Code.Hub

The first Hub for Developers

Ztoupis Konstantinos

ES6 and beyond

Code.Learn Program:
React

Table of Contents

ECMAScript

- Defined by **European Computer Manufacturers Association** (ECMA)
- Specification is called **ECMAScript** or ECMA-262
  - JavaScript 5.1 (**ES5**) - https://www.ecma-international.org/ecma-262/5.1/
  - JavaScript 6 (**ES6**) - https://www.ecma-international.org/ecma-262/6.0/
- **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. ES6

- ECMAScript 5 did not add any new syntax
- ECMAScript 6 does!
- ES6 is backward compatible with ES5, which is backward compatible with ES3
- Many ES6 features provide
- **syntactic sugar** for more concise code
- Spec sizes
  - **ES5 - 258 pages**
  - **ES6 - 652 pages**
- One goal of ES6 and beyond is to make JavaScript a **better target for compiling to from other languages**

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 ES6 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 ES6 code to ES5

ES6 Transpilers

- **Traceur - 64%**
  - 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/
- **Babel - 76%**
  - aims to generate ES5 code that is as close a possible to the input ES6 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

ES6 Transpilers

- **TypeScript - 9%**
  - 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 ES6 generates source maps
  - to install, **npm install -g typescript**
  - to compile, **tsc some-file.ts**
    - generates **some-file.js**
  - http://www.typescriptlang.org

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/compat-table/es6/

Use ES6 Today?

