ES6

The *Awesome* Parts

Domenic Denicola

- http://domenic.me/ (blog)
- https://github.com/domenic
- https://npmjs.org/~domenic
- http://slideshare.net/domenicdenicola

Things I'm doing:

- @esdiscuss on Twitter
- The Promises/A+ spec
- The Extensible Web Manifesto

LAB49

Why ES6?

"Stagnation on the web is a social ill."

—Brendan Eich

http://news.ycombinator.com/item?id=4634735

Why JavaScript?

"*It is the* one *language that* every *vendor is committed to shipping compatibly*. Evolving JS has the leverage to add/change the semantics of the platform in a way that no other strategy credibly can."

—Alex Russell

http://infrequently.org/2013/07/why-javascript/

Why Should You Care?

- It's not often that, as a programmer, you get entirely new tools and capabilities in your toolbox.
- Learning Haskell or a Lisp will do it, but then what do you build?
- ES6 provides a unique opportunity.

The Awesome Parts

- Generators
- Template strings
- Proxies (no time today ☹)

Generators

```
function zeroOneTwo() {
  return [0, 1, 2];
}

var array = zeroOneTwo();

for (var i of array) {
  console.log(i);
}
```

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

var generator = zeroOneTwo();

for (var i of generator) {
  console.log(i);
}
```

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

var generator = zeroOneTwo();

for (var i of generator) {
  console.log(i);
}
```

AWESOME?

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

var generator = zeroOneTwo();

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

```
function* fibonacci() {
  var previous = 0, current = 1;
  while (true) {
    var temp = previous;
    previous = current;
    current = temp + current;
    yield current;
  }
}

for (var i of fibonacci()) {
  console.log(i);
}

// 1, 2, 3, 5, 8, 13, ..., Infinity!
```

```
function* take(iterable, numberToTake) {
  var i = 0;
  for (var taken of iterable) {
    if (i++ === numberToTake) {
      return;
    }
    yield taken;
  }
}

for (var i of take(fibonacci(), 5)) {
  console.log(i);
}

// 1, 2, 3, 5, 8 (and done!)
```

Lazy sequences

- You can write lazy versions of everything: filter, map, reduce, all those Underscore.js methods, …

```
var awesomeEls = filter(domEls, isAwesome);
var awesomeTexts = map(awesomeEls, el => el.textContent);
var awesomeString = reduce(awesomeTexts, (acc, t) => acc + t);
```

- Since `filter` and `map` return generators, nothing happens until `reduce`, a non-generator function, executes: then it's all done in a single pass.
- You get the benefits of declarative functional data manipulation without the performance and memory downsides of intermediate arrays.

Lazy sequences

- You can write lazy versions of everything: filter, map, reduce, all those Underscore.js methods, …

```
var awesomeEls = filter(domEls, isAwesome);
var awesomeTexts = map(awesomeEls, el => el.textContent);
var awesomeString = reduce(awesomeTexts, (acc, t) => acc +
```

- Since `filter` and `map` return generators, nothing happ e, a non-generator function, executes: then it's all don ss.
- You get the benefits of declarative functiona ation without the performance and memory downsides of i rrays.

AWESOME!

But wait

- Suspending execution until someone calls `next()` is a powerful feature.
- What if you suspended execution until an async operation finished?

Lots of people are starting to explore this in Node.js right now, leveraging e.g. promises to turn `yield` into a kind of "await."

```
function* loadUI() {
  showLoadingScreen();
  yield loadUIDataAsynchronously();
  hideLoadingScreen();
}

spawn(loadUI);
```

This function returns a promise

Write the function `spawn` so that:

- It calls `next()` and thus gets the loading promise.
- It waits for the promise to fulfill before calling `next()` again.

So we've suspended execution until the async operation finishes!

```
function* loadAndShowData() {
  var data = yield loadData();
  showData(data);
}

spawn(loadAndShowData);
```

This function returns a promise for data

You can actually pass data back in to the generator, causing `yield` to either return a value or throw an exception inside the generator body.

