Using ES6 Today!

R. Mark Volkmann
Object Computing Inc.
mark@ociweb.com

based on an article at http://sett.ociweb.com/sett/settApr2014.html

ECMAScript

- Defined by **European Computer Manufacturers Association** (ECMA)
- Specification is called **ECMAScript** or ECMA-262
  - JavaScript 5.1 (**ES5**) - http://www.ecma-international.org/publications/standards/Ecma-262.htm
  - JavaScript 6 (**ES6**) - http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts
- **ECMAScript Technical Committee** is called **TC39**
- Meetings
  - "TC39 has bi-monthly face to face meetings, usually in California (Bay area).
    In addition, at least one meeting is held in Redmond, WA (July meeting usually)."
- Besides defining the standard,
  - "TC39 members create and test implementations of the candidate specification
    to verify its correctness and the feasibility of creating interoperable implementations."
- **Current members** include
  - **Brendan Eich** (Mozilla, JavaScript inventor), **Douglas Crockford** (PayPal), Brandon Benvie, Dave Herman (Mozilla), Luke Hoban, Yehuda Katz (Tilde Inc., Ember.js), Mark Miller (Google), Alex Russell (Dojo Toolkit), Rick Waldron (Boucoup, jQuery), **Allen Wirfs-Brock** (Mozilla), and many more

ES6 Features

- See **Luke Hoban's** (TC39 member) **summary** at https://github.com/lukehoban/es6features

ES6 includes the following new features:

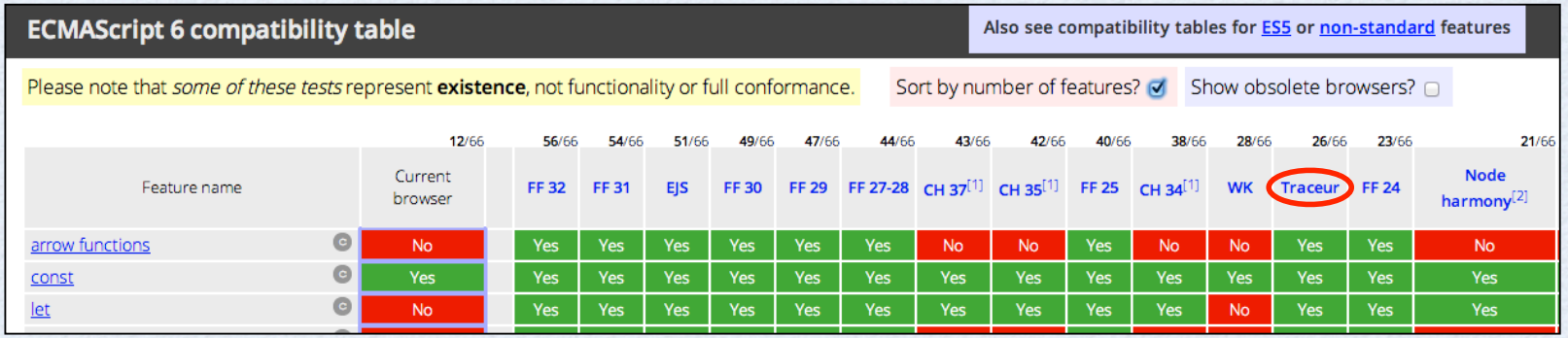
- arrows
- classes
- enhanced object literals
- template strings
- destructuring
- default + rest + spread
- let + const
- iterators + for..of
- generators
- comprehensions
- unicode
- modules
- module loaders
- map + set + weakmap + weakset
- proxies
- symbols
- subclassable built-ins
- promises
- math + number + string + object APIs
- binary and octal literals
- reflect api
- tail calls

Transpilers

- Compilers translate code one language to another
  - ex. Java to bytecode
- Transpilers translate code to the same language
- There are transpilers that translate ES6 code to ES5
- Examples
  - **Traceur** from Google
    - discussed more later
  - **ES6 Module Transpiler** from Square
    - converts module syntax to AMD, CommonJS, or globals
    - doesn't support other ES6 features
    - http://square.github.io/es6-module-transpiler/
  - **esnext** from Square
    - as of 7/4/14 considered early alpha
    - supports fewer ES6 features than Traceur
    - https://github.com/square/esnext

Use ES6 Today?

- It **may take years** for all the features in ES6 to be supported in all major browsers
- That's **too long to wait** and you **don't have to wait**
- **Use a transpiler** to get comfortable with new features sooner and allow writing more compact, more expressive code now
- For a **summary of ES6 feature support in browsers**, and in the Traceur tool discussed next, see ES6 compatibility table from Juriy Zaytsev (a.k.a. kangax)
  - http://kangax.github.io/es5-compat-table/es6/
  - try selecting "Sort by number of features?" checkbox



ECMAScript 6 compatibility table

Also see compatibility tables for ES5 or non-standard features

Please note that *some of these tests* represent **existence**, not functionality or full conformance.