checking the current feature-wise support for all engines

	98%	Compilers/polyfills							Desktop browsers																				Servers/runtimes															Mobile		
		56%	71%	71%	72%	50%	69%	17%	5%	11%	96%	96%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	98%	99%	99%	99%	99%	66%	95%	59%	97%	97%	98%	98%	98%	98%	2%	26%	7%	28%	98%	98%	99%	97
Feature name	Current browser	Traceur	Babel 6 + core-js 2	Babel 7 + core-js 2	Babel 7 + core-js 3	Closure 2019.07	Type-Script + core-js 3	es6-shim	Konq 4.14[1]	IE 11	Edge 17	Edge 18	FF 60 ESR	FF 67	FF 68 ESR	FF 69 Beta	FF 70 Nightly	CH 75, OP 62	CH 76, OP 63	CH 77, OP 64	CH 78, OP 65	SF 12	SF 12.1	SF 13 Beta	SF TP	WK	Echo JS	XS6	JXA	Node >=6.5 <7[3]	Node >=8.10 <9[3]	Node >=10.9 <11[3]	Node 11[3]	Node 12.0-12.4[3]	Node 12.5+[3]	DUK 1.8	DUK 2.3	JJS 1.8	JJS 10	GraalVM 19.0.0[4]	iOS 12	iOS 12.2	Samsung 8.2
Optimisation																																												
proper tail calls (tail call optimisation)	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	0/2	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	0/2	0/2	0/2	2/2	2/2	0/2
Syntax																																												
default function parameters	7/7	4/7	4/7	4/7	4/7	5/7	5/7	0/7	0/7	0/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	4/7	7/7	0/7	7/7	7/7	7/7	7/7	7/7	7/7	0/7	0/7	0/7	4/7	7/7	7/7	7/7	7/7
rest parameters	5/5	4/5	3/5	3/5	3/5	2/5	4/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	3/5	5/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	0/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5
spread syntax for iterable objects	15/15	15/15	13/15	13/15	14/15	11/15	14/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	10/15	15/15	11/15	15/15	15/15	15/15	15/15	15/15	15/15	0/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15
object literal extensions	6/6	6/6	6/6	6/6	6/6	5/6	6/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	5/6	6/6	5/6	6/6	6/6	6/6	6/6	6/6	6/6	0/6	4/6	0/6	2/6	6/6	6/6	6/6	6/6
for..of loops	9/9	9/9	9/9	9/9	9/9	6/9	9/9	0/9	0/9	0/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	7/9	9/9	8/9	9/9	9/9	9/9	9/9	9/9	9/9	0/9	0/9	0/9	4/9	9/9	9/9	9/9	9/9
octal and binary literals	4/4	2/4	4/4	4/4	4/4	2/4	4/4	2/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	2/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	0/4	4/4	0/4	2/4	4/4	4/4	4/4	4/4
template literals	7/7	6/7	6/7	6/7	6/7	5/7	5/7	0/7	0/7	0/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	6/7	6/7	7/7	7/7	7/7	6/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	0/7	0/7	0/7	5/7	7/7	6/7	6/7	7/7
RegExp "y" and "u" flags	6/6	4/6	4/6	4/6	4/6	0/6	0/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	2/6	2/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	0/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6
destructuring, declarations	22/22	20/22	21/22	21/22	21/22	20/22	21/22	0/22	0/22	0/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	12/22	21/22	19/22	22/22	22/22	22/22	22/22	22/22	22/22	0/22	0/22	0/22	0/22	22/22	22/22	22/22	22/22
destructuring, assignment	24/24	23/24	24/24	24/24	24/24	22/24	24/24	0/24	0/24	0/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	14/24	24/24	21/24	24/24	24/24	24/24	24/24	24/24	24/24	0/24	0/24	0/24	0/24	24/24	24/24	24/24	24/24
destructuring, parameters	24/24	19/24	21/24	21/24	21/24	20/24	21/24	0/24	0/24	0/24	23/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	12/24	23/24	18/24	24/24	24/24	24/24	24/24	24/24	24/24	0/24	0/24	0/24	0/24	24/24	24/24	24/24	24/24
Unicode code point escapes	2/2	1/2	1/2	1/2	1/2	1/2	1/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	2/2	0/2	0/2	2/2	2/2	2/2	2/2
new.target	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	1/2	0/2	0/2	2/2	2/2	2/2	2/2
Bindings																																												
const	18/18	16/18	16/18	16/18	16/18	16/18	16/18	0/18	2/18	14/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	10/18	18/18	12/18	18/18	18/18	18/18	18/18	18/18	18/18	1/18	4/18	12/18	16/18	18/18	18/18	18/18	18/18
let	14/14	12/14	12/14	12/14	12/14	12/14	12/14	0/14	0/14	12/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	10/14	14/14	0/14	14/14	14/14	14/14	14/14	14/14	14/14	0/14	0/14	10/14	12/14	14/14	14/14	14/14	14/14
block-level function declaration[18]	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Functions																																												
arrow functions	13/13	11/13	9/13	9/13	9/13	10/13	9/13	0/13	0/13	0/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	7/13	12/13	0/13	13/13	13/13	13/13	13/13	13/13	13/13	0/13	0/13	0/13	6/13	13/13	13/13	13/13	13/13
class	24/24	17/24	19/24	19/24	19/24	14/24	19/24	0/24	0/24	0/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	22/24	24/24	18/24	24/24	24/24	24/24	24/24	24/24	24/24	0/24	0/24	0/24	0/24	24/24	24/24	24/24	24/24
super	8/8	7/8	4/8	4/8	4/8	7/8	7/8	0/8	0/8	0/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	6/8	8/8	7/8	8/8	8/8	8/8	8/8	8/8	8/8	0/8	0/8	0/8	0/8	8/8	8/8	8/8	8/8
generators	27/27	24/27	24/27	24/27	24/27	19/27	22/27	0/27	0/27	0/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	16/27	27/27	0/27	27/27	27/27	27/27	27/27	27/27	27/27	0/27	0/27	0/27	0/27	27/27	27/27	27/27	27/27

https://kangax.github.io/compat-table/es6/

Code.Hub

Traceur

- **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
  - **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 --out** *es5-file-path es6-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

Babel

- **Implemented in ES6** and uses itself to transpile to ES5 code that runs on Node.js
- http://babeljs.io
- **Online tool** at http://babeljs.io/repl/
  - 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 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**

Running Babel

- To get help on options
  - **babel --help**
- To run code in an ES6 file
  - **babel-node** *es6-file-path*
  - file extension can be omitted and defaults to **.js**
- To compile an ES6 file to an ES5 file
  - **babel** *es6-file-path* **-o** *es5-file-path*
- To compile a many ES6 files to ES5 files
  - **babel** *es6-dir* **-o** *es5-dir*
- Experimental features
  - to use some ES7 features, add **--experimental** option
- Optional babel-runtime
  - https://babeljs.io/docs/en/babel-plugin-transform-runtime

