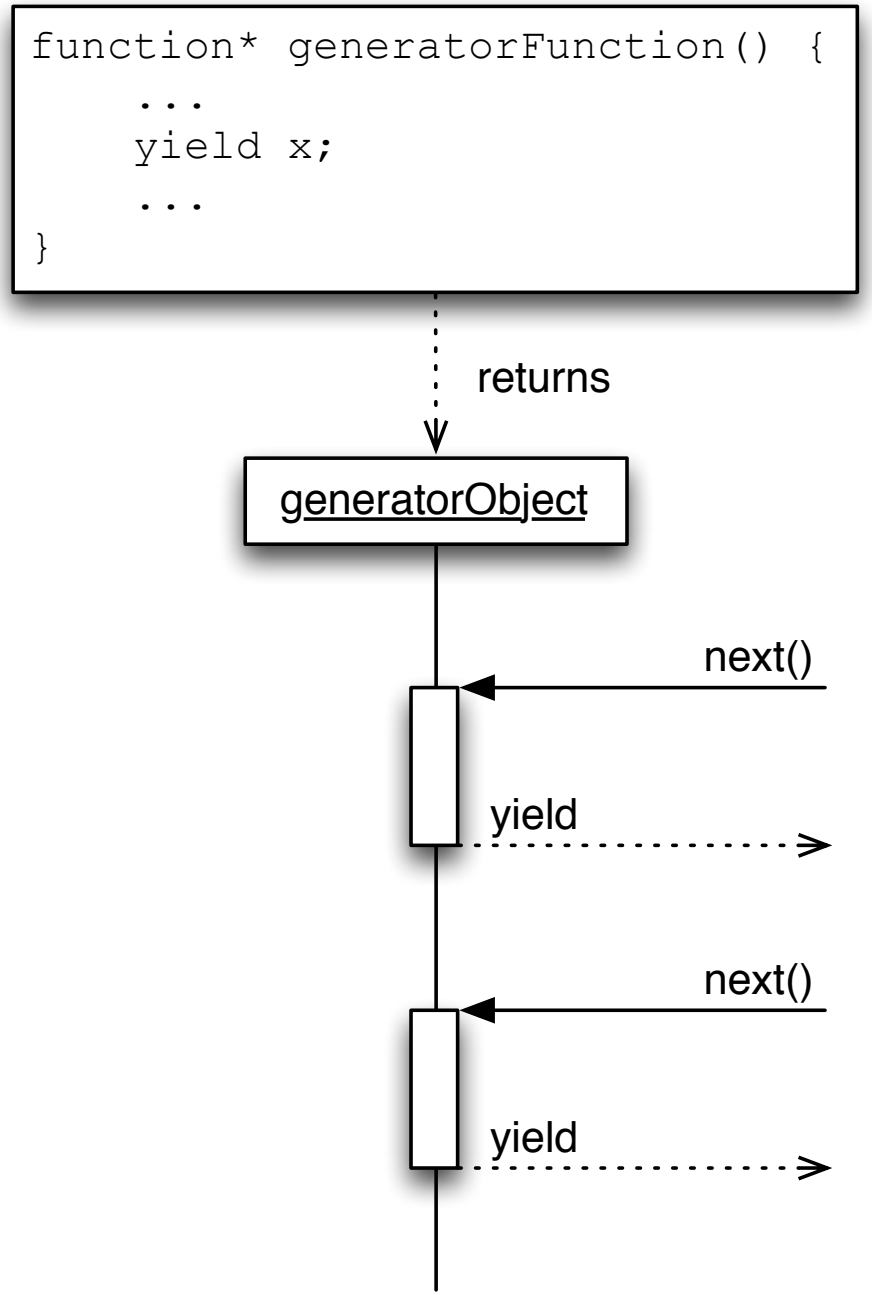
```
import {iterate} from '@iter';  // symbol
function iterArray(arr) {
    ...
    return {
        [iterate]() {  // is iterable
            return this;  // returns iterator
        },
        next() {  // is iterator
            ...  // code is on next slide
        }
    };
}

for (let elem of iterArray(['a', 'b'])) {
    console.log(elem);
}
```

