Control Flow Goodness
in ECMAScript 2015 and beyond

Tom Van Cutsem
jsconf.be 2016
Talk Outline

- Part I: the ECMAScript standards process & ES7 sneak peek
- Part II: ECMAScript 6 implementation progress
- Part III: Control flow goodness in ES6
- Part IV: Control flow goodness beyond ES6
- Wrap-up
Part I
The ECMAScript standards process &
ES7 sneak peek
TC39: the JavaScript “standardisation committee”

• Representatives from major Internet companies, browser vendors, web organisations, popular JS libraries and academia. Meets bi-monthly.

• Maintains the ECMA-262 specification.

• The spec is a handbook mainly intended for language implementors.

Allen Wirfs-Brock,
ECMA-262 technical editor (5th & 6th ed.)
A brief history of the ECMAScript spec

1st ed. ‘97
2nd ed. ‘98
3rd ed. ‘99
4th ed. 2008
5th ed. 2009
6th ed. 2015 (a.k.a. ES 2015)
A brief history of the ECMAScript spec

1st ed. '97
2nd ed. '98
3rd ed. '99
4th ed. 2008
5th ed. 2009
6th ed. 2015 (a.k.a. ES 2015)

10 years
6 years

10 years
6 years

2008 2009
2015
Beyond ECMAScript 6: timeline

- 5th ed. (2009)
- June 2015
- ES2015
- June 2016
- ES2016
- 6th ed. (2016)

[Standard ECMAScript 2015 Language Specification]
ES2016 Completed Features

• Few fancy new features included in the spec in the last year

• Exponentiation operator:  \( x^{**} y \)

• Array.prototype.includes:  \([1,2,3].includes(1) === true\)

• Bug fixes and minor details
ES2017 and beyond: proposals on the table

- Draft and Candidate proposals:

| Proposal                                                      | Champion                                           | Stage |
|                                                              |                                                   |       |
| Object.values/Object.entries                                  | Jordan Harband                                     | 4     |
| SIMD.JS - SIMD APIs + polyfill                                | John McCutchan, Peter Jensen, Dan Gohman, Daniel Ehrenberg | 3     |
| Async Functions                                               | Brian Terlison                                     | 3     |
| String padding                                                | Jordan Harband & Rick Waldron                      | 3     |
| Trailing commas in function parameter lists and calls         | Jeff Morrison                                      | 3     |
| Object.getOwnPropertyDescriptor                                 | Jordan Harband & Andrea Giammarchi                 | 3     |
| Function.prototype.toString_revision                          | Michael Ficarra                                    | 3     |
| Asynchronous Iterators                                        | Kevin Smith                                        | 2     |
| function.sent metaproperty                                   | Allen Wirfs-Brock                                  | 2     |
| Rest/Spread Properties                                        | Sebastian Markbage                                 | 2     |
| Shared memory and atomics                                     | Lars T Hansen                                      | 2     |
| System.global                                                 | Jordan Harband                                     | 2     |

See https://github.com/tc39/ecma262 for full list
Part II
ECMAScript 6: implementation progress
ECMAScript 6: timeline

5th ed.  2009

ECMAScript 6 support (April 2016)

(Source: Juriy Zaytsev (kangax)
ECMAScript 6 support (april 2016)

Desktop browsers support nearly all of ES6…
ECMAScript 6 support (April 2016)

... except for Safari (v9 at 53%)
ECMAScript 6 support (April 2016)

Mobile browsers are lagging behind…
**ECMAScript 6 compilers**

- Compile ECMAScript 6 to ECMAScript 5
  
- **Babel**: focus on producing readable (as-if hand-written) ES5 code. Supports JSX as well.
  
- Microsoft **TypeScript**: technically not ES6 but roughly a superset of ES6. Bonus: type inference and optional static typing.
ECMAScript 6: server, desktop and mobile

• V8 5.0 (March 2016) implements over 93% of ES6

• Server: node.js v6.0.0 released 26th of April uses V8 5.0 and so is ES6-ready!