Sort by number of features? ☑ Show obsolete browsers? ☐

Feature name	Current browser	FF 32	FF 31	EJS	FF 30	FF 29	FF 27-28	CH 37[1]	CH 35[1]	FF 25	CH 34[1]	WK	Traceur	FF 24	Node harmony[2]
	12/66	56/66	54/66	51/66	49/66	47/66	44/66	43/66	42/66	40/66	38/66	28/66	26/66	23/66	21/66
arrow functions	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	No	No	Yes	Yes	No
const	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
let	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes

Google Traceur

- **Most capable** ES6 to ES5 transpiler in terms of feature support
- **Implemented in ES6** and uses itself to transpile to ES5 code that runs on Node.js
- https://github.com/google/traceur-compiler
- **Online tool** at http://google.github.io/traceur-compiler/demo/repl.html
  - enter ES6 on left side and see resulting ES5 code on right
  - useful for testing support for specific ES6 features and gaining an understanding of what Traceur generates
  - does not execute code
  - "Options" menu includes ability to enable experimental features
- **To install**
  - install Node.js
  - run `npm install -g traceur`

Running Traceur

- To get help on options
  - **`traceur --help`**
  - **`traceur --longhelp`**
- To run code in an ES6 file
  - **`traceur`** *`es6-file-path`*
  - requires file extension to be **`.js`**, but it can be omitted in the command
- To compile an ES6 file to an ES5 file
  - **`traceur --script`** *`es6-file-path`* **`--out`** *`es5-file-path`*
  - generated code depends on provided file **`traceur-runtime.js`**
    - can be copied from directory where Traceur is installed
  - to use generated code in a browser, include a script tag for **`traceur-runtime.js`**
- Experimental features
  - to use, add **`--experimental`** option
  - examples of features currently considered experimental include **`let`** keyword, symbols, and async functions

doesn't check for native browser support; does some feature detecting like not adding shim methods if already present

Sourcemaps

- Allow browser debuggers to step through code that was transpiled from another language into JavaScript
  - for example, debug CoffeeScript code
  - can debug ES6 code that was transpiled to ES5
- Traceur option `--sourcemap` causes it to generate a sourcemap
  - places them in same directory as generated ES5 files
  - browser looks for them there

Using Sourcemaps

- In Chrome
  - open a page that uses transpiled ES6 code
  - open Developer Tools
  - click gear icon in upper-right
  - check "Search in content scripts"
  - check "Enable JavaScript source maps"
  - select ES6 .js files from "Sources" tab
  - set breakpoints
  - refresh page
- In Firefox
  - enabled by default
  - to open Firefox debugger,
    select Tools ... Web Developer ... Debugger

Linting

- It is important to use some linting tool when writing JavaScript
- Saves time and reduces errors by catching coding issues before code is run
- Can be run from command-line,
  integrated into editors/IDEs, and
  run automatically when files are saved from any editor using tools like Grunt/Gulp
- Most popular JavaScript linting tools
  - JSLint - http://jslint.org; unclear if or when JSLint will support ES6
  - JSHint - http://jshint.org; has good support now using "esnext" option
  - ESLint - http://eslint.org; plans to support ES6, but doesn't yet
- I highly recommend using JSHint to check ES6 code

Automation

- **Grunt** - http://gruntjs.com
  - great tool for automating web development tasks
  - over 3,100 plugins available
  - several related to Traceur including "traceur", "traceur-latest", "traceur-build", "**traceur-simple**", and "node-traceur"
  - see example `Gruntfile.js` in article
    - uses "**watch**" plugin to watch for changes to HTML, CSS and JavaScript files
    - when watch detects these, it automatically runs specified tasks including linting CSS and JavaScript, running Traceur to generate ES5 code, and refreshing browser to immediately show results of changes
    - last part is enabled by "**livereload**" option and including a special script tag in main HTML file
- **Gulp** - http://gulpjs.com
  - similar to Grunt
  - also supports watch and livereload
  - emphasizes use of file streaming for better efficiency

ES6 Features

- We will focus primarily on the subset currently supported by Traceur

Block Scope ...

- These require **`--experimental`** option in Traceur
- **`const`** declares constants with block scope
  - must be initialized
  - can't be modified (but Traceur doesn't currently enforce, issue #18)
- **`let`** declares variables like **`var`**, but they have block scope

  support for **`let`** in Traceur 0.0.25 seems sketchy
  - not hoisted to beginning of enclosing block, so references before declaration are errors
  - all uses of **`var`** can be replaced with **`let`**
  - when a file defines a module, top-level uses of **`let`** are file scoped, unlike **`var`**
  - Traceur implements block scopes in ES5 with a catch block (see in Web REPL)

    see https://github.com/getify/You-Dont-Know-JS/blob/master/scope%20&%20closures/apB.md
  - when a **`let`** variable is accessed out of its scope, Traceur throws a **`ReferenceError`** with message "***`name`*** **`is not defined`**"