Source Maps

- 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 **--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 ES6 code
  - open Developer Tools (cmd-option-i on Mac, ctrl-shift-i on Windows)
  - select ES6 .js files from “Sources” tab
  - set breakpoints and refresh page

Using Source Maps

- In Firefox
  - open Firefox debugger by selecting Tools ... Web Developer ... Debugger (cmd-option-s on Mac, ctrl-shift-c on Windows)
  - click gear icon in upper-right to see “Debugger Options” and verify that “Show Original Sources” is selected
  - select ES6 .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 ES6
  - JSHint - http://jshint.org: has good support now using "**esnext**" option
  - ESLint - http://eslint.org: added support ES6; needs more testing
- Recommend: use JSHint to check ES6 code

Automation

- **Grunt** - http://gruntjs.com
  - great tool for automating web development tasks
  - 4,472 plugins available as of 2015
    - 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**
    - 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

Automation

- **Gulp** - http://gulpjs.com
  - similar in goal to Grunt, but configuration is different
  - 1,457 plugins available as of 2015
  - also supports watch and livereload
  - emphasizes use of file streaming for better efficiency
  - see gulp-traceur and gulp-babel plugins

ES6 Features

- The following slides describe most of the features in ES6 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

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

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

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

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

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)); // Tue Apr 16 2019
console.log(makeDate(16)); // Fri Aug 16 2019
```

- Default value expressions can return to preceding parameters
- Explicitly passing undefined triggers use of default value
- Idiom for required parameters (from Allen Wirfs-Brock)

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

Rest Operator

- Gather 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', 'Hofmann', 'yellow');
// Mark Hofmann likes the color yellow.
report('Paul', 'Falag', 'pink', 'blue');
// Paul Falag 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
- Mostly removes need to use **Function** **apply** method

```
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
```

Destructuring

- Assigns values to any number of variables from values in arrays 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;
```

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

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

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

Destructuring

- LHS variables can specify default values
  - default values can refer to variables that precede their variable
- Positional destructuring can use rest operator for last variable
- 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
- Useful with functions that have multiple return values
  - really one array or object

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

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

```
[a, b] = [b,
```

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{1,4})/;
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'
});
```

Destructuring

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

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

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

Arrow Functions

- (*params*) => { *expressions* }
  - if only one parameter and not using destructuring, can omit parens
  - need parens if no parameters
  - 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

```
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

- 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 methods, only plain functions
- Also provides “lexical super”
- Immediately invoked functions (IIFEs)
  - not typically needed in ES6 since modules provide file scope
  - can write like this:

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

  - ending like this is a syntax error:

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

Symbols

- Immutable identifiers that are guaranteed to be unique
  - unlike strings
- To create a 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 retrieve description
  - ***sym*.toString()** or **String(*sym*)**
  - returns **'Symbol(*description*)'**
  - concatenating a symbol to a string throws **TypeError**
- A new primitive type
  - **typeof *sym* === 'symbol'**

Symbols

```
// Using computed
// property syntax  let obj = {
  [sym1]: value,
  [sym2](params) {
    ...
  }
};
```

- Can use as object keys
  - ***obj*[*sym*] = *value*;**
- They become non-enumerable properties
  - **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**

Public Symbols

- There are several predefined symbols that can be used as method names to customize how JavaScript treats specific objects
- To customize instanceof, implement Symbol.hasInstance method
- To customize conversion to a primitive, implement Symbol.toPrimitive method
- To customize conversion to a string, implement Symbol.toStringTag method
- To make an object “iterable”, implement Symbol.iterator method

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}
```

Enhanced Object Literals ...

- Computed properties names can be specified inline

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

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

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
- Code in class definition is always evaluated in strict mode

Classes

```
class Shoe {
  constructor(brand, model, size) {
    this.brand = brand;
    this.model = model;
    this.size = size;
    Shoe.count += 1;
  }
  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;
```

Classes

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

Classes

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