Example: iterator (2/2)

```
function iterArray(arr) {
    var i = 0;
    return {
        ...
        next() {
            if (i < arr.length) {
                return { value: arr[i++] };
            } else {
                return { done: true };
            }
        }
    };
}
```

Generators



Generators: suspend and resume a function

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

Example: a simple generator

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());  // { value: 0, done: false }
console.log(genObj.next());  // { value: 1, done: false }
console.log(genObj.next());  // ( value: 2, done: false }
console.log(genObj.next());  // ( value: undefined, done: true }
```

Example: passing values via next()

```
function *logYields() {
    while (true) {
        let x = yield;
        console.log(x);
    }
}
```

```
# let genObj = logYields();
# genObj.next()
{value: undefined, done: false}
# genObj.next('hello')
hello
{value: undefined, done: false}
# genObj.next('world')
world
{value: undefined, done: false}
```

Generators: implementing an iterator

```
function *iterArray(arr) {
    for (let i=0; i < arr.length; i++) {
        yield arr[i];
    }
}
```

Generators: 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.

```
for (let x of iterTree(['a', [['b', 'c'], 'd'], 'e'])) {
    console.log(x);
}
```

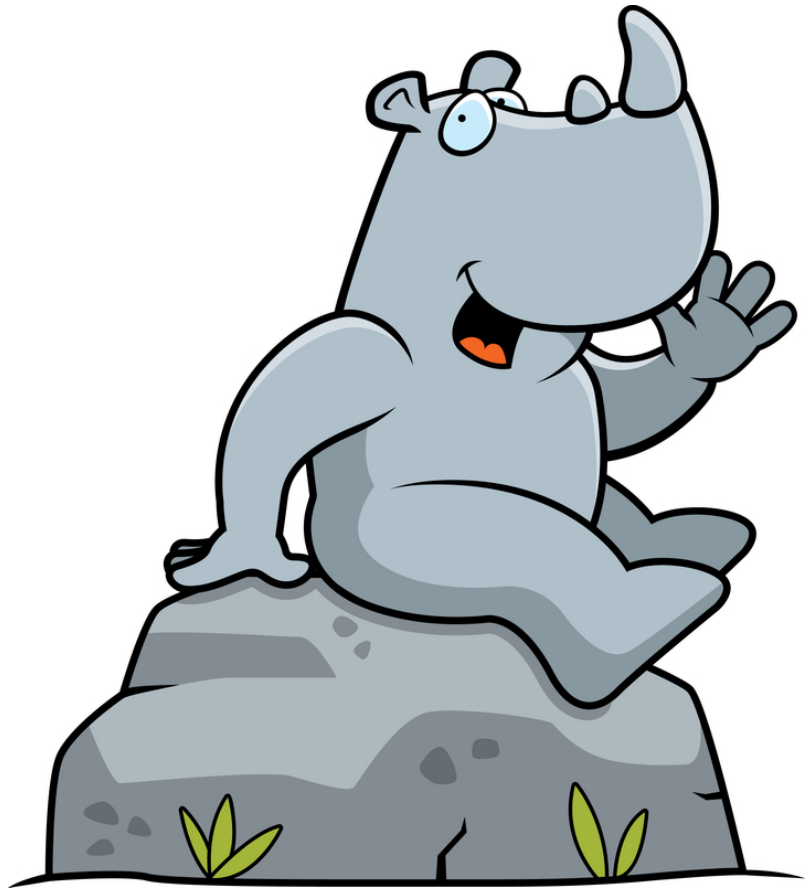
Generators: asynchronous programming

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).

Thank you!



Blog posts on ECMAScript 6:
2ality.com/search/label/esnext