ECMAScript (ES) 2015

a.k.a. ES6

Table of Contents

- Overview - 3
- Transpilers - 6
- Source Maps - 13
- Block Scope - 18
- Default Parameters - 20
- Rest Parameters - 21
- Spread Operator - 22
- Destructuring - 23
- Arrow Functions - 27
- Symbols - 29
- Enhanced Object Literals - 32
- Classes - 34
- Getters and Setters - 37
- New **`Math`** Functions - 39
- New **`Number`** Functions - 40
- Numeric Literals - 41
- New **`String`** Methods - 42
- Template Literals - 43
- New **`Array`** Functions and Methods - 46
- New **`Object`** Functions - 48
- **`Reflect`** Object - 50
- **`for-of`** Loops - 52
- Collections (**`Set`**, **`Map`**, **`WeakSet`**, **`WeakMap`**) - 53
- Promises - 64
- Modules - 69
- jspm - 75
- Iterators and Iterables - 80
- Generators - 87
- Proxies - 97
- Tail Call Optimization - 99
- **`async`** and **`await`** - 101
- Type Annotations - 103

ECMAScript

- Defined by **European Computer Manufacturers Association** (ECMA)
- Specification is called **ECMAScript** or ECMA-262
  - http://www.ecma-international.org/publications/standards/Ecma-262.htm
  - **ES 2015** was approved on June 17, 2015
- **ECMAScript Technical Committee** is called **TC39**
- TC39 has bi-monthly face-to-face meetings
- 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), **Allen Wirfs-Brock** (Mozilla), Dave Herman (Mozilla), Brandon Benvie (Mozilla), Mark Miller (Google), Alex Russell (Google, Dojo Toolkit), Erik Arvidsson (Google, Traceur), Domenic Denicola (Google), Luke Hoban (Microsoft), Yehuda Katz (Tilde Inc., Ember.js), Rick Waldron (Boucoup, jQuery), and many more

ES5 vs. ES 2015

- ECMAScript 5 did not add any new syntax
- ECMAScript 2015 does!
- ES 2015 is backward compatible with ES5, which is backward compatible with ES3
- Many ES 2015 features provide **syntactic sugar** for more concise code

- One goal of ES 2015 and beyond is to make JavaScript a **better target for compiling to from other languages**

- Spec sizes
  - **ES5** - **258 pages**
  - **ES 2015** (6th edition) - **566 pages** (approved on June 17, 2015)

"One JavaScript"

- Approach named by David Herman
- Allows JavaScript to evolve without versioning
  - avoids migration issues like Python 2 to Python 3
- "Don't break the web!"
  - removing features would cause existing web apps to stop working
  - can add new, better features
  - ES5 strict mode was a bit of a mistake since it broke some existing code
    - this is why ES 2015 supports "sloppy mode" code outside modules and class definitions
- Use linting tools to detect use of "deprecated" features
  - ex. switching from **`var`** to **`let`** and **`const`** and using rest parameters in place of **`arguments`** object

Transpilers

- Compilers translate code one language to another
  - ex. Java to bytecode
- Transpilers translate code to the same language
- There are several transpilers that translate ES 2015 code to ES5

ES 2015 Transpilers

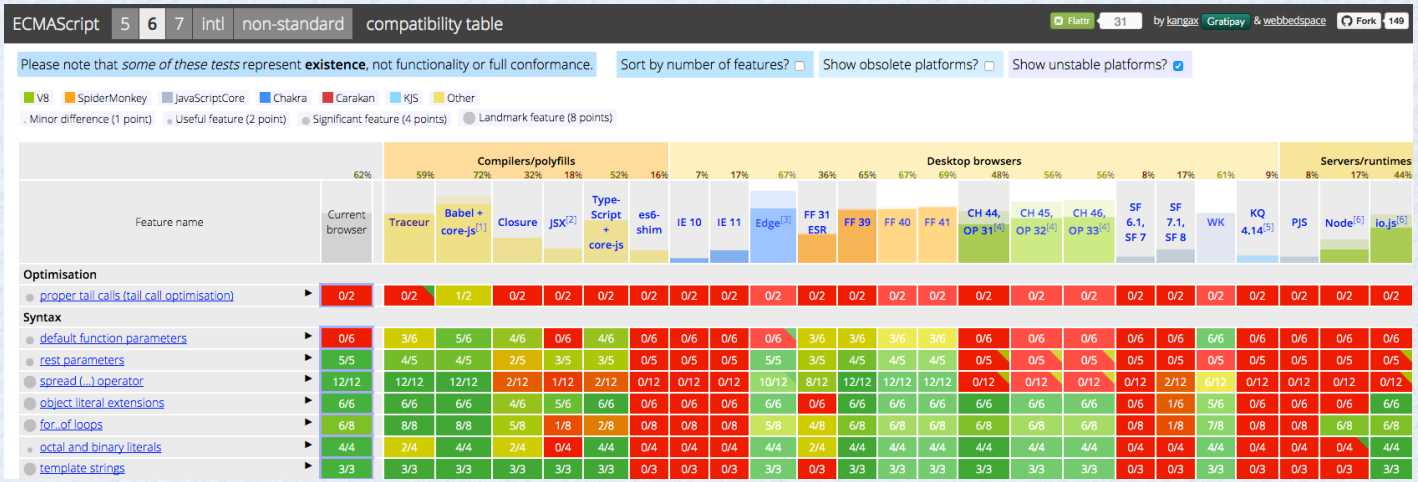
percentages are as of 8/6/15

- **Babel - 72%**
  - aims to generate ES5 code that is as close a possible to the input ES 2015 code
  - generates source maps
  - some features don't work with IE10 and below
    - see https://babeljs.io/docs/usage/caveats/#internet-explorer
  - https://babeljs.io
- **Traceur - 59%**
  - from Google
  - generates source maps
  - doesn't work with IE8 and below
    - due to use of ES5 `get`/`set` syntax
  - https://github.com/google/traceur-compiler/
- **TypeScript - 52%**
  - from Microsoft
  - "a typed superset of JavaScript that compiles to plain JavaScript. Any browser. Any host. Any OS. Open Source."
  - supports optional type specifications for variables, function return values, and function parameters
  - has goal to support all of ES 2015
  - not currently a goal to transpile all ES 2015 features to ES5!
  - generates source maps
  - to install, `npm install -g typescript`
  - to compile, `tsc some-file.ts`
    - generates `some-file.js`
  - http://www.typescriptlang.org

there are more, but these are the most popular and/or support the most features

Use ES 2015 Today?

- It **may take years** for all the features in ES 2015 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 ES 2015 feature support in browsers and transpilers**, see ES6 compatibility table from Juriy Zaytsev (a.k.a. kangax)
  - http://kangax.github.io/compat-table/es6/



Traceur

- **Implemented in ES 2015** 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 ES 2015 on left side and see resulting ES5 code on right
  - useful for testing support for specific ES 2015 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
  - `npm install -g traceur`

AngularJS 2 uses Traceur for ES 2015 support

Running Traceur

- To get help on options
  - **`traceur --help`**
  - **`traceur --longhelp`**
- To run code in an ES 2015 file
  - **`traceur`** *`es2015-file-path`*
  - requires file extension to be **`.js`**, but it can be omitted in the command
- To compile an ES 2015 file to an ES5 file
  - **`traceur --out`** *`es5-file-path es2015-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 symbols, **`async`**/**`await`** keywords, and type annotations

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

Babel

- **Implemented in ES 2015** and uses itself to transpile to ES5 code that runs on Node.js
- http://babeljs.io
- **Online tool** at http://babeljs.io/repl/
  - enter ES 2015 on left side and see resulting ES5 code on right
  - useful for testing support for specific ES 2015 features and gaining an understanding of what Babel generates
  - optionally executes code (when "Evaluate" checkbox is checked)
    - output is at bottom of ES5 code
  - "Experimental" and "Playground" checkboxes enable ES7 features and other "ideas"
- **To install**
  - install Node.js
  - **`npm install -g babel`**

"Babel works perfectly with React, featuring a built-in JSX transformer."