```
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( 'Nire', '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
```

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 specifyin **constructor(...args) { super(...args); }**

- 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

- Precede method names with “* ” for generators

Getters and Setters

- ES5 supports these using **Object.defineProperty/defineProperties**
- ES6 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
```

```
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
```

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

ES5 vs. ES6 Functions

	ES5	ES6
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

Numeric Literals

- Hexadecimal
  - preceded with zero and **x**
  - **0xa === 10**
  - supported before ES6
- 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*)**

Template Strings

- String interpolation
- Surrounded by backticks
- Can contain newline characters for multi-line strings

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

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

Can contain any number of embedded expressions

${*expression*}

Tagged Template Strings

- 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 = 'Voidman';
console.log(upValues `Hello ${firstName} ${lastName}!`);
// Hello MARK VOIDMAN!
```

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

Code.Hub

Tagged Template Strings

```
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];
}
let homeTeam = 'Cardinals';
let visitingTeam = 'Cubs';
console.log(dedent `Today the ${homeTeam}
                    are hosting the ${visitingTeam}.`);
// Outputs
// If template starts with an expression, strings will start with ''.
// If template ends with an expression, strings will end with ''.
console.log(dedent `${homeTeam}
                    versus
                    ${visitingTeam}`);
```

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
  - ***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 the [***index***, ***value***] pairs of ***arr***
- ***arr*.keys()** - returns an iterator over the indices of ***arr***
- ***arr*.values()** - returns an iterator over the values in ***arr***

New Object Functions

- **Object.assign(*target*, *src1*, ... *srcN*)**
  - copies properties from src objects to target, replacing those already present
  - returns ***target***
  - can use to create a shallow clone an object
  - to create a clone with the same prototype

```
let copy = Object.assign({},
```

```
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 can use to add default values to an object

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

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

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**
- **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

Reflect Functions

- **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	√			√

Hub

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
- Iteration variable is scoped to loop
- Value after **of** can be anything that is iterable such as an array
  - iterators are described later

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

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
- To create, **let mySet = new Set()**
  - can pass iterable object (such as an array) to constructor to add all its elements
- To add an element, **mySet.add(*value*);**
- To test for element, **mySet.has(*value*)**
- To delete an element, **mySet.delete(*value*)**
- To delete all elements, **mySet.clear()**

Set Class

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

Common Set Operations

- 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));`
- 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 arra `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 key, myMap.has(*key*)
- To delete a pair, myMap.delete(*key*)
- To delete all pairs, myMap.clear()
- size property holds number of keys

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 array of [*key*, *value*] arrays
  - used by default in for-of loop
- forEach method is like in Array, but passes *value*, *key* and *map* to callback

Common Map Operations

Map

```
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]));
```

Code.Hub

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
}
```

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

- 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 elements

WeakMap Class

- Similar API to Map, but differs in that
  - keys must be objects
  - keys are “weakly held”
    - 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
- 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 or catch method
  - then method takes success and failure callbacks and returns a Promise to support chaining
  - catch method only takes failure callback
  - “success callback” is passed a value of any kind
  - “failure callback” is passed a “reason” which can be an Error object or a string
- 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, fullfilled, and rejected
  - once state is fullfilled or rejected, can’t return to pending

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));
```

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 reduces 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));
```

- Some fine print
  - 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, it becomes the Promise returned by then
  - 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 resolve value for the next success callback in the chain

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?"
```

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”
- 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 by default
  - no need to specify 'use strict';
- Supports cyclic module dependencies
- 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

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 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*, ...};
- To specify a default export
  - export default *expr*;
  - export {*name* as default};
  - export default function (*params*) { ... };
  - export default class { ... };

Modules - Importing

- Can import values from other modules
- To import all exports into a single object
  - import * as obj from 'module-path';
  - obj is read-only
- To import specific exports
  - import {name1, name2 as other-name, ...} from 'module-path';
- To import the default export
  - import name from 'module-path';
  - import {default as 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';

More on Modules

- A module can export values from another module without first importing them
  - adds to its own exports
  - export * from 'module-path';
  - 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 ES6 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 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 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/main6.js"></script>
  </head>
  <body>
    See console output.
  </body>
</html>
```

Useful packages

- 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
- 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”

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
  - 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 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

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

jspm Example #1

jspm install jquery

index.html