• Desktop: NW.js and Electron use node v5.x

• Mobile: React-native has Babel integration by default [on top of Safari’s JSC]
ECMAScript 6: server, desktop and mobile

http://node.green
Part III
Control flow goodness in ECMAScript 6
New control flow features

- Iterators
- Generators
- Promises
interface Tree<T> {
    key : T,
    left? : Tree<T>,
    right? : Tree<T>
}

let tree: Tree<string> = {
    key: "a",
    left: {
        key: "b",
        left: { key: "c" },
        right: { key: "d" }
    },
    right: {
        key: "e",
        left: { key: "f" },
        right: { key: "g" }
    }
};
Computer Science 101: pre-order tree walk

- Visit node, then left subtree, then right subtree

```javascript
let tree: Tree<string> = {
  key: "a",
  left: {
    key: "b",
    left: { key: "c" },
    right: { key: "d" }
  },
  right: {
    key: "e",
    left: { key: "f" },
    right: { key: "g" }
  }
};
```

assert.deepEqual(preOrder(tree), ["a", "b", "c", "d", "e", "f", "g"])
Computer Science 101: pre-order tree walk

- Visit node, then left subtree, then right subtree

```javascript
function preOrder(tree, accum = []) {
  if (tree) {
    accum.push(tree.key);
    preOrder(tree.left, accum);
    preOrder(tree.right, accum);
  }
  return accum;
}
```

```javascript
assert.deepEqual(preOrder(tree), ["a", "b", "c", "d", "e", "f", "g"])```

![Tree diagram with pre-order traversal](image)
Iterators

• How to support incremental iteration? Change the algorithm so that it returns an iterator.

```plaintext
function preOrderIter(tree: Tree<T>): Iterator<T>

interface Iterator<T> {
    next(): IteratorResult<T>;
}

interface IteratorResult<T> {
    value: T;
    done: bool;
}
```

tree
Using Iterators in ECMAScript 5

- Iteration protocol is explicit in the code

```javascript
function preOrderIter(tree: Tree<T>): Iterator<T> {
  let iter = preOrderIter(tree);
  let nxt = iter.next();
  while (!nxt.done) {
    let k = nxt.value;
    if (k == "d")
      break;
    console.log(k);
    nxt = iter.next();
  }
}
```
Using Iterators in ECMAScript 6

• New for-of loop enumerates all the elements of an iterator or iterable collection

```javascript
function preOrderIter(tree: Tree<T>): Iterator<T>;

for (let k of preOrderIter(tree)) {
  if (k === "d")
    break;
  console.log(k);
}
```
Using Iterators in ECMAScript 6

- The iteration protocol is entirely implicit

```javascript
function preOrderIter(tree: Tree<T>): Iterator<T>;

ES5
let iter = preOrderIter(tree);
let nxt = iter.next();
while (!nxt.done) {
    let k = nxt.value;
    if (k == "d")
        break;
    console.log(k);
    nxt = iter.next();
}

ES6
for (let k of preOrderIter(tree)) {
    if (k == "d")
        break;
    console.log(k);
}
Defining Iterators in ECMAScript 5

- We still need to implement our incremental pre-order tree walk algorithm

```javascript
function preOrderIter(tree: Tree<T>): Iterator<T>;

interface Iterator<T> {
  next(): IteratorResult<T>;
}

interface IteratorResult<T> {
  value: T;
  done: boolean;
}
```
Defining Iterators in ECMAScript 5

- Iteration protocol is explicit. Execution state (call stack) is explicit. Can’t use recursion anymore.

```javascript
function preOrderIter(tree) {
  let todo = [];
  if (tree) {
    todo.push(tree);
  }
  return {
    next() {
      if (todo.length === 0) {
        return {done: true};
      } else {
        let top = todo.pop();
        if (top.right) {
          todo.push(top.right);
        }
        if (top.left) {
          todo.push(top.left);
        }
        return {done: false, value: top.key};
      }
    }
  };
}
```
Defining Iterators in ECMAScript 5

• Can we have our cake and eat it too?

Elegant but batch

```javascript
function preOrder(tree, accum = []) {
  if (tree) {
    accum.push(tree.key);
    preOrder(tree.left, accum);
    preOrder(tree.right, accum);
  }
  return accum;
}
```

Hairy but incremental

```javascript
function preOrderIter(tree) {
  let todo = [];
  if (tree) {
    todo.push(tree);
  }
  return {
    next() {
      if (todo.length === 0) {
        return {done: true};
      } else {
        let top = todo.pop();
        if (top.right) {
          todo.push(top.right);
        }
        if (top.left) {
          todo.push(top.left);
        }
        return {done: false, value: top.key};
      }
    }
  }
}
```
Generators to the rescue!