Running Babel

- To get help on options
  - **`babel --help`**
- To run code in an ES 2015 file
  - **`babel-node`** *`es2015-file-path`*
  - file extension can be omitted and defaults to **`.js`**
- To compile an ES 2015 file to an ES5 file
  - **`babel`** *`es2015-file-path`* **`-o`** *`es5-file-path`*

in *nix environments, can use redirection
`babel es2015-file-path > es5-file-path`

- To compile a many ES 2015 files to ES5 files
  - **`babel`** *`es2015-dir`* **`-o`** *`es5-dir`*
- Experimental features
  - to use some ES 2016 features, add **`--experimental`** option
- Optional babel-runtime
  - http://babeljs.io/docs/usage/transformers/#self-contained

Source Maps

- Allow browser debuggers to step through code that was transpiled from another language into JavaScript
  - for example, debug CoffeeScript code
  - can debug ES 2015 code that was transpiled to ES5
- **Traceur**
  - option `--source-maps` causes it to generate a source map
  - places them in same directory as generated ES5 files
  - browser looks for them there
- **Babel**
  - plugins for Grunt and Gulp can generate source maps

Using Source Maps

- In Chrome
  - open a page that uses transpiled ES 2015 code
  - open Developer Tools (cmd-option-i on Mac, ctrl-shift-i on Windows)
  - click gear icon in upper-right to see settings
  - check "Search in content scripts"
  - check "Enable JavaScript source maps"
  - select ES 2015 .js files from "Sources" tab
  - set breakpoints and refresh page
- In Firefox
  - open Firefox debugger by selecting Tools ... Web Developer ... Debugger (cmd-option-s on Mac, ctrl-shift-s on Windows?)
  - click gear icon in upper-right to see "Debugger Options" and verify that "Show Original Sources" is selected
  - select ES 2015 .js files from "Sources" tab
  - set breakpoints and refresh page

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 ES 2016
  - JSHint - http://jshint.org; has good support now using "**esnext**" option
  - ESLint - http://eslint.org; recently added support ES 2015; needs more testing
- I highly recommend using JSHint to check ES 2015 code

Automation

- **Grunt** - http://gruntjs.com
  - great tool for automating web development tasks
  - 4,472 plugins available as of 3/8/15
    - for Traceur support, see these plugins: traceur, traceur-latest, traceur-build, **traceur-simple**, and node-traceur
    - for Babel support, see the plugin grunt-babel
  - 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 in goal to Grunt, but configuration is different
  - 1,457 plugins available as of 3/8/15
  - also supports watch and livereload
  - emphasizes use of file streaming for better efficiency
  - see gulp-traceur and gulp-babel plugins

see Grunt and Gulp examples at https://github.com/mvolkmann/gulp-traceur-demo

ES 2015 Features

- The following slides describe most of the features in ES 2015
- Also see Luke Hoban's (TC39 member) summary
  - https://github.com/lukehoban/es6features

Block Scope ...

- **`const`** declares constants with block scope
  - must be initialized
  - reference can't be modified, but object values can
    - to prevent changes to object values, use `Object.freeze(obj)`
- **`let`** declares variables like **`var`**, but they have block scope
  - not hoisted to beginning of enclosing block, so references before declaration are errors
  - most uses of **`var`** can be replaced with **`let`** (not if they depend on hoisting)
  - when a file defines a module, top-level uses of **`let`** are file-scoped, unlike **`var`**
  - Traceur and Babel implement block scopes by renaming variables declared in block
  - when a **`let`** variable is accessed out of its scope,
    a **`ReferenceError`** is thrown with message "***`name`*** **`is not defined`**"

… Block Scope

- Block functions
  - functions declared in a block are scoped to that block
  - for example, in if and for-loop blocks
- Use a `{}` block
  in place of an IIFE

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

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

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

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

Default Parameters

- Example

```
let 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

- Default value expressions can refer to preceding parameters
- Explicitly passing **`undefined`** triggers use of default value
  - makes it okay for parameters with default values to precede those without
- Idiom for required parameters (from Allen Wirfs-Brock)

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

Rest Operator

- Gathers variable number of arguments after named parameters into an array
- If no corresponding arguments are supplied, value is an empty array, not **`undefined`**
- Removes need to use **`arguments`** object

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

Spread Operator

- Spreads out elements of any "iterable" (discussed later) so they are treated as separate arguments to a function or elements in a literal array
- Mostly removes need to use `Function apply` method

examples of things that are iterable include arrays and strings

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

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

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

```
let arr1 = ['bar', 'baz'];
let arr2 = ['foo', ...arr1, 'qux'];
console.log(arr1); // ['foo', 'bar', 'baz', 'qux']
```

Destructuring ...

- Assigns values to any number of variables from values in iterables and objects

```
// Positional destructuring of iterables
let [var1, var2] = some-iterable;
// Can skip elements (elision)
let [,,var1,,var2] = some-iterable;

// Property destructuring of objects
let {prop1: var1, prop2: var2} = some-obj;
// Can omit variable name if same as property name
let {prop1, prop2} = some-obj;
```

get error if RHS is `null` or `undefined`

- Can be used in variable declarations/assignments, parameter lists, and for-of loops (covered later)
- Can't start statement with `{`, so when assigning to existing variables using object destructuring, surround with parens

```
({prop1: var1, prop2: var2} = some-obj);
```

... Destructuring ...

- LHS expression can be nested to any depth
  - arrays of objects, objects whose property values are arrays, ...
- LHS variables can specify default values

```
[var1 = 19, var2 = 'foo'] = some-iterable;
```

  - default values can refer to preceding variables
- Positional destructuring can use rest operator for last variable

```
[var1, ...others] = some-iterable;
```

- When assigning rather than declaring variables, any valid LHS variable expression can be used
  - ex. `obj.prop` and `arr[index]`
- Can be used to swap variable values `[a, b] = [b, a];`
- Useful with functions that have multiple return values
  - really one array or object

... Destructuring ...


```
let arr = [1, [2, 3], [[4, 5], [6, 7, 8]]];
let [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

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

let team = {
  catcher: {
    name: 'Yadier Molina',
    weight: 230
  },
  pitcher: {
    name: 'Adam Wainwright',
    height: 79
  }
};
let {pitcher: {name}} = team;
console.log('pitcher name =', name); // Adam Wainwright
let {pitcher: {name: pName}, catcher: {name: cName} = team;
console.log(pName, cName); // Adam Wainwright Yadier Molina
```

extracting array elements by position

extracting object property values by name

creates `name` variable, but not `pitcher`

… 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 of any time named parameters are desired (common when many)

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

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

order is irrelevant

Arrow Functions ...

- **(*params*) => { *expressions* }**
  - if only one parameter and not using destructuring, can omit parens
  - if no parameters, need parens
  - cannot insert line feed between parameters and =>
  - if only one expression, can omit braces and its value is returned without using **`return`** keyword
    - ***`expression`*** can be another arrow function that is returned
  - if expression is an object literal, wrap it in parens to distinguish it from a block of code

All functions now have a **`name`** property. When an anonymous function, including arrow functions, is assigned to a variable, that becomes the value of its **`name`** property

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

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

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

Arrow functions are typically used for anonymous functions like those passed to **`map`** and **`reduce`**.

Functions like **`product`** and **`average`** are better defined the normal way so their names appear in stack traces.

… Arrow Functions

- Inside arrow function, **`this`** has same value as containing scope, not a new value (called "lexical this")
  - so can't use to define constructor functions or prototype methods, only plain functions
- Also provides "lexical super" for use in class constructors and methods
  - can use **`super`** keyword to invoke a superclass method
- Immediately invoked functions (IIFEs)
  - not typically needed in ES 2015 since modules provide file scope
  - can write like this