```
<!DOCTYPE html>
<html>
  <head>...</head>
  <body>
    <div id="content"></div>
    <!-- Enable ES6 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>
```

main.js

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

$('#content').text('initials are ' +
strUtil.initials('Rick Mack Vann'));
```

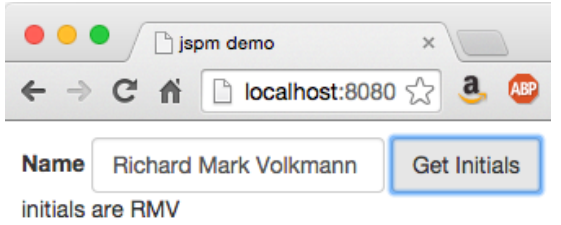
str-util.js

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

de.Hub

jspm Example #2

jspm install bootstrap



index.html

```
....
    <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>
```

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 can visit elements in a sequence
  - not created with a custom class; can be Object
  - have a next method and optional return and throw methods
    - described on next slide
- Iterables are objects that have a method whose name is the value of Symbol.iterator
  - this method returns an iterator

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 a new object with value and done properties
  - done will be true if end of sequence has been reached; can omit if false
  - when done is true, value is not valid; typically undefined; can omit
- return method (optional)
  - called if iteration ends before iterator returns done: true
  - can end iteration with break, return, throw, and continue (with label of outer loop; rarely used)
  - allows iterator to clean up (ex. close files)
- throw method (optional)
  - takes error argument and throws it inside generator function that created the iterator
  - can catch inside generator function

Iterable Objects

- Objects from these builtin classes are iterable
  - Array - over elements
  - Map - over key/value pairs as [*key*, *value*]
  - Set - over elements
  - DOM NodeList - over Node objects (coming soon)
- Primitive strings are iterable
  - over Unicode code points
- These methods on Array (including typed arrays), Map, and Set return an iterable
  - entries - over key/value pairs as [*key*, *value*]
  - keys - over keys
  - values - over values
- Custom objects can be made iterable
  - by adding Symbol.iterator method

Iterable Objects

```
function objectEntries(obj) {
  let index = 0;
  let keys = Reflect.ownKeys(obj); // gets both string and symbol
keys
  return { // note how 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);
}
```

To get an iterable represention 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 new Map class instead or write a function like the following

Iteratable Consumers

- for-of loop
  - for (let *value* of *someIterable*) { ... } // iterates over all values
- spread operator
  - can add all values from an 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*);
- destructuring to an array
  - let [a, b, c] = *someIterable*; // gets first three values
- Map constructor takes an iterable over key/value pairs
- Set constructor takes an iterable over elements
- Promise methods all and race take an iterable over promises
- In a generator, yield* yields all values in an iterable one at a time

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);
}
```

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

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 getFilterIterator(arr, filter) {
  let index = 0;
  return {
    [Symbol.iterator] = () => ({
      next() {
        while (true) {
          if (index >= arr.length) return {done: true};
          let value = arr[index++];
          if (filter(value)) return {value};
        }
      }
    })
  };
}

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

Generators

- **Generator functions**
  - implicitly return a **generator** which is a special kind of **iterator**
  - have multiple return points, each specified using **yield** keyword
  - each **yield** is hit in a separate call to the iterator **next** method
- Can obtain values from a sequence one at a time
  - supports lazy evaluation and 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
    - can exit using **return** keyword; value returned is not yielded
- Can create generator methods in class definitions
  - preceded method name with *
  - ex. to make instances iterable, * **[Symbol.iterator]() { … }**

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 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

Generator yield

- To return a “normal” value
  - yield value;

```
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);
}
```

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

- 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);
  - not commonly used

```
// 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);
}
```

More Generator Examples

```
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
```

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

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

Code.Hub

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!'));
}
```

Generators For Async

```
function async(generatorFn) {
  let iter = generatorFn();
  function success(result) {
    let next = iter.next(result);
    // next.value is a promise
    // next.done will be true if 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) {
    let 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();
}
```

Generators for Async

```
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);
  }
});
```

Proxies

- Can intercept all operations whose names match functions on the Reflect object
  - 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 in modern browsers - 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;
  }
});
```

Proxies

```
// 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());
```

Proxies

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 function
  - 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

```
"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);
}
```

```
// 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

What's Next?