• A generator function implicitly creates and returns an iterator

```javascript
function preOrderIter(tree: Tree<T>): Iterator<T> {
    if (tree) {
        yield tree.key;
        yield* preOrderIter(tree.left);
        yield* preOrderIter(tree.right);
    }
}
```
Generators in ECMAScript 6

- Both iteration protocol and execution state become implicit

**ES5**

Hairy but incremental

```javascript
function preOrderIter(tree) {
    let todo = [];
    if (tree) {
        todo.push(tree);
    }
    return {
        next() {
            if (todo.length === 0) {
                return {done: true};
            } else {
                let top = todo.pop();
                if (top.right) {
                    todo.push(top.right);
                }
                if (top.left) {
                    todo.push(top.left);
                }
                return {done: false, value: top.key};
            }
        }
    };
}
```

**ES6**

Elegant *and* incremental

```javascript
function* preOrderIter(tree) {
    if (tree) {
        yield tree.key;
        yield* preOrderIter(tree.left);
        yield* preOrderIter(tree.right);
    }
}
```
New control flow features

- Iterators
- Generators
- Promises
ECMAScript 6 Promises

- A promise is a placeholder for a value that may only be available in the future

```javascript
readFile("hello.txt", function (err, content) {
    if (err) {
        // handle error
    } else {
        // use content
    }
});
```

ES5

```javascript
var pContent = readFile("hello.txt");
pContent.then(function (content) {
    // use content
}, function (err) {
    // handle error
});
```

ES6
ECMAScript 6 Promises

- A promise is a placeholder for a value that may only be available in the future

```javascript
var pContent = readFile("hello.txt");
var p2 = pContent.then(function (content) {
  // use content
}, function (err) {
  // handle error
});
```

ES5

```javascript
readFile("hello.txt", function (err, content) {
  if (err) {
    // handle error
  } else {
    // use content
  }
});
```

ES6
ECMAScript 6 Promises

- Promises can be *chained* to avoid callback hell

```javascript
function step1(value, callback): void;

step1(function (e, value1) {
  if (e) {
    return handleError(e);
  }
  step2(value1, function (e, value2) {
    if (e) {
      return handleError(e);
    }
    step3(value2, function (e, value3) {
      if (e) {
        return handleError(e);
      }
      step4(value3, function (e, value4) {
        if (e) {
          return handleError(e);
        }
        // do something with value4
      });
    });
  });
});
```

```javascript
function step1(value): Promise;

step1(value)
  .then(step2)
  .then(step3)
  .then(step4)
  .then(function (value4) {
    // do something with value4
  })
  .catch(function (error) {
    // handle any error here
  });
```

Example adapted from [https://github.com/kriskowal/q](https://github.com/kriskowal/q)
ECMAScript 6 Promises

- Promises already exist as a library in ES5 (e.g. Q, Bluebird)

- Then why standardize?
  - Wide disagreement on a single Promise API. ES6 settled on an API called “Promises/A+”. See promisesaplus.com
  - Standard API allows platform APIs to use Promises as well
  - W3C’s latest DOM APIs already use promises
Part IV
Control flow goodness *beyond* ECMAScript 6
Upcoming control flow features

- Likely in ES2017:
  - Async functions
  - Async iterators
Upcoming control flow features

- Likely in ES2017:
  - Async functions
  - Async iterators
async functions in ECMAScript 2017

- A C# 5.0 feature that enables asynchronous programming using “direct style” control flow (i.e. no callbacks)

ES6

```
function step1(value): Promise;
step1(value).
  .then(step2)
  .then(step3)
  .then(step4)
  .then(function (value4) {
    // do something with value4
  })
  .catch(function (error) {
    // handle any error here
  });
```

ES2017

```
function step1(value): Promise;

(async function() {
  try {
    var value1 = await step1();
    var value2 = await step2(value1);
    var value3 = await step3(value2);
    var value4 = await step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
})();
```
async functions in ECMAScript 6

- Generators can be used as async functions, with some tinkering
- `co` npm library for node (^4.x or ^0.11.x with --harmony flag)

ES2017

```javascript
(async function() {
  try {
    var value1 = await step1();
    var value2 = await step2(value1);
    var value3 = await step3(value2);
    var value4 = await step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
})();
```

ES6

```javascript
co(function*() {
  try {
    var value1 = yield step1();
    var value2 = yield step2(value1);
    var value3 = yield step3(value2);
    var value4 = yield step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
});
```
async functions in ECMAScript 6

- Generators can be used as async functions, with some tinkering

- `co` npm library for node (^4.x or ^0.11.x with --harmony flag)

```javascript
ES2017
(async function() {
  try {
    var value1 = await step1();
    var value2 = await step2(value1);
    var value3 = await step3(value2);
    var value4 = await step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
}())

ES6
co(function*() {
  try {
    var value1 = yield step1();
    var value2 = yield step2(value1);
    var value3 = yield step3(value2);
    var value4 = yield step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
})
```
async functions in ECMAScript 5 (!)