```
(() => {
  ...
})();
```

  - ending like this is a syntax error

```
(() => {
  ...
}()));
```

Some have requested an alternative without lexical this that would be useful for adding a method to an object, maybe `->`. So far that has been rejected because existing anonymous functions can be used for that.

Symbols ...

- Immutable identifiers that are guaranteed to be unique
  - unlike strings
- To create a "local" symbol
  - `let sym = Symbol(description);`
  - note `new` keyword is not used
    - throws `TypeError`; it's a function, not a constructor
  - description is optional and mainly useful for debugging
- To get description
  - `sym.toString()`
  - returns `'Symbol(description)'`
  - concatenating a symbol to a string throws `TypeError`
- A new primitive type
  - `typeof sym === 'symbol'`

Global Symbols

`let gs = Symbol.for(description);`
creates a new global symbol
if none with the description exists;
otherwise returns existing global symbol

To get description, `Symbol.keyFor(gs)`
- returns `undefined` for non-global symbols

... Symbols

- Can use as object keys
  - `obj[sym] = value;`
- They become non-enumerable properties (not private)
  - `Object.getOwnPropertyNames(obj)` gets string keys, but not symbol keys
  - `Object.getOwnPropertySymbols(obj)` gets symbol keys, but not string keys
  - `Reflect.ownKeys(obj)` gets both string and symbol keys
- Can use for constants that only serve as unique identifiers
  - `const NAME = Symbol();`
- Can use to add "meta-level" properties or internal methods to an object that avoid clashing with normal properties
  - `Symbol.iterator` is an example (described later)
- To use in Traceur and Babel, enable experimental mode

Well Known Symbols

- Used as method names in custom classes to override how instances are processed by certain operators and built-in class methods
  - **`Symbol.hasInstance`** method affects **`instanceof`** operator
  - **`Symbol.isConcatSpreadable`** method affects the **`Array`** **`concat`** method
  - **`Symbol.iterator`** method makes objects "iterable" and affects for-of loops
  - **`Symbol.match`** method affects the **`String`** **`match`** method
  - **`Symbol.replace`** method affects the **`String`** **`replace`** method
  - **`Symbol.search`** method affects the **`String`** **`search`** method
  - **`Symbol.split`** method affects the **`String`** **`split`** method
  - **`Symbol.species`** method allows methods of a class to create new objects using the same or another (ex. superclass) constructor
  - **`Symbol.toPrimitive`** method affects conversion to a primitive
  - **`Symbol.toStringTag`** method affects conversion to a string
  - **`Symbol.unscopables`** method affects **`with`** statements (which you shouldn't use); for legacy code

unlike constants whose names are all uppercase, these have camelcase names

Enhanced Object Literals ...

- Literal objects can omit value for a key if it's in a variable with the same name
  - similar to destructuring syntax

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

- Computed property names can be specified inline

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

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

one use is to define properties and methods whose keys are symbols instead of strings

... Enhanced Object Literals

- Property method assignment
  - alternative way to attach a method to a literal object

```
let 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
```

Classes ...

- Use `class` keyword
- Define constructor and methods inside
  - one constructor function per class
- Really just sugar over existing prototypal inheritance mechanism
  - creates a constructor function with same name as class
  - adds methods to prototype

```
class Shoe {
  constructor(brand, model, size) {
    this.brand = brand;
    this.model = model;
    this.size = size;
    Shoe.count++;
  }
  static createdAny() { return Shoe.count > 0; }
  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;

let s1 = new Shoe('Mizuno', 'Precision 10', 13);
let s2 = new Shoe('Nike', 'Free 5', 12);
let 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('s1.equals(s3) =', s1.equals(s3)); // true
```

class method

not a standard JS method

class property

… 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; }
}

let 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
```

value after **`extends`** can be an expression that evaluates to a class/constructor function

inherits both instance and static methods

inside **`constructor`**, **`super(args)`** calls the superclass constructor; can only call **`super`** like this in a constructor and only once

inside a method, **`super.name(args)`** calls the superclass method ***`name`***

- In subclasses, **`constructor`** **must** call **`super(args)`** and it must be **before** **`this`** is accessed because the highest superclass creates the object

`this` is not set until call to **`super`** returns

… Classes

- In a class with no **`extends`**,
  omitting **`constructor`** is the same as specifying
  `constructor() {}`
- In a class with **`extends`**,
  omitting **`constructor`** is the same as specifying
  `constructor(...args) { super(...args); }`
  rest spread
- Can extend builtin classes like **`Array`** and **`Error`**
  - requires JS engine support; transpilers cannot provide
  - instances of **`Array`** subclasses can be used like normal arrays
  - instances of **`Error`** subclasses can be thrown like provided **`Error`** subclasses
- Class definitions are
  - block scoped, not hoisted, and evaluated in strict mode
- For generator methods (discussed later), precede name with "**`*`** "

Getters and Setters

- ES5 supports these using `Object.defineProperty`/`defineProperties`
- ES 2015 supports `get` and `set` keywords in class definitions

```
class Shoe {
  ...
  get size() {
    return this._size;
  }
  set size(size) {
    this._size = size;
  }
  ...
}
let s = new Shoe();
s.size = 13; // invokes setter
console.log(s.size); // invokes getter
```

can do more here

using `size` instead of `_size` for the "backing field" would cause a `ModuleEvaluationError` with message "Maximum call stack size exceeded"

```
class Person {
  constructor(name) {
    this._name = name;
  }
  get name() {
    return this._name;
  }
}
let p = new Person('Mark');
console.log('name is', p.name); // Mark
p.name = 'Jason';
// throws ModuleEvaluationError
// with message "Cannot set property name
// of #<Person> which has only a getter
```

can use a `Symbol` in place of `_size` and `_name` to make them a non-enumerable properties

- ES5 also allows use `get` and `set` in object literals, but that seems less useful

ES5 vs. ES 2015 Functions

	ES5	ES 2015
normal function	**`function`**	**`function`** or arrow function
method	**`function`** on prototype	method in **`class`**
constructor	**`function`**	**`constructor`** in **`class`**

New `Math` Functions

- `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)`
  - `Math.log1p` returns `Math.log(1 + number)`
- hyperbolic trig functions - `Math.asinh(number)`, `Math.acosh(number)`, `Math.atanh(number)`

New `Number` Functions

- **`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 a float, **`NaN`**, **`Infinity`** or **`-Infinity`**
- **`Number.isNaN(n)`** - returns boolean indicating whether ***`n`*** is the special **`NaN`** value
- **`Number.isSafeInteger(n)`** - returns boolean indicating whether ***`n`***
  can be represented exactly in a double (within 53 bits)
  - also new constants **`Number.MIN_SAFE_INTEGER`** and **`Number.MAX_SAFE_INTEGER`**
- **`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

note how some of these are functions on other objects in ES5

Numeric Literals

- Hexadecimal
  - preceded with zero and **`x`**
  - **`0xa === 10`**
  - supported before ES 2015
- Octal
  - preceded with zero and **`o`**
  - **`0o71 === 57`**
- Binary
  - preceded with zero and **`b`**
  - **`0b1101 === 13`**
- When used in strings,
  all of these can be parsed with **`Number(s)`**

New `String` Methods

- ***s1*.startsWith(*s2*)** - determines if starts with given characters
- ***s1*.endsWith(*s2*)** - determines if ends with given characters
- ***s1*.includes(*s2*)** - determines if includes given characters
- ***s*.repeat(*count*)** - creates new string by copying ***s count*** times
- 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
    - to get length in code points, **[...*string*].length**
  - no easy way to get or create 4-byte characters in ES5
  - ***string*.codePointAt(*pos*)**
    gets UTF-16 integer value at a given position
    - to convert to hex, call **toString(16)** on this value
  - **String.fromCodePoint(*int1*, ..., *intN*)**
    returns string created from any number of UTF-16 integer values

can specify starting position of test for each of these

ES7 may add **trimLeft** and **trimRight** methods

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

new Unicode escape syntax inside literal strings for specifying a code point
\u{*code*}
(really include the braces)

Template Literals

- Surrounded by backticks
- Can contain any number of embedded expressions
  - `${expression}`

```
console.log(`${x} + ${y} = ${x + y}`);
```

- Can contain newline characters for multi-line strings

```
let greeting = `Hello,
World!`;
```

Tagged Template Literals ...