... Block Scope

- block functions
  - functions declared in a block are scoped to that block

```
function outer() {
  console.log('in outer');
}

{
  function inner() {
    console.log('in inner');
  }

  outer(); // works
  inner(); // works
}

outer(); // works
inner(); // throws ReferenceError
```

Symbols

- Immutable identifiers that are guaranteed to be unique
  - unlike strings
- Can use as object keys
  - but they become non-enumerable properties
  - even `Object.getOwnPropertyNames(obj)` cannot see them!
- To create a symbol
  - `let sym = Symbol(description);`
  - description is optional and only useful for debugging
  - retrieve description with `sym.name` (not supported in Traceur)
  - note `new` keyword is not used
- To use a symbol
  - `obj[sym] = value;`

Modules ...

- A module is defined by a JavaScript file
  that exports values and functions
  to be shared with other files that import it
- The contents of a module are not wrapped
  in any special construct
- Top-level variables and functions that are not exported
  are not visible in other source files
- Module code is evaluated in strict mode by default
  - no need to specify `'use strict';`
- Syntax is still being debated
  - concept of defining a default export and importing only it
    may be dropped, so not presented here

... Modules ...

- To **export** a value from a module
  - `export var name = value;`
- To **export** a function from a module
  - `export function name(params) { ... }`

can export any number of values and functions from a module

- To **export** multiple, previously defined values and functions in a single line
  - `export {name1, name2, ...}`
- To **import** all exports from a module into a single object
  - `module obj from 'module-path';`
  - *obj* is read-only
  - JSHint doesn't recognize this syntax
- To **import** specific exports from a module
  - `import {name1, name2, ...} from 'module-path';`

module paths **do not** include the `.js` file extension; can start with `./` or `../`

... Modules ...

- To transpile ES6 files that use modules
  - transpile just main file to generate a single ES5 file that contains all required code
  - `traceur --script main6.js --out main.js --sourcemap`
- Traceur generated sourcemaps support modules
  - can step through each of the original ES6 files
    that make up a single generated ES5 file
- Use in browsers requires `traceur-runtime.js`
  - if Traceur was installed using `npm install -g traceur`,
    determine where global modules are installed with `npm -g root`
    and copy `traceur-runtime.js` from `traceur/bin` below that directory
  - add `script` tag for this in main HTML file

… Modules

bar6.js

```
export var bar1 = 'the value of bar1';

export function bar2() {
  console.log('in bar2');
}
```

foo6.js

```
import {bar1, bar2} from './bar6';

export var foo1 = 'the value of foo1';
console.log('foo6: bar1 =', bar1);

export function foo2() {
  console.log('in foo2');
  bar2();
}
```

main6.js

```
import {foo1, foo2} from './foo6';
console.log('in main');
console.log('foo1 =', foo1);
foo2();
```

index.html

```
<html>
  <head>
    <title></title>
    <script src="lib/traceur-runtime.js"></script>
    <script src="gen/main.js"></script>
  </head>
  <body>
    See console output.
  </body>
</html>
```

To run from command-line:

```
traceur main6
```

To generate ES5 and sourcemap:

```
traceur --script main6.js \
--out gen/main.js --sourcemap
```

Output:

```
foo6: bar1 = the value of bar1
in main
foo1 = the value of foo1
in foo2
in bar2
```

Arrow Functions

- **`(params) => { expressions }`**
  - can omit parens if only one parameter, but need if no parameters
  - if only one expression, can omit braces and its value is returned without using **`return`** keyword

    `expression` can even be another arrow function that is returned
  - cannot insert line feed between parameters and **`=>`**
  - if expression is an object literal, wrap it in parens
  - **`this`** has same value as containing scope, not a new value (called "lexical this")
    - so can't use to define constructors or methods, just plain functions
- Examples

```
var arr = [1, 2, 3, 4];
var doubled = arr.map(x => x * 2);
console.log(doubled); // [2, 4, 6, 8]

var product = (a, b) => a * b;
console.log(product(2, 3)); // 6

var average = numbers => {
  var sum = numbers.reduce((a, b) => a + b);
  return sum / numbers.length;
};
console.log(average(arr)); // 2.5
```

Classes ...