- Babel plug-in based on Facebook Regenerator
  facebook.github.io/regenerator
- Also in TypeScript 1.7+
  github.com/lukehoban/ecmascript-asyncawait

ES2017

```javascript
(async function() {
  try {
    var value1 = await step1();
    var value2 = await step2(value1);
    var value3 = await step3(value2);
    var value4 = await step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
}())
```

ES5

```javascript
(function callee$0$0() {
  var value1, value2, value3, value4;
  return regeneratorRuntime.async(function callee$0$0$(context$1$0) {
    while (1) switch (context$1$0.prev = context$1$0.next) {
      case 0:
        context$1$0.prev = 0;
        context$1$0.next = 3;
        return regeneratorRuntime.awrap(step1());
      case 3:
        value1 = context$1$0.sent;
        context$1$0.next = 6;
        return regeneratorRuntime.awrap(step2(value1));
      case 6:
        value2 = context$1$0.sent;
        context$1$0.next = 9;
        return regeneratorRuntime.awrap(step3(value2));
      case 9:
        value3 = context$1$0.sent;
        context$1$0.next = 12;
        return regeneratorRuntime.awrap(step4(value3));
      case 12:
        value4 = context$1$0.sent;
        context$1$0.next = 17;
        break;
      case 15:
        context$1$0.prev = 15;
        context$1$0.pl0 = context$1$0["catch"](0);
        break;
      case 17:
        case "end":
          return context$1$0.stop();
    }, null, this, [[0, 15]]);
  })()
```

5 (!)

Babel plug-in based on Facebook Regenerator
facebook.github.io/regenerator
Also in TypeScript 1.7+
github.com/lukehoban/ecmascript-asyncawait
Async functions (and generators) can be compiled into state machines. Each intermediate state represents a “yield point”.

ES2017

```javascript
(async function() {
  try {
    var value1 = await step1();
    var value2 = await step2(value1);
    var value3 = await step3(value2);
    var value4 = await step4(value3);
    // do something with value4
  } catch (error) {
    // handle any error here
  }
})();
```

ES5

```javascript
(function callee$0$0() {
  var value1, value2, value3, value4;
  return regeneratorRuntime.async(function callee$0$0$(context$1$0) {
    while (1) switch (context$1$0.prev = context$1$0.next) {
      case 0:
        context$1$0.prev = 0;
        context$1$0.next = 3;
        return regeneratorRuntime.awrap(step1());
      case 3:
        value1 = context$1$0.sent;
        context$1$0.next = 6;
        return regeneratorRuntime.awrap(step2(value1));
      case 6:
        value2 = context$1$0.sent;
        context$1$0.next = 9;
        return regeneratorRuntime.awrap(step3(value2));
      case 9:
        value3 = context$1$0.sent;
        context$1$0.next = 12;
        return regeneratorRuntime.awrap(step4(value3));
      case 12:
        value4 = context$1$0.sent;
        context$1$0.next = 17;
        break;
      case 15:
        context$1$0.prev = 15;
        context$1$0.t0 = context$1$0["catch"](error);
        break;
      case 17:
        context$1$0.prev = 17;
        return context$1$0.stop();
    }
    return context$1$0.stop();
  }, null, this, [[0, 15]]);
})();
```
Upcoming control flow features

• Likely in ES2017:
  • Async functions
  • **Async iterators**
Async Iterators

- ES6 iterator and generator protocols are synchronous...

- …but many Iterable sources are asynchronous in JS

```typescript
interface Iterator<T> {  
    next(): IteratorResult<T>;  
}  

interface IteratorResult<T> {  
    value: T;  
    done: boolean;  
}  

interface AsyncIterator<T> {  
    next(): Promise<IteratorResult<T>>;  
}
```

https://github.com/tc39/proposal-async-iteration
Async Iterators

- Async for-of loop can be used in an async function to consume an async iterator

function readLines(path: string): AsyncIterator<string>;

async function printLines() {
    for await (let line of readLines(filePath)) {
        print(line);
    }
}
Async Generators

Async generators can await, and yield promises

```javascript
function readLines(path: string): AsyncIterator<string> {

async function* readLines(path) {

    let file = await fileOpen(path);

    try {
        while (!file.EOF) {
            yield file.readLine();
        }
    } finally {
        await file.close();
    }
}
```

Polyfill implemented in FB Regenerator
Wrap-up
Take-home messages

ES is alive and kicking. The new yearly release process works!  
Server and desktop browser engines have mostly transitioned, mobile is lagging behind.
Take-home messages

Elegant *and* incremental iteration using iterators and generators

Promises help avoid callback hell

Async functions are a game-changer

(Photo credit: https://www.twilio.com/blog/2015/10/asyncawait-the-hero-javascript-deserved.html)
Thanks for listening!

Control Flow Goodness
in ECMAScript 2015 and beyond

Tom Van Cutsem
jsconf.be 2016