- Preceded by a function name that will produce a customized result
  - examples include special escaping (ex. HTML encoding), language translation, and DSLs
- Passed array of template strings outside expressions ("raw") and expression values as individual parameters ("cooked")

```
function upValues(strings, ...values) {
  let result = strings[0];
  values.forEach((value, index) =>
    result += value.toUpperCase() + strings[index + 1]);
  return result;
}
let firstName = 'Mark';
let lastName = 'Volkmann';
console.log(upValues `Hello ${firstName} ${lastName}!`);
// Hello MARK VOLKMANN!
```

In this example `strings` is `['Hello ', ' ', '!']` and `values` is `['Mark', 'Volkmann']`

- Provided template function `String.raw`
  - treats characters like `\n` as separate `\` and `n` characters

... Tagged Template Literals

```
function dedent(strings, ...values) {
  let last = strings.length - 1, re = /\n\s+/g, result = '';
  for (let i = 0; i < last; i++) {
    result += strings[i].replace(re, '\n') + values[i];
  }
  return result + strings[last].replace(re, '\n');
}

let homeTeam = 'Cardinals';
let visitingTeam = 'Cubs';
console.log(dedent `Today the ${homeTeam}
                    are hosting the ${visitingTeam}.`);

// If template starts with an expression, strings will start with ''.
// If template ends with an expression, strings will end with ''.
console.log(dedent `${homeTeam}
                    versus
                    ${visitingTeam}`);
```

Output

```
Today the Cardinals
are hosting the Cubs.
Cardinals
versus
Cubs
```

New `Array` Functions

- **`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 or an iterable
  - ***`mapFn`*** is an optional function that is called on each element to transform the value

New `Array` Methods

- ***arr*.copyWithin(*targetIndex*, *srcStartIndex*, [*srcEndIndex*])** - copies elements from ***srcStartIndex*** to ***srcEndIndex*** `- 1`, or to the end of the array, to ***targetIndex***, replacing existing elements
  - indexes can be negative to count from end
- ***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
- ***arr*.entries()** - returns an iterator over **[*index*, *value*]** pairs of ***arr***
- ***arr*.keys()** - returns an iterator over indices of ***arr***
- ***arr*.values()** - returns an iterator over values in ***arr***

same API as in **`Set`** and **`Map`**

New `Object` Functions ...

- **`Object.assign(target, src1, ... srcN)`**
  - copies properties from src objects to target (left to right), replacing those already present
  - returns ***`target`***
  - can create a shallow clone of an object

```
let copy = Object.assign({}, obj);
```

  - to create clone with same prototype

```
function clone(obj) {
  let proto = Object.getPrototypeOf(obj);
  return Object.assign(
    Object.create(proto), obj);
}
let copy = clone(obj);
```

  - can use in constructors to assign initial property values →

```
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

  - can use to add default properties to an object

```
const DEFAULTS = {
  color: 'yellow',
  size: 'large'
};
let obj = {size: 'small'};
obj = Object.assign({}, DEFAULTS, obj);
```

order is significant!

... New `Object` Functions

- **`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
- **`Object.getOwnPropertySymbols(obj)`**
  - returns array of symbol keys
  - alternative to existing `Object.keys` and `Object.getOwnPropertyNames` functions
  - also see functions on `Reflect` object (described next)

`Reflect` Functions

supported by Babel, but not Traceur

- `get(obj, propName)` - alternative to `obj[propName]`
- `set(obj, propName, value)` - alternative to `obj[propName] = value`
- `has(obj, propName)` - alternative to `propName in obj`
- `deleteProperty(obj, propName)` - alternative to `delete obj[propName]`

- `construct(ctorFn, args)` - alternative to using `new ctorFn(...args)`
- `apply(fn, thisValue, args)` - alternative to using `fn.apply(thisValue, args)`

- `getOwnPropertyDescriptor(obj, propName)` - similar to same function in `Object`
- `defineProperty(obj, propName, propAttrs)` - similar to same function in `Object`

- `getPrototypeOf(obj)` - same as function in `Object`
- `setPrototypeOf(obj, prototype)` - changes prototype of `obj`

- `ownKeys(obj)` - returns an array of string and symbol keys
- `enumerate(obj)` - returns an iterator over all string keys (not symbols) including those in prototype chain

---

- `isExtensible(obj)` - same as function in `Object`
- `preventExtensions(obj)` - similar to same function in `Object`

Getting Object Keys

	string keys	symbol keys	only own	only enumerable
`Object.keys`	√		√	√
`Object.getOwnPropertyNames`	√		√	
`Object.getOwnPropertySymbols`		√	√	
`Reflect.ownKeys`	√	√	√	
`Reflect.enumerate`	√			√

for-of Loops

- New way of iterating over elements in an "iterable"
  - for arrays, this is an alternative to for-in loop and `Array` `forEach` method
  - better because its use isn't restricted to arrays
- Iteration variable is scoped to loop
- Value after `of` can be any iterable (ex. an array)
  - cannot be an iterator

doesn't include `value` property when `next` is `true`

```
let stooges = ['Moe', 'Larry', 'Curly'];

for (let stooge of stooges) {
  console.log(stooge);
}

for (let [index, stooge] of stooges.entries()) {
  console.log(index, stooge);
}
```

can use `const` instead of `let`

Collections

- New collection classes include
  - `Set`
  - `Map`
  - `WeakSet`
  - `WeakMap`

`Set` Class ...

- Instances hold collections of unique values
  - when values are objects, they are compared by reference
- Values can be any type including objects and arrays

could store references to DOM nodes

- To create, `let mySet = new Set()`
  - can pass iterable object (such as an array) to constructor to add all its elements
- To add a value, `mySet.add(value);`

chain to add multiple values

- To test for a value, `mySet.has(value)`
- To delete a value, `mySet.delete(value)`
- To delete all values, `mySet.clear()`

... `Set` Class

- **`size`** property holds number of keys
- **`keys`** method returns iterator over values
- **`values`** method returns iterator over values
  - used by default in for-of loop
- **`entries`** method returns iterator over [value, value] pairs
- **`forEach`** method is like in that in **`Array`**, but passes ***`value`***, ***`value`***, and the **`Set`** to callback

these iterate in insertion order

methods for **`Set`** iteration treat sets like maps where corresponding keys and values are equal for API consistency

iterators are described later

Common `Set` Operations

Thanks Dr. Axel Rauschmayer

- All of these work by creating `Array`s from `Set`s, operating on them, and creating a new `Set`
- Map
```
let newSet = new Set([...set].map(elem => some-code));
```
- Filter
```
let newSet = new Set([...set].filter(elem => some-code));
```

also see `map` and `filter` generator functions later

- Union
```
let union = new Set([...set1, ...set2]);
```
- Intersection
```
let intersection = new Set([...set1].filter(elem => set2.has(elem)));
```
- Difference
```
let union = new Set([...set1].filter(elem => !set2.has(elem)));
```
- Remove duplicates from an array
```
let newArr = [...new Set(arr)];
```

`Set` Example

```
let colors = new Set();
colors.add('red');
colors.add('green');
colors.add('blue');
colors.add('red');

// Another way to populate a Set
let arr = ['red', 'green', 'blue', 'red'];
colors = new Set(arr);

console.log(colors.size); // 3
console.log(colors.has('red')); // true
console.log(colors.has('pink')); // false

console.log('\nkeys are:');
colors.forEach(key => console.log(key));
// red green blue
```