- Use `class` keyword
- Define constructor and instance methods inside
  - can only have one constructor function per class

```
class Shoe {
  constructor(brand, model, size) {
    this.brand = brand;
    this.model = model;
    this.size = size;
    Shoe.count += 1;
  }
  equals(obj) {
    return obj instanceof Shoe &&
      this.brand === obj.brand &&
      this.model === obj.model &&
      this.size === obj.size;
  }
  toString() {
    return this.brand + ' ' + this.model +
      ' in size ' + this.size;
  }
}

Shoe.count = 0;
Shoe.createdAny = () => Shoe.count > 0;

var s1 = new Shoe('Mizuno', 'Precision 10', 13);
var s2 = new Shoe('Nike', 'Free 5', 12);
var s3 = new Shoe('Mizuno', 'Precision 10', 13);
console.log('created any?', Shoe.createdAny()); // true
console.log('count =', Shoe.count); // 3
console.log('s2 = ' + s2); // Nike Free 5 in size 12
console.log('s1.equals(s2) =', s1.equals(s2)); // false
console.log('s3.equals(s3) =', s3.equals(s3)); // true
```

not a standard JS method

class property

class method

… Classes

- Inherit with **`extends`** keyword

```
class RunningShoe extends Shoe {
  constructor(brand, model, size, type) {
    super(brand, model, size);
    this.type = type;
    this.miles = 0;
  }
  addMiles(miles) { this.miles += miles; }
  shouldReplace() { return this.miles >= 500; }
}

var rs = new RunningShoe(
  'Nike', 'Free Everyday', 13, 'lightweight trainer');
rs.addMiles(400);
console.log('should replace?', rs.shouldReplace()); // false
rs.addMiles(200);
console.log('should replace?', rs.shouldReplace()); // true
```

`super` calls corresponding method in superclass; using it inside `constructor` is an example of this

Enhanced Object Literals ...

- Literal objects can omit value for a key if it's in a variable with the same name
- Example

```
var fruit = 'apple', number = 19;
var obj = {fruit, foo: 'bar', number};
console.log(obj);
// {fruit: 'apple', foo: 'bar', number: 19}
```

JSHint doesn't recognize this syntax yet

... Enhanced Object Literals

- Properties with function values can be specified more easily

```
var obj = {
  oldStyle: function (params) { ... },
  newStyle(params) { ... }
};
```

- Computed properties names can be specified inline

```
// Old style
var obj = {};
obj[expression] = value;

// New style
var obj = {
  [expression]: value;
};
```

Property Method Assignment

- Alternative way to attach a function to a literal object
- Example

```
var obj = {
  number: 2,
  multiply: function (n) { // old way
    return this.number * n;
  },
  times(n) { // new way
    return this.number * n;
  },
  // This doesn't work because the
  // arrow function "this" value is not obj.
  product: n => this.number * n
};

console.log(obj.multiply(2)); // 4
console.log(obj.times(3)); // 6
console.log(obj.product(4)); // NaN
```

New `Math` Methods

- `Math.fround(number)` - returns nearest single precision floating point number to *`number`*
- `Math.sign(number)` - returns sign of *`number`*; -1, 0 or 1
- `Math.trunc(number)` - returns integer part of *`number`*
- `Math.cbrt(number)` - returns cube root of *`number`*
- `Math.expm1(number)` - returns `exp(number) - 1`;
  `Math.exp` returns e (Euler's constant) raised to *`number`* power
- `Math.hypot(x, y, ...)` - returns square root of sum of squares of arguments
- `Math.imul(n1, n2)` - multiplies two 32-bit integers; for performance
- logarithmic functions - `Math.log1p(number)`, `Math.log10(number)`, `Math.log2(number)`
- hyperbolic trig functions - `Math.asinh(number)`, `Math.acosh(number)`, `Math.atanh(number)`

New `Number` Methods

- **`Number.isFinite(n)`** - returns boolean indicating whether ***`n`*** is a **`Number`** and is not **`NaN`**, **`Infinity`** or **`-Infinity`**
- **`Number.isInteger(n)`** - returns boolean indicating whether ***`n`*** is an integer and not **`NaN`**, **`Infinity`** or **`-Infinity`**
- **`Number.isNaN(n)`** - returns boolean indicating whether ***`n`*** is the special **`NaN`** value
- **`Number.toInteger(n)`** - converts a number to an integer
- **`Number.parseInt(string)`** - parses a string into an integer; same as the global function
- **`Number.parseFloat(string)`** - parses a string into a double; same as the global function

New `String` Methods

- ***s1*.startsWith(*s2*)** - determines if starts with given characters
- ***s1*.endsWith(*s2*)** - determines if ends with given characters
- ***s1*.contains(*s2*)** - determines if contains given characters
- ***s*.repeat(*count*)** - creates new string by copying ***s count*** times

can specify starting position of test for each of these

- JavaScript uses UTF-16 characters
  - each occupies two or four bytes
  - `length` property of JavaScript strings, as well as `charAt` and `charCodeAt` methods assume two bytes per character
  - no easy way to get or create 4-byte characters in ES5
  - ***string*.codePointAt(*pos*)**
    gets UTF-16 integer value at a given position
  - **String.fromCodePoint(*int1*, ..., *intN*)**
    returns string created from any number of UTF-16 integer values

use of 4-byte UTF-16 characters is somewhat rare (ex. Egyptian Hieroglyphs), so this is often not a problem