- So if the promise fulfills, send back its fulfillment value, so that the `data` variable ends up holding it.
- But if the promise rejects, tell the generator to throw the rejection reason as an exception.

```
function* loadAndShowData() {
  showLoadingIndicator();
  try {
    var data = yield loadData();
    showData(data);
  } catch (e) {
    showError(e);
  } finally {
    removeLoadingIndicator();
  }
}

spawn(loadAndShowData);
```

```
function* loadAndShowData() {
  showLoadingIndicator();
  try {
    var data = yield loadData();
    showData(data);
  } catch (e) {
    showError(e);
  } finally {
    removeLoadingIndicator();
  }
}

spawn(loadAndShowData);
```

AWESOME!

Where Can I Use This?

- Traceur: yes
- Continuum: yes
- Browsers: Chrome, Firefox*
- Node.js: 0.11.3+

es6ify

http://thlorenz.github.io/es6ify/

Developer Tools - http://thlorenz.github.io/es6ify/

Elements Resources Network Sources Timeline Profiles Audits Console

Sources Content scri... Snippets

- thlorenz.github.io
  - es6ify
    - javascripts
      - example/src
        - features
          - arrow-functions.js
          - block-scope.js
          - default-parameters.js
          - destructuring.js
          - generators.js
          - index.js
          - iterators.js
          - rest-parameters.js
          - spread-operator.js
        - main.js
        - make-monster.js
      - node_modules/node-traceur
      - respond.js
    - stylesheets
    - (index)
- ajax.googleapis.com

generators.js

```
1  // A binary tree class.
2  function Tree(left, label, right) {
3    this.left = left;
4    this.label = label;
5    this.right = right;
6  }
7
8  // A recursive generator that iterates the Tree labels in-order.
9  function* inorder(t) {
10   if (t) {
11     yield* inorder(t.left);
12     yield t.label;
13     yield* inorder(t.right);
14   }
15 }
16
17 // Make a tree
18 function make(array) {
19   // Leaf node:
20   if (array.length == 1) return new Tree(null, array[0], null);
21   return new Tree(make(array[0]), array[1], make(array[2]));
22 }
23
24 module.exports = function () {
25   let tree = make([[['a'], 'b', ['c']], 'd', [['e'], 'f', ['g']]]);
26   console.log('generating tree labels in order:');
27
28   // Iterate over it
29   for (let node of inorder(tree)) {
30     console.log(node); // a, b, c, d, ...
31   }
32 };
33
```

Paused

Watch Expressions

No Watch Expressions

Call Stack

$__generatorWrap	generators.js:70
inorder	generators.js:1
$G.innerFunction	generators.js:13
$G.moveNext	generators.js:2
traceur.runtime.addIterator.send	generators.js:17
$G.innerFunction	generators.js:20
$G.moveNext	generators.js:2
traceur.runtime.addIterator.send	generators.js:17
$G.innerFunction	generators.js:20
$G.moveNext	generators.js:2
traceur.runtime.addIterator.send	generators.js:17
traceur.runtime.addIterator.next	generators.js:31
module.exports	generators.js:5
./make-monster	main.js:17
i	[VM] bundle.js (97):1
$create	[VM] bundle.js (97):1
(anonymous	[VM] bundle.js (97):1

Template Strings

```
var someArithmetic = `${x} + ${y} = ${x + y}`;

var longString = `long
string
is
long`;
```

```
var someArithmetic = `${x} + ${y} = ${x + y}`;

var longString = `long
string
is
long`;
```

USEFUL

```
var someArithmetic = `${x} + ${y} = ${x + y}`;

var longString = `long
string
is
long`;
```

NOT AWESOME

```
var whatsThis = func`${x} + ${y}\n= ${x + y}`;

// becomes

var whatsThis = func(
  {
    raw:    ['', ' + ', '\\n = ', ''],
    cooked: ['', ' + ', '\n = ', '']
  },
  x,
  y,
  x + y
);
```

```
var whatsThis = func`${x} + ${y}\n= ${x + y}`;

// becomes

var whatsThis = func(
  {
    raw:    ['', ' + ', '\\n = ', ''],
    cooked: ['', ' + ', '\n = ', '']
  },
  x,
  y,
  x + y
);
```

```
var whatsThis = func`${x} + ${y}\n= ${x + y}`;

// becomes

var whatsThis = func(
  {
    raw:    ['', ' + ', '\\n = ', ''],
    cooked: ['', ' + ', '\n = ', '']
  },
  x,
  y,
  x + y
);
```

```
var whatsThis = func`${x} + ${y}\n= ${x + y}`;

// becomes

var whatsThis = func(
  {
    raw:    ['', ' + ', '\\n = ', ''],
    cooked: ['', ' + ', '\n = ', '']
  },
  x,
  y,
  x + y
);
```

AWESOME?