```
console.log('\nvalues are:');
for (let value of colors.values()) {
  console.log(value); // red green blue
}
for (let value of colors) { // same
  console.log(value); // red green blue
}

console.log('\nentries are:');
for (let entry of colors.entries()) {
  console.log(entry);
  // ['red', 'red']
  // ['green', 'green']
  // ['blue', 'blue']
}

colors.delete('red');
console.log(colors.size); // 2
console.log(colors.has('red')); // false
```

Map Class ...

- Instances hold key/value pairs where keys are unique
  - when keys are objects, they are compared by reference
- Keys and values can be any type including objects and arrays
  - differs from JavaScript objects in that keys are not restricted to strings
- To create, `let myMap = new Map()`
  - can pass iterable object to constructor to add all its pairs (ex. array of `[key, value]`)
- To add or modify a pair, `map.set(key, value)`
- To get a value, `myMap.get(key);`
  - returns `undefined` if not present
- To test for a key, `myMap.has(key)`
- To delete a pair, `myMap.delete(key)`
- To delete all pairs, `myMap.clear()`

could use DOM nodes as keys or values

chain to add/modify multiple values

… Map Class

- `size` property holds number of keys
- `keys` method returns iterator over keys
- `values` method returns iterator over values
- `entries` method returns iterator over `[key, value]` arrays
  - used by default in for-of loop
- `forEach` method is like in `Array`, but passes `value`, `key`, and the `Map` to callback

these iterate in insertion order

```
let teams = new Map();
teams.set('Chicago', 'Cubs');
teams.set('Kansas City', 'Royals');
teams.set('St. Louis', 'Cardinals');

for (let [city, name] of teams) {
  console.log(name, 'is from', city);
}
```

can omit `city` or `name` to iterate over just values or just keys

Common `Map` Operations

Thanks Dr. Axel Rauschmayer

- Map

an array of key/value arrays

```
let newMap = new Map([...map].map(
  ([key, value]) => [new-key-expr, new-value-expr]));
```

- Filter

```
let newMap = new Map([...map].filter(
  ([key, value]) => boolean-expr]));
```

also see `map` and `filter` generator functions later

Map Example

```
let teams = new Map();
teams.set('Chicago', 'Cubs');
teams.set('Kansas City', 'Royals');
teams.set('St. Louis', 'Cardinals');

// Another way to populate a Map
let arr = [
  ['Chicago', 'Cubs'],
  ['Kansas City', 'Royals'],
  ['St. Louis', 'Cardinals']
];
teams = new Map(arr);

console.log(teams.size); // 3
console.log(teams.has('St. Louis')); // true
console.log(teams.has('Los Angeles')); // false
console.log(teams.get('St. Louis')); // Cardinals

console.log('\\nkeys are:');
teams.forEach((value, key) => console.log(key));
// Chicago, Kansas City, St. Louis

// Another way to iterate over keys
for (let key of teams.keys()) {
  console.log(key);
  // Chicago, Kansas City, St. Louis
}
```

alternative is
`for (let [key] of teams)`

```
console.log('\\nvalues are:');
for (let value of teams.values()) {
  console.log(value);
  // Cubs, Royals, Cardinals
}
```

alternative is
`for (let [, value] of teams)`

```
console.log('\\nentries are:');
for (let entry of teams.entries()) {
  console.log(entry);
  // ['Chicago', 'Cubs']
  // ['Kansas City', 'Royals']
  // ['St. Louis', 'Cardinals']
}
for (let [city, team] of teams) { // same
  console.log(
    'The', team, 'plays in', city);
}

teams.delete('Chicago');
console.log(teams.size); // 2
console.log(teams.has('Chicago')); // false
```

`WeakSet` Class

supported by Babel, but not Traceur

- Similar API to `Set`, but differs in that
  - values must be objects
  - values are "weakly held",
    i.e. can be garbage collected if not referenced elsewhere
  - don't have a `size` property
  - can't iterate over values
  - no `clear` method to remove all values

`WeakMap` Class

supported by Babel, but not Traceur

- Similar API to `Map`, but differs in that
  - keys must be objects
  - keys are "weakly held",
    i.e. a pair can be garbage collected if the key is not referenced elsewhere
    - at that point the value can be garbage collected if not referenced elsewhere
  - don't have a `size` property
  - can't iterate over keys or values
  - no `clear` method to remove all pairs

Promises ...

- Proxy for a value that may be known in the future after an asynchronous operation completes
  such as a REST call
- Create with `Promise` constructor, passing it a function that takes `resolve` and `reject` functions, and calls one of them
- Register to be notified when promise is **resolved** or **rejected** with `then` and/or `catch` method
  - `then` method takes success and failure callbacks
    call omit one callback
  - `catch` method only takes failure callback
  - both return a `Promise` to support chaining
  - "success callback" is passed a value of any kind
  - "failure callback" is passed a "reason" which can be any kind of value, but is typically an `Error` object or a string

`.then(cb1, cb2)` is similar to `.then(cb1).catch(cb2)`, but differs in that `cb2` won't be invoked if `cb1` throws

ES 2016 will likely add `finally` method

... Promises ...

- Can call `then` on a promise after it has been resolved or rejected
  - the success or failure callback is called immediately
- Three possible states: pending, resolved, and rejected
  - once state is resolved or rejected, can't return to pending
  - "resolved" state is sometime called "fullfilled"

... Promises ...

```
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));
```

in real usage, some asynchronous operation would happen above

- Static methods
  - `Promise.resolve(value)` returns promise that is resolved immediately with given value
  - `Promise.reject(reason)` returns promise that is rejected immediately 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

… Promises …

- Supports chaining to reduce code nesting

```
asyncDouble(1).
  then(v => asyncDouble(v)).
  then(v => asyncDouble(v)).
  //then((v) => asyncDouble('bad')).
  then(v => console.log('success: v =', v)).
  catch(err => console.error('error:', err));
```

Output

```
success: v = 8
```

- Fine print
  - success callbacks should do one of three things
    - return a value, return the next promise to wait for, or throw
  - if a success callback returns a non-`Promise` value,
    it becomes the resolved value of the `Promise` returned by `then`
  - if a success callback returns a `Promise` value,
    the current promise resolves or rejects the same as it
  - if any `Promise` in the chain is rejected or throws,
    the next failure callback in the chain receives it
  - if a failure callback returns a value,
    it becomes the resolved value for the next success callback in the chain

Without promises,
using only callbacks,
if an async function throws,
the calling function cannot catch it
and the error is swallowed.

… Promises

- If an error is thrown inside a success or failure callback the promise returned by **then** is rejected