New `Array` Methods

not supported by Traceur yet

- `Array.of(values)` - creates an `Array` from it's arguments
  - can use literal array syntax instead
- `Array.from(arrayLikeObj, mapFn)` - creates an `Array` from an `Array`-like object
  - *`mapFn`* is an optional function that is called on each element to transform the value

- *`arr`*`.find(`*`predicateFn`*`)` - returns first element in *`arr`* that satisfies a given predicate function
  - *`predicateFn`* is passed element, index, and *`arr`*
  - if none satisfy, `undefined` is returned
- *`arr`*`.findIndex(`*`predicateFn`*`)` - same as `find`, but returns index instead of element
  - if none satisfy, -1 is returned
- *`arr`*`.fill(`*`value, startIndex, endIndex`*`)` - fills *`arr`* with a given value
  - *`startIndex`* defaults to 0; *`endIndex`* defaults to the array length

New `Object` Methods ...

- `Object.assign(target, src1, ... srcN)`
  - copies properties from src objects to target, replacing those already present
  - can use to create a shallow clone an object
  - useful in constructors →

```
class Shoe {
  constructor(brand, model, size) {
    this.brand = brand;
    this.model = model;
    this.size = size;
    // or
    Object.assign(this,
      {brand, model, size});
  }
  ...
}
```

uses enhanced object literal

- `Object.is(value1, value2)`
  - determines if value1 and value2 are the same
    - values can be primitives or objects; objects are the same only if they are the same object
    - unlike `===`, this treats `Number.NaN` as the same as `Number.NaN`
    - google "MDN JavaScript Object" for more detail
- `Object.setPrototypeOf(obj, prototype)`
  - changes prototype of an existing object
  - use is discouraged because it is slow and makes subsequent operations on the object slow

... New `Object` Methods

- **`Object.values(obj)`**
  - returns iterator over values; similar to ES5 `Object.keys(obj)`
  - in ES7
- **`Object.entries(obj)`**
  - returns iterator over `[key, value]` pairs
  - in ES7

```
for (let [k, v] of Object.entries(myObj)) {
  // use k an v
}
```

Default Parameters

- Parameters with default values must follow those without
- Example

```
var today = new Date();

function makeDate(day, month = today.getMonth(), year = today.getFullYear()) {
  return new Date(year, month, day).toDateString();
}

console.log(makeDate(16, 3, 1961)); // Sun Apr 16 1961
console.log(makeDate(16, 3)); // Wed Apr 16 2014
console.log(makeDate(16)); // Sun Feb 16 2014
```

run on 2/28/14

- Idiom for required parameters (from Allen Wirfs-Brock)

```
function req() { throw new Error('missing argument'); }
function foo(p1 = req(), p2 = req(), p3 = undefined) {
  ...
}
```

Rest Parameters

- Gather variable number of arguments after named parameters into an array
- If no corresponding arguments are supplied, value is an empty array, not undefined

```
function report(firstName, lastName, ...colors) {
  var phrase = colors.length === 0 ? 'no colors' :
    colors.length === 1 ? 'the color ' + colors[0]:
    'the colors ' + colors.join(' and ');
  console.log(firstName, lastName, 'likes', phrase + '.');
}

report('Mark', 'Volkmann', 'yellow');
// Mark Volkmann likes the color yellow.
report('Tami', 'Volkmann', 'pink', 'blue');
// Tami Volkmann likes the colors pink and blue.
report('John', 'Doe');
// John Doe likes no colors.
```

Spread Operator

- Spreads out elements of an array
  so they are treated as separate arguments to a function
- Examples

```
var arr1 = [1, 2];
var arr2 = [3, 4];
arr1.push(...arr2);
console.log(arr1); // [1, 2, 3, 4]

var dateParts = [1961, 3, 16];
var birthday = new Date(...dateParts);
console.log(birthday.toDateString());
// Sun Apr 16, 1961
```

alternative to

```
arr1.push.apply(arr1, arr2);
```

Destructuring ...

- Assigns values to multiple variables and parameters from values in objects and arrays
- Can be used to swap variable values
- LHS expression can be nested to any depth

```
var a = 1, b = 2, c = 3;
var [a, b, c] = [b, c, a];
console.log('a =', a); // 2
console.log('b =', b); // 3
console.log('c =', c); // 1

function report([name, color]) {
  console.log(name + "'s favorite color is", color + '.');
}
var data = ['Mark', 'yellow'];
report(data); // Mark's favorite color is yellow.

var arr = [1, [2, 3], [[4, 5], [6, 7, 8]]];
var [a, [, b], [[c], [,, d]]] = arr;
console.log('a =', a); // 1
console.log('b =', b); // 3
console.log('c =', c); // 4
console.log('d =', d); // 8

var obj = {color: 'blue', weight: 1, size: 32};
var {color, size} = obj;
console.log('color =', color); // blue
console.log('size =', size); // 32

function report2(p1, {weight, color}) {
  console.log(p1, color, weight);
}
report2(19, obj); // 19 blue 1
```

extracting array elements by position

extracting object property values by name

… Destructuring

- Great for getting parenthesized groups of a `RegExp` match

```
let dateStr = 'I was born on 4/16/1961 in St. Louis.';
let re = /(\\d{1,2})\\/(\\d{1,2})\\/(\\d{4})/;
let [, month, day, year] = re.exec(dateStr);
console.log('date pieces =', month, day, year);
```