Contextual Auto-Escaping

```
var els = document.querySelectorAll('.' + className);
// what if className contains "."?


var els = qsa`.${className}`;
// => qsa({ raw: ['.', ''], … }, className);
// if we write qsa well, it can detect the preceding "."
// and escape className!

function qsa({ raw, cooked }, ...vars) {
  // `raw[0]` ends in '.'
  // so `vars[0]` needs to be escaped like a CSS class
  // similar rules for IDs, attribute selectors, etc.
}
```

Contextual Auto-Escaping In Overdrive

validate (no "s or such)
filter (e.g. no javascript: URLs)

percent-encode

```
safehtml`<a href="${url}?q=${query}"
          onclick="alert('${message}')"
          style="color: ${color}">
        ${message}
        </a>`
```

escape JS (e.g. closing quote)
HTML encode

HTML encode

escape CSS (e.g. ; or :)
censor unsafe CSS (e.g. expression())
HTML encode

Contextual Auto-Escaping In Overdrive

```
var url = 'http://example.com/', query = 'Hello & Goodbye';
var message = 'Goodbye & Hello', color = 'expression(alert(1337))';

safehtml`<a href="${url}?q=${query}"
            onclick="alert('${message}')"
            style="color: ${color}">
           ${message}
         </a>`
```

```
<a href="http://example.com/?q=Hello%20%26%20Goodbye"
   onclick="alert(&#39;Goodbye&#32;\x26&#32;Hello&#39;)"
   style="color: CENSORED">
  Goodbye &amp; Hello
</a>
```

http://js-quasis-libraries-and-repl.googlecode.com/svn/trunk/safetemplate.html

Localization and Formatting

```
l10n`Hello ${name}; you are visitor number ${visitor}:n!
     You have ${money}:c in your account!`

// Write a l10n function that:
// - if it sees nothing after the var, it does nothing
// - if it sees :n, it uses localized number formatter
// - if it sees :c, it uses localized currency formatter
```

Dynamic Regular Expressions

```
/\d+,\d+/

// But now ',' needs to be configurable, so you do

new RegExp('\\d+' + separator + '\\d+')

// Ick! Instead, write a re function to make this work

re`\d+${separator}\d+`
```

Embedded HTML/XML

```
jsx`<a href="${url}">${text}</a>`

// write a smart jsx function that transforms this into

React.DOM.a({ href: url }, text)

// zomg, no compile-to-JS language required!
```

DSLs for Code Execution

```
var childProcess = sh`ps ax | grep ${pid}`;

var xhr = POST`http://example.org/service?a=${a}&b=${b}

                Content-Type: application/json
                Authorization: ${credentials}

                { "foo": ${foo}, "bar": ${bar} }`;
```

We Just Solved:

- String interpolation/basic templating
- Multiline strings
- Contextual auto-escaping (giving *generic* injection protection)
- Localization and formatting
- Dynamic regular expressions
- Embedded HTML

… and opened up a whole new world of DSLs for writing intuitive, readable JS

We Just Solved:

- String interpolation/basic templating
- Multiline strings
- Contextual auto-escaping (giving *generic* injection protection)
- Localization and formatting
- Dynamic regular expressions
- Embedded HTML

… and opened up a whole new world of DS[illegible]ituitive, readable JS

AWESOME!

Where Can I Use This?

- Traceur: yes
- Continuum: yes
- Browsers: no
- Node.js: no

The Future Now

- The future of JS: it's important!
- Learn more:
  - es6isnigh.com
  - harmony:proposals wiki page
- Follow @esdiscuss / read esdiscuss.org.
- Be adventurous; use a transpiler!
- *Don't stagnate.*