```
let p = asyncDouble(3).then(
  v => {
    // This causes the promise returned by
    // the call to then above to be rejected.
    throw 'Did you see this?';
  },
  err => console.error('error:', err)); // not reached

p.then(
  value => console.log('resolved with', value),
  reason => console.log('rejected with', reason));
// Output is "rejected with Did you see this?"
```

also see examples in
javascript-labs/es6/promises/chaining.js

Modules

- A JavaScript file that is imported by another is treated as a "module"
  - defined by a single, entire source file
  - contents are not wrapped in any special construct
  - also code in an HTML **`<module>`** tag is treated as a "module" (will anyone use this?)
- Modules typically export values to be shared with other files that import it
- Top-level variables and functions that are not exported are not visible in other source files (like in Node.js)
- Module code is evaluated in strict mode
- Cyclic module dependencies are supported
- Enables APIs to be defined in modules instead of global variables
  - eliminates need to use objects for namespaces - ex. **`JSON`** and **`Math`**
  - future versions of jQuery **`$`** and Underscore **`_`** will be defined using modules

simply containing **`import`** or **`export`** statements does not determine whether a file will be treated as a module; can't determine just by looking at the file

Modules - Exporting

- Can export any number of values from a module
  - values can be any JavaScript type including functions and classes
  - can optionally specify a default export
    which is actually a named export with the name "default"
- To define and export a value
  - `export let name = value;`
  - `export function name(params) { ... }`
  - `export class name { ... }`
- To export multiple, previously defined values
  - `export {name1, name2 as other-name2, ...};`

note ability to export a value under a different name

- To specify a default export
  - `export default expr;`
  - `export {name as default};`

same as previous line if value of *expr* is *name*

  - `export default function (params) { ... };`
  - `export default class { ... };`

Modules - Importing

- Can import values from other modules
- Imports are hoisted to top of file
- To import all exports into a single object
  - `import * as obj from 'module-path';`
  - bindings from imports like *obj* is read-only
- To import specific exports
  - `import {name1, name2 as other-name, ...} from 'module-path';`
- To import the default export
  - `import default-name from 'module-path';`
  - `import {default as default-name} from 'module-path';`
- To import the default export and specific exports
  - `import default-name, {name1, name2, ...} from 'module-path';`
- To import a module only for its side effects
  - `import 'module-path';`

module paths are relative to containing file; can start with `./` (the default) or `../`

note ability to import a value under a different name

same as previous line

More on Modules

- A module can export values from another module without first importing them
  - adds to its own exports
  - `export * from 'module-path';` exports everything exported by the given module
  - `export {name1, name2 as other-name} from 'module-path';`

- Module Loader API
  - supports conditionally loading modules
  - allows customized resolving of `'module-path'` strings (see `Reflect.Loader`)

```
System.import('module-path').
  then(theModule => { ... }).
  catch(err => { ... });
```

  - `System.import` returns a promise
    - can use `Promise.all` to wait for multiple modules to be loaded
  - there is much more to this!

Modules in Traceur ...

- To transpile ES 2015 files that use modules
  - transpile just main file to generate a single ES5 file that contains all required code
  - `traceur --out main.js --source-maps main6.js`
- Traceur generated source maps support modules
  - can step through each of the original ES 2015 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 in Traceur

bar6.js

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

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

foo6.js

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

export let 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 source map:

```
traceur --out gen/main.js \
--source-maps main6.js
```

Output:

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

Guy Bedford Rocks!

- **ES6 Module Loader** - https://github.com/ModuleLoader/es6-module-loader
  - "dynamically loads ES6 modules in browsers and NodeJS"
  - will track "JavaScript Loader Standard" at https://github.com/whatwg/loader

needed because browsers and Node.js don't support ES 2015 modules yet

- **SystemJS** - https://github.com/systemjs/systemjs
  - "universal dynamic module loader - loads ES6 modules (using **ES6 Module Loader**), AMD, CommonJS, and global scripts (like jQuery and lo-dash) in the browser and NodeJS."
  - dependency management handles circular references and modules that depend on different versions of the same module (like Node.js does)
  - supports "loading assets ... such as CSS, JSON or images"
- **jspm** - http://jspm.io and https://github.com/jspm
  - JavaScript Package Manager for **SystemJS**
  - "load any module format (ES6, AMD, CommonJS, and globals) directly from any endpoint such as **npm** and **GitHub**"
  - "custom endpoints can be created"
  - "for development, load modules as separate files with ES6"
  - "for production, optimize into a bundle ... with a single command"

all of these support Babel and Traceur

Using jspm ...

- **To install and configure jspm**
  - `npm install -g jspm`
  - `jspm init`
    - prompts and creates `package.json` and `config.js`
    - can accept all defaults
  - create `index.html`
  - setup a local file server
    - a good option is live-server
    - `npm install -g live-server`
    - `live-server`
  - browse localhost:8080
  - automatically transpiles using Traceur (default) or Babel
  - automatically generates sourcemaps
- **To install modules**

  lesser used modules require jspm configuration before they can be installed
  - for packages in npm
    - `jspm install npm:`*`module-name`* (ex. jsonp)
    - by default, installs in `jspm_packages/npm`
  - for packages in GitHub
    - `jspm install github:`*`module-name`*
    - by default, installs in `jspm_packages/github`
  - for well-known packages
    - `jspm install `*`module-name`*
    - includes angularjs, bootstrap, d3, jquery, lodash, moment, and underscore
    - see list at https://github.com/jspm/registry/blob/master/registry.json
  - adds dependencies to `package.json`
  - adds `System.config` call in `config.js`

… Using jspm

- **To reinstall all dependencies**
  - similar to npm, run **`jspm install`**
  - recreates and populates **`jspm_packages`** directory
  - recreates **`config.js`** if it is missing
- **To make your own packages compatible with jspm**
  - see https://github.com/jspm/registry/wiki/Configuring-Packages-for-jspm
  - can publish in npm or GitHub
  - allows others to install them using jspm
- **To bundle for production**
  - **`jspm bundle-sfx --minify`** ***`main`***
  - removes all dynamic loading and transpiling
  - generates **`build.js`** and **`build.js.map`**
  - replace all script tags in main HTML file with one for **`build.js`**
  - if using Traceur, add
    ```
    <script src="jspm_packages/traceur-runtime.js">
    </script>
    ```
  - there are other bundling options, but this seems like the best
  - won't be necessary in the future when browsers support HTTP2
    - will be able to download many files efficiently
    - today browsers limit concurrent HTTP requests to the same domain to 6 or 8

sfx is short for "self executing"

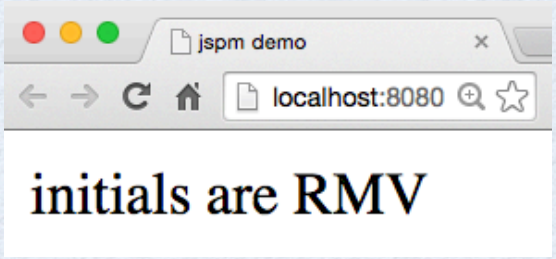
jspm Example #1

the basics plus a little jQuery

```
jspm install jquery
```

index.html

```
<!DOCTYPE html>
<html>
  <head>...</head>
  <body>
    <div id="content"></div>

    <!-- Enable ES 2015 module loading and more. -->
    <script src="jspm_packages/system.js"></script>

    <!-- Enable loading dependencies
         that were installed with jspm. -->
    <script src="config.js"></script>

    <!-- Load the main JavaScript file
         that can import others.  In this
         example, main.js is in same directory.
         Can also specify a relative directory path. -->
    <script>System.import('main');</script>
  </body>
</html>
```



may need `.js` file extension in next version of jspm

main.js

```
import $ from 'jquery';
import * as strUtil from './str-util';

$('#content').text('initials are ' +
  strUtil.initials(
    'Richard Mark Volkmann'));
```

str-util.js

```
export function initials(text) {
  return text.split(' ').
    map(word => word[0]).
    join('');
}
```

jspm Example #2

adds Bootstrap and more jQuery

```
jspm install bootstrap
```

index.html

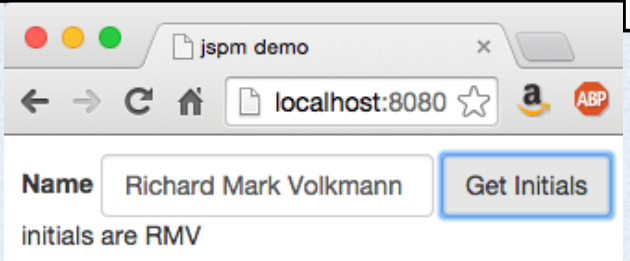
```
<!DOCTYPE html>
<html>
  <head>
    <title>jspm demo</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="main.css">
    <script src="jspm_packages/system.js"></script>
    <script src="config.js"></script>
    <script>System.import('main');</script>
  </head>
  <body>
    <label>Name</label>
    <input id="name-input"
      class="form-control"
      value="Richard Mark Volkmann">
    <button id="get-initials-btn"
      class="btn btn-default">
      Get Initials
    </button>
    <div id="content"></div>
  </body>
</html>
```

main.css