- Great for configuration kinds of parameters or any time named parameters are desired (common when there are many)

```
function config({color, size, speed, volume}) {
  console.log('color =', color); // yellow
  console.log('size =', size); // 33
  console.log('speed =', speed); // fast
  console.log('volume =', volume); // 11
}

config({
  size: 33,
  volume: 11,
  speed: 'fast',
  color: 'yellow'
});
```

order is irrelevant

Collections ...

- New collection classes include
  - `Set`
  - `Map`
  - `WeakSet`
  - `WeakMap`
- Not supported by Traceur, but can use a shim
  - one option is es6-shim at https://github.com/paulmillr/es6-shim/ (supports IE9+)
- Find a good shim and try it!

`Set` Class

- Values can be any type
- To create, `var set = new Set()`
  - can pass iterable object to constructor to add all its elements
- To add an element, `set.add(value);`
- To delete an element, `set.delete(value)`
- To delete all elements, `set.clear()`
- To test for element, `set.has(value)`
- `keys` method is an alias to `entries` method
- `values` method returns iterator over elements
- `entries` method returns iterator over elements
- `forEach` method is like in Array,
  but passes *`value`*, *`value`* and *`set`* to callback

for API consistency

these iterate in insertion order

Map Class

- Differs from JavaScript objects in that keys are not restricted to strings
- To create, `var map = new Map()`
  - can pass iterable object to constructor to add all its pairs (array of `[key, value]`)
- To add or modify a pair, `map.set(key, value)`
- To get a value, `map.get(key);`
- To delete a pair, `map.delete(key)`
- To delete all pairs, `map.clear()`
- To test for key, `map.has(key)`
- `size` property holds number of keys
- `keys` method returns iterator over keys
- `values` method returns iterator over values
- `entries` method returns iterator over array of `[key, value]` arrays
- `forEach` method is like in Array, but passes `value`, `key` and `map` to callback

these iterate in insertion order

`WeakSet` Collection

- Similar API to `Set`, but has no `size` property or iteration methods
- Differs in that
  - values are "weakly held",
    i.e. can be garbage collected if not referenced elsewhere
  - can't iterate over values

`WeakMap` Collection

- Similar API to `Map`, but has no `size` property or iteration methods
- Differs in that
  - keys and values are "weakly held",
    i.e. can be garbage collected if not referenced elsewhere
  - can't iterate over keys or values

Proxies

- Can intercept getting and setting properties in an object to provide extra or alternate functionality
- Can intercept calls to a specific function and provide alternate behavior
- Uses new `Proxy` class
- Can intercept these operations
  - `get`, `set`, `has`, `deleteProperty`
  - `construct`, `apply`
  - `getOwnPropertyDescriptor`, `defineProperty`
  - `getPrototypeOf`, `setPrototypeOf`
  - `enumerate`, `ownKeys`
  - `isExtensible`, `preventExtensions`
- Not supported yet by Traceur

Promises ...

- Proxy for a value that may be known in the future
  after an asynchronous operation completes
- Can register functions to be invoked when a promise is
  resolved (with a value) or
  rejected (with a reason)
- Create with `Promise` constructor, passing it
  a function that takes `resolve` and `reject` functions
- Register to be notified when promise is resolved or rejected
  with `then` (resolve or reject) or `catch` (only reject) method
- See example on next slide

… Promises

in real usage, some asynchronous operation would happen here

```
function asyncDouble(n) {
  return new Promise((resolve, reject) => {
    if (typeof n === 'number') {
      resolve(n * 2);
    } else {
      reject(n + ' is not a number');
    }
  });
}

asyncDouble(3).then(
  data => console.log('data =', data), // 6
  err => console.error('error:', err));
```

ISSUE:
Errors in resolve and reject callbacks of `then` method are silently ignored!
Remember to wrap code with `try/catch`.
This is not the case in more advanced promise implementations like Bluebird.

- Static methods
  - `Promise.resolve(value)` returns promise that is resolved with given value
  - `Promise.reject(reason)` returns promise that is rejected with given reason
  - `Promise.all(iterable)` returns promise that is resolved when all promises in `iterable` are resolved
    - resolves to array of results in order of provided promises
    - if any are rejected, this promise is rejected
  - `Promise.race(iterable)` returns promise that is resolved when any promise in `iterable` is resolved or rejected when any promise in `iterable` is rejected

for-of Loops

- New way of iterating over elements in a sequence where iteration variable is scoped to loop
  - for arrays, this is an alternative to for-in loop and `Array` `forEach` method
- Value after `of` can be an **array or iterator**
  - iterators are described next
- Example

```
var stooges = ['Moe', 'Larry', 'Curly'];
for (let stooge of stooges) {
  console.log(stooge);
}
```

Iterators

- Iterators are objects that can visit elements in a sequence
  - constructor is `Object`, not a custom class
  - have a method whose name is the value of `Symbol.iterator`
  - this method returns an object with a `next` method and an optional `throw` method
    - described on next slide
- Iterators for objects
  - TC39 is considering adding class methods named `keys`, `values`, and `entries` to some class (maybe `Dict` or `Object`) for obtaining iterators over object properties

Iterator Methods

- **`next`** method
  - gets next value in sequence
  - takes optional argument, but not on first call
    - specifies value that the **`yield`** hit in this call will return at the start of processing for the next call
  - returns object with **`value`** and **`done`** properties
  - **`done`** will be true if end of sequence has been reached
  - when **`done`** is true, **`value`** is not valid; typically **`undefined`**
- **`throw`** method
  - optional
  - takes error argument and throws it inside generator function that created the iterator
  - can catch inside generator function

generators will be discussed soon

Iterator Example #1

```
let fibonacci = {
  [Symbol.iterator]() {
    let prev = 0, curr = 1, result = {done: false};
    return {
      next() {
        [prev, curr] = [curr, prev + curr];
        result.value = curr;
        return result;
      }
    }
  }
}

for (let n of fibonacci) {
  if (n > 100) break;
  console.log(n);
}
```

compare to slide 37

```
1
2
3
5
8
13
21
34
55
89
```

Iterator Example #2

```
var arr = [1, 2, 3, 5, 6, 8, 11];
var isOdd = (n) => n % 2 === 1;

// This is less efficient than using an iterator because
// the Array filter method builds a new array and
// iteration cannot begin until that completes.
arr.filter(isOdd).forEach((n) => console.log(n));

// This is more efficient, but requires more code.
function getFilterIterator(arr, filter) {
  var index = 0, iter = {}, result = {done: false};
  iter[Symbol.iterator] = () => {
    return {
      next() {
        while (true) {
          if (index >= arr.length) return {done: true};
          result.value = arr[index++];
          if (filter(result.value)) return result;
        }
      }
    };
  };
  return iter;
}

for (let v of getFilterIterator(arr, isOdd)) {
  console.log(v); // 1 3 5 11
}
```

Generators

- Functions that have multiple return points
  - each is specified using **`yield`** keyword
- Generator functions implicitly return an iterator object
  - each **`yield`** is hit in separate calls to the iterator **`next`** method
- Can obtain values from a sequence one at a time
  - lazy evaluation or infinite sequences
- Defined with **`function*`** ***`name`*****`(`*****`params`*****`) { `*****`code`*****` }`**
  - ***`code`*** uses **`yield`** keyword to return each value in sequence, often inside a loop
  - ends when generator function exits or **`return`** keyword is used (value returned is not yielded)

Steps to Use Generators

- 1) Call generator function to obtain an iterator object
- 2) Call iterator `next` method to request next value
  - optionally pass a value that iterator can use to compute the subsequent value
  - after iterator "yields" next value,
    its code is "suspended" until next request
- 3) Process value
- 4) Repeat from step 2

When an iterator is used in a `for-of` loop
it performs steps 2 and 4.
Step 3 goes in loop body.

```
for (let v of someGenerator()) {
  // use v
}
```

Generator `yield`

- To return a "normal" value
  - `yield value;`
- To return the value returned by another generator
  - `yield* otherGenerator(params);`
  - delegates to other generator

```
function* fib() {
  var [prev, curr] = [0, 1];
  while (true) {
    [prev, curr] = [curr, prev + curr];
    yield curr;
  }
}

for (let value of fib()) {
  if (value > 100) break;
  console.log(value);
}
```

compare to slide 33

```
1
2
3
5
8
13
21
34
55
89
```

More Generator Examples

```
function* gen1() {
  yield 'foo';
  yield 'bar';
  yield 'baz';
}

for (let value of gen1()) {
  console.log(value);
}
```

```
function* gen2(v) {
  try {
    v = yield 'foo' + v;
    v = yield 'bar' + v;
    yield 'baz' + v;
  } catch (e) {
    console.error('caught', e);
  }
}

var iter = gen2(1);
var result = iter.next(); // can't pass data in first call to next
console.log(result.value); // foo1

result = iter.next(2);
console.log(result.value); // bar2

//iter.throw('stop now');

result = iter.next(3);
console.log(result.value); // baz3

if (!result.done) {
  console.log(iter.next(4)); // not called
}
```