```
body {
  display: none;
  padding: 10px;
}

input.form-control {
  display: inline-block;
  vertical-align: middle;
  width: 180px;
}
```

main.js

```
import 'bootstrap';
import $ from 'jquery';
import * as strUtil from './str-util';

$('#get-initials-btn').click(() => {
  let name = $('#name-input').val();
  let initials = strUtil.initials(name);
  $('#content').text(
    'initials are ' + initials);
});

$('body').show();
```



str-util.js

```
export function initials(text) {
  return text.split(' ').
    map(word => word[0]).
    join('');
}
```

Iterators and Iterables

- Iterators are objects that visit elements in a sequence
  - not created with a custom class; can be any kind of object
  - have a `next` method, described on next slide
- Iterables are objects that have a method whose name is the value of `Symbol.iterator`
  - this method returns an iterator
- An object can be both an iterable and an iterator
  - `obj[Symbol.iterator]() === obj` and `obj` has a `next` method

Iterator `next` Method

- Gets next value in sequence
- Returns an object with **`value`** and **`done`** properties
- If end of sequence has been reached, **`done`** will be true
  - can omit otherwise
- Whether **`value`** has meaning when **`done`** is **`true`** depends on the iterator
  - but the for-of loop, spread operator, and destructuring will ignore this value
  - can omit **`value`** property

using **`value`** when **`done`** is **`true`** is primarily useful in conjunction with **`yield*`** in a generator

Why return a new object from `next` method instead of returning the same object with modified `value` and `done` properties?

It is possible for an iterator to be used by more than one consumer and those consumers could access the object returned by `next` asynchronously.
If each call doesn't return a new object, its properties could be modified after the object is received, but before it checks the properties.
While this is a rare situation, implementers of iterators can't be sure how they will be used.

From Allen Wirfs-Brock ... "The specification of the Iterator interface does not require that the 'next' method return a fresh object each time it it called. So a userland iterator would not be violating anything by reusing a result object.

However, the specifications for all ES2015 built-in iterators require that they return fresh objects.

None of the built-in consumers of the Iterator interface (for-of, Array.from, etc.) retain references to IteratorResult objects after testing for 'done' and accessing the 'value', so semantically they don't care whether the ResultObject is reused. However, such reuse might preclude some otherwise plausible engine level optimizations."

Iterable Objects ...

- Objects from these builtin classes are iterable
  - `Array` - over elements
  - `Set` - over elements
  - `Map` - over key/value pairs as `[key, value]`
  - DOM `NodeList` - over `Node` objects (when browsers add support)
- Primitive strings are iterable
  - over Unicode code points
- These methods on `Array` (including typed arrays), `Set`, and `Map` return an iterator
  - `entries` - over key/value pairs as `[key, value]`
  - `keys` - over keys
  - `values` - over values

for arrays, keys are indices;
for sets, keys are same as values

objects returned are both iterators and iterable

- Custom objects can be made iterable
  - by adding `Symbol.iterator` method

… Iterable Objects

- To get an iterable represention (an `Array`) of an array-like object
  - `let iterable = Array.from(arrayLike)`
- **Ordinary objects** such as those created from object literals are **not iterable**
  - when this is desired, use `Map` class instead **or** write a function like the following

this serves as an example of how to implement an iterator

to exclude symbol keys, use `Object.getOwnPropertyNames(obj)`

```
function objectEntries(obj) {
  let index = 0;
  let keys = Reflect.ownKeys(obj); // gets both string and symbol keys
  return { // the iterable and iterator can be same object
    [Symbol.iterator]() { return this; },
    next() {
      if (index === keys.length) return {done: true};
      let k = keys[index++], v = obj[k];
      return {value: [k, v]};
    }
  };
}

let obj = {foo: 1, bar: 2, baz: 3};
for (let [k, v] of objectEntries(obj)) {
  console.log(k, 'is', v);
}
```

```
// Using a generator
function* objectEntries(obj) {
  let keys = Reflect.ownKeys(obj);
  for (let key of keys) yield([key, obj[key]]);
}
```

can get an iterable for keys in an object with `Reflect.enumerate(obj);`

Iterable Consumers

- **`for-of`** loop
  - `for (let value of someIterable) { ... } // iterates over all values`
- spread operator
  - can add all values from iterable into a new array
    - `let arr = [firstElem, ...someIterable, lastElem];`
  - can use all values from iterable as arguments to a function, method, or constructor call
    - `someFunction(firstArg, ...someIterable, lastArg);`
- positional destructuring
  - `let [a, b, c] = someIterable; // gets first three values`
- **`Set`** constructor takes an iterable over values
- **`Map`** constructor takes an iterable over key/value pairs
- **`Promise`** methods **`all`** and **`race`** take an iterable over promises
- In a generator, **`yield*`** yields all values in an iterable one at a time

will make sense after generators are explained

Iterable/Iterator Example #1

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

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

iterators can also be implemented with generators - see slide 92

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

skipping initial values of 0 and 1 and starting at the second 1

stops iterating when **done** is true which never happens here

Iterable/Iterator Example #2

```
let arr = [1, 2, 3, 5, 6, 8, 11];
let 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)); // 1 3 5 11

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

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

Generators

- **Generator functions**
  - return a **generator** which is a special kind of **iterator**
    - and same object is an iterable (has `Symbol.iterator` method)
  - can be paused and resumed via multiple return points, each specified using `yield` keyword
  - each `yield` is hit in a separate call to `next` method
  - exit by
    - running off end of function
    - returning a specific value using `return` keyword
    - throwing an error

`yield` keyword can only be used in generator functions

`done` will be `true` after any of these and will remain `true`

- Can use as a **producer**
  - get values from a sequence one at a time by calling `next` method
  - supports lazy evaluation and infinite sequences
- Can use as a **consumer**
  - provide data to be processed by passing values one at a time to `next` method

Defining Generators

- `function* name(params) { code }`
  - *`code`* uses `yield` keyword to return each value in sequence, often inside a loop
- Can define **generator methods** in class definitions
  - precede method name with `*`
  - ex. to make instances iterable using a generator,
    `* [Symbol.iterator]() { code }`
    - code would `yield` each value in the sequence

Generator Methods

called on a generator object returned by a generator function

typically these methods are not used directly

- **`next(value)`** method
  - gets next value, similar to iterator **`next`** method
  - takes optional argument, but not on first call
    - specifies value that the **`yield`** hit in this call will return at start of processing for next call
- **`return(value)`** method used on slide 95
  - terminates generator from the outside just as if the generator returned the specified value
  - returns **`{value: value; done: true}`**
- **`throw(error)`** method used on slide 95
  - throws error inside generator at **`yield`** where execution paused
  - if generator catches error and yields a value, generator is not terminated yet
  - otherwise generator is terminated and this method returns **`{value: undefined; done: true}`**

from Dr. Axel Rauschmayer, "The only iterating mechanism that lets you access the "done value" is **`yield*`**. All other constructs (`for-of`, spread, destructuring, …) ignore it. Its main purpose is to let **`yield*`** make recursive generator calls with results, without disrupting normal yielding. That is, for normal iteration it is an out-of-band value."