Generators For Async ...

```
function double(n) {
  return new Promise((resolve) => resolve(n * 2));
}

function triple(n) {
  return new Promise((resolve) => resolve(n * 3));
}

function badOp(n) {
  return new Promise((resolve, reject) => reject('I failed!'));
}

function async(generatorFn) {
  var iter = generatorFn();
  function success(result) {
    var next = iter.next(result);
    // next.value is a promise
    // next.done will be false when iter.next is called after
    // the last yield in workflow (on next slide) has run.
    if (!next.done) next.value.then(success, failure);
  }
  function failure(err) {
    var next = iter.throw(err);
    // next.value is a promise
    // next.done will be false if the error was caught and handled.
    if (!next.done) next.value.then(success, failure);
  }
  success();
}
```

multiplies a given number by 2 asynchronously

multiplies a given number by 3 asynchronously

called on next slide

The magic! This obtains and waits for each of the promises that are yielded by the specified generator function. It is a utility method that would only be written once.

`next.value` will be a promise

BUT DON'T DO THIS! See `async` and `await` keywords ahead.

compare to slide 43

... Generators for Async

Call multiple asynchronous functions in series
in a way that makes them appear to be synchronous.
This avoids writing code in the pyramid of doom style.

```
async(function* () { // passing a generator
  var n = 1;
  try {
    n = yield double(n);
    n = yield triple(n);
    //n = yield badOp(n);
    console.log('n =', n); // 6
  } catch (e) {
    // To see this happen, uncomment yield of badOp.
    console.error('error:', e);
  }
});
```

These yield promises that the **async** function waits on to be resolved or rejected.

This can be simplified with
new language keywords!

What's Next?

- The next version is always referred to as "JS-next"
- Currently that is ES7
- Will include
  - **`async`** and **`await`** keywords
  - type annotations
  - new **`Object`** method **`observe`**
  - collections of weak references
  - value objects - immutable datatypes for representing many kinds of numbers
  - more

`async` and `await` ...

- Keywords to be added in ES7
  - already implemented in Traceur as an experimental feature
  - JSHint doesn't recognize these yet
- Hide use of generators for managing async operations, simplifying code
- Replace use of **`yield`** keyword with **`await`** keyword to wait for a value to be returned asynchronously
  - **`await`** can be called on any function
  - not required to be marked as **`async`** or return a **`Promise`**
- Mark functions that use **`await`** with **`async`** keyword

... async and await

```
function sleep(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}

async function double(n) {
  await sleep(50);
  return n * 2;
}

function triple(n) {
  return new Promise(resolve => resolve(n * 3));
}

function quadruple(n) {
  return n * 4;
}

function badOp() {
  return new Promise(
    (resolve, reject) => reject('I failed!'));
}
```

compare to slide 39

async function

function that returns a promise

a "normal" function

Call multiple asynchronous functions in series in a way that makes them appear to be synchronous. This avoids writing code in the pyramid of doom style.

```
async function work() {
  var n = 1;
  try {
    n = await double(n);
    n = await triple(n);
    //n = await badOp(n);
    n = await quadruple(n);
    console.log('n =', n); // 24
  } catch (e) {
    // To see this happen,
    // uncomment await of badOp.
    console.error('error:', e);
  }
}

work();
```

Types ...

- Optional type annotations for
  variables, properties, function parameters, and function return types
  - current syntax: ***`thing-to-annotate`*`:`*`type-expression`***
  - details of syntax are still being determined
  - if not specified, can hold any kind of value
- Will provide run-time type checking
- Can specify builtin types and names of custom classes
- Types are first-class values
  - can be stored in variables and passed to functions
- Builtin types: **`boolean`**, **`number`**, **`string`**, **`void`**, **`any`**
- Traceur experimental mode supports specifying types,
  but doesn't enforce them yet

... Types

```
function initials(name:string):string {
  return name.split(' ').map(part => part.charAt(0)).join('');
}

function isFullName(name:string):boolean {
  return name.split(' ').length >= 3;
}

var name = 'Richard Mark Volkmann';
//var name = 'Mark Volkmann';
console.log('initials are', initials(name)); // RMV
console.log('full name?', isFullName(name)); // true
```

```
// Polyfill for new ES6 method not supported yet by Traceur.
Math.hypot = (x, y) => Math.sqrt(x * x + y * y);

class Point {
  constructor(x:number, y:number) {
    this.x = x;
    this.y = y;
  }

  distanceFrom(point:Point) {
    return Math.hypot(this.x - point.x, this.y - point.y);
  }
}

var p1 = new Point(1, 2);
var p2 = new Point(4, 6);
console.log('distance =', p1.distanceFrom(p2));
```

Summary

- Which features of ES6 should you start using today?
- I recommend choosing those in the intersection of the set of features supported by Traceur and JSHint
- Includes at least these
  - arrow functions
  - block scope (const, let, and functions)
  - classes
  - default parameters
  - destructuring
  - for-of loops
  - iterators
  - generators
  - modules
  - rest parameters
  - spread operator
  - template strings
  - new methods in String and Object classes