Steps to Use Generators

- 1) Call generator function to obtain generator
- 2) Call generator **`next`** method to request next value
  - optionally pass a value that the generator can use, possibly to compute subsequent value
    - but not on first call
  - after generator "yields" next value, its code is "suspended" until next request
- 3) Process value
  unless **`done`** property is **`true`** (typically)
- 4) Repeat from step 2
  unless **`done`** property is **`true`**

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()) {
  // process v
}
```

call

Basic Generator

```
function* myGenFn() {
  yield 1;
  yield 2;
  return 3;
}

let myGen = myGenFn();
console.log(myGen.next()); // {"value":1,"done":false}
console.log(myGen.next()); // {"value":2,"done":false}
console.log(myGen.next()); // {"value":3,"done":true}

for (let n of myGenFn()) {
  console.log(n); // 1, then 2, not 3
}
```

without **return** statement in **myGenFn**, this disappears

Infinite Generator

- To return a "normal" value
  - `yield value;`

if a generator returns a value `v` using the `return` keyword, the `next` call that trigger that will return `{value: v, done: true}`

Don't think of the returned value as another value in the sequence produced by yields. It can be a different "category" of data. For example, the returned value could be a final result and the yielded values could be intermediate results.

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

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

compare to slide 84

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

```
// Iterables can be
// implemented with generators.
let fib = {
  * [Symbol.iterator]() {
    let [prev, curr] = [0, 1];
    while (true) {
      [prev, curr] = [curr, prev + curr];
      yield curr;
    }
  }
};

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

- To yield each value returned by an iterable one at a time
  - `yield* some-iterable;`
  - can obtain an iterable by calling another generator function - `otherGenerator(params);`

can use to iterate over a tree structure; can make recursive calls to the same or a different generator function

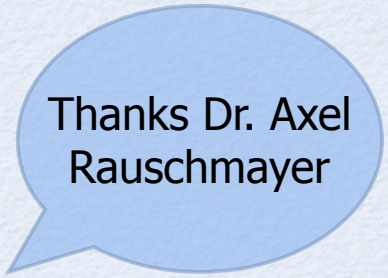
More Generator Examples

```
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);
  }
}

let iter = gen2(1); // can pass value to generator function,
let result = iter.next(); // but can't pass in first call to next
console.log(result.value); // foo1; result.done is false

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

//iter.throw('stop now'); // triggers catch in gen2

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

result = iter.next(4);
console.log(result.done ? 'done' : result.value); // done
```

map/filter Any Iterable

```
function* map(iterable, fn) {
  for (let elem of iterable) {
    yield fn(elem);
  }
}

function* filter(iterable, fn) {
  for (let elem of iterable) {
    if (fn(elem)) yield elem;
  }
}

let arr = [1, 2, 3];
let double = x => x * 2;
let isOdd = x => x % 2;

console.log('doubled');
for (let elem of map(arr, double)) {
  console.log(elem); // 2, 4, 6
}

console.log('\nodds');
for (let elem of filter(arr, isOdd)) {
  console.log(elem); // 1, 3
}
```



Generators For Async ...

workflow6.js

```
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) {
  let gen = generatorFn();
  function success(result) {
    let obj = gen.next(result);
    // obj.value is a promise
    // obj.done will be true if gen.next is called after
    // the last yield in workflow (on next slide) has run.
    if (!obj.done) obj.value.then(success, failure);
  }
  function failure(err) {
    let obj = gen.throw(err);
    // obj.value is a promise
    // obj.done will be false if the error was caught and handled.
    if (!obj.done) obj.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. There are libraries that provide this function.

compare to slide 100

... 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
  let 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 ES 2016 keywords!

Proxies ...

- Can intercept all operations whose names match functions on the `Reflect` object
  - see slide 50
  - can provide additional or alternate functionality
- Uses new `Proxy` class
  - constructor takes "target" (the object for which operations are to be intercepted) and "handler" (an object that defines alternate handling)
- Must use proxy object instead of target object or interceptions won't occur!
- Methods called on proxy that aren't defined there are forwarded to the target object
- Can create proxies that can be later turned off (revoked)
  - after being revoked, calls on proxies object are just forwarded to target
- Proxies can be the prototype of other objects
- Support - currently only Firefox; no transpilers

... Proxies

```
var obj = {
  p1: 'some value',
  m1: () => 'm1 result',
  m2: () => 'm2 result'
};

var proxy = new Proxy(obj, {
  get: (target, key) => {
    console.log('intercepted get for key =', key);
    var value = target[key];
    return value === undefined ? () => 'missing method ' + key :
      typeof value === 'string' ? value.toUpperCase() :
      value;
  },
  set: (target, key, value) => {
    console.log('intercepted set for key =', key);
    target[key] = value;
  }
});

// Replace a method on obj with a proxy for it.
obj.m1 = new Proxy(obj.m1, {
  apply: (fn, target, args) => {
    console.log('intercepted call to function', fn);
    var result = fn.apply(target, args);
    return typeof result === 'string' ? result.toUpperCase() : value;
  }
});

proxy.p1 = 'other value';
console.log('proxy.p1 =', proxy.p1);
console.log('obj.p1 =', obj.p1);

console.log('proxy.m1() =', proxy.m1()); // has a proxy
console.log('proxy.m2() =', proxy.m2()); // doesn't have a proxy

console.log(proxy.makeMeUpOnTheFly());
```

At the time this was written, only Firefox supported proxies. However, there were other ES 2015 features it did not yet support such as "let" and enhanced object literals.

Proxies **can't distinguish** between gets for **property lookup** and gets for **method calls**, so "method missing" can only be implemented if it can be assumed that all missing property lookups should provide a method. It could only supply methods for key names that match a certain pattern.

This works because functions are objects.

Output

```
intercepted set for key = p1
intercepted get for key = p1
proxy.p1 = OTHER VALUE
obj.p1 = other value

intercepted get for key = m1
intercepted call to function function obj.m1()
proxy.m1() = M1 RESULT

intercepted get for key = m2
proxy.m2() = m2 result

intercepted get for key = makeMeUpOnThFly
missing method makeMeUpOnTheFly
```

Tail Call Optimization

- Makes it possible to avoid growing the call stack when making recursive calls or invoking callback functions

  continuation passing style (CPS)

  - otherwise could exceed maximum call stack allowed
  - alternative to recursion is using loops
- Possible when the last operation in a function is a function call

```
function fac(n, acc) {
  return n == 0 ? acc : fac(n - 1, acc * n);
}
function factorial(n) {
  return fac(n, 1);
}
```

  - translates to →

```
// This version can't use TCO because
// multiplication occurs AFTER the recursive call.
function factorial(n) {
  return n <= 1 ? n : n * factorial(n - 1);
}
```

- Support - currently only Babel; no browsers

```
"use strict";

function fac(_x, _x2) {
  var _again = true;

  _function: while (_again) {
    _again = false;
    var n = _x,
        acc = _x2;
    if (n == 0) {
      return acc;
    } else {
      _x = n - 1;
      _x2 = acc * n;
      _again = true;
      continue _function;
    }
  }
}

function factorial(n) {
  return fac(n, 1);
}
```

a label

why not drop the `_again` flag, label, and `continue` and change the loop condition to `true`?

What's Next?

- The next version is always referred to as "JS-next"
- Currently that is ES 2016 (7th edition)
- Will include
  - **`async`** and **`await`** keywords
  - type annotations (like TypeScript)
  - new `Object` method `observe`
  - array comprehensions
  - generator comprehensions
  - value objects - immutable datatypes for representing many kinds of numbers
  - more

`async` and `await` ...

- New keywords
  - already supported by Babel and Traceur
  - JSHint doesn't recognize these yet
- Hides 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 slides 93-94

async function

function that returns a promise

"normal" function

Can 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() {
  let 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();
```

runs in next turn of event loop

Type Annotations ...

- 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`**
- To use in Traceur, enable experimental mode
  - supports specifying types, but doesn't enforce them yet
- See http://wiki.ecmascript.org/doku.php?id=strawman:types&s=types

... Type Annotations

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

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

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

```
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);
  }
}

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

Summary

- Which features of ES 2015 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
  - enhanced object literals
  - `for-of` loops
  - iterators and iterables
  - generators
  - promises
  - rest parameters
  - spread operator
  - template literals
  - new methods in `Array`, `Math`, `Number`, `Object`, and `String` classes