- The next version is always referred to as "JS-next"
- Next ES6 is ES7
- Will include
  - async and await keywords
  - type annotations
  - new Object instance method observe
  - array comprehensions
  - generator comprehensions
  - 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
- Works in Traceur and Babel now in experimental mode

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!'));
}
```

```
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();
```

Code.Hub

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 = 'Rick Mark Folk';
console.log('initials are', initials(name)); // RMF
console.log('full name?', isFullName(name)); //
```

```
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));
```

Cheatsheet

Scopes

- Block Scope (a.k.a lexical scope):
    - Anything within **{ }** that isn't a function
    - Examples: **{ }**, **if { }**, **else** { }, **for () { }**, **while () { }**
    - **let** and **const** respect block scope, while **var** does not
- Function Scope:
    - Anything within a function
    - Example: **function func() { }**

Variables

- **let** and **const** respect block scope, while **var** only respects function scope
- **const** values are constant, cannot be reassigned, and must be initialized on declaration
- **const** can't be reassigned, but the contents of complex **const** values, such as arrays and objects, can be modified

Cheatsheet

String Helpers

- **includes()** returns **true** or **false** depending on if the given text is found anywhere within the string
- **startsWith()** returns **true** or **false** depending on if the given text is found at the beginning of the string
- **endsWith()** returns **true** or **false** depending on if the given text is found at the end of the string

Template Strings

const message = `User ${name} scored ${score} on the exam`;

Use backticks (``) around the string and **${}** around variables or expressions

Cheatsheet

Arrow Functions

const getMessage = name => `Hello ${name}!`;

To make an arrow function:

- Remove the word function
- Place a fat arrow (=>) after parameters
- If there's only one parameter, remove the surrounding parentheses **()**
- If there's only one expression in the function body, remove **{ }**, **return**, and the semicolon (;) after the object you're returning.

Features of arrow functions:

- In arrow functions, the value of **this** is lexically fetched from the scope the arrow function sits inside

Cheatsheet

Array Helpers

Each array helper has an iterator function, which has the parameters of **element**, **index**, and the **array** itself (**reduce()**'s iterator function has an extra first parameter, **initialValue**). Only use the parameters you need.

Array Helper	Use it to...	Returns
arr.forEach((element, index, array) => { });	loop through elements in an array	**undefined**
arr.every((element, index, array) => { });	check if every element passes a test	**true||false**
arr.some((element, index, array) => { });	check if some (at least 1) **arr** elements pass a test	**true||false**
arr.filter((element, index, array) => { });	create a new array by filtering the original array elements	**Array**
arr.map((element, index, array) => { });	create a new array by modifying the original array elements	**Array**
arr.find((element, index, array) => { });	find the first element that passes a test	1 element
arr.findIndex((element, index, array) => { });	find the index of the first element that passes a test	1 element
arr.reduce((calculatedValue, element, index, array) => { }, initialValue);	pass in an **initialValue** and modify it according to the current **element** value	1 reduced value

Cheatsheet

Methods for Altering Arrays

Array Method	Use it to...	Returns
arr.fill((value, startIndex, endIndex) => { });	fill all the elements of an array from start index to end index with **value**	the modified array
arr.copyWithin((targetIndex, startIndex, endIndex) => { });	copy part of an array to another location in the same array	the modified array

Cheatsheet

Function Parameters

- Default Function Parameter Values

  Use an equal sign (=) after the parameter to which you'd like to assign a default value.

```
function request(url, timeout = 2000, callback = () => {}) {
    callback();
}
```

- Enforcing Mandatory Parameters

```
function mandatory() {
  throw new Error('You forgot a parameter!');
}

function doSomething(mustBeProvided = mandatory()) {
  return mustBeProvided;
}
```

Cheatsheet

- Rest Parameters

  Rest parameters compress the rest (i.e. all following parameters) of the parameters following ... in a function to an array, for example:

```
function giveItARest(param1, ...allTheRest) {
  console.log(param1); // 1
  console.log(allTheRest); // [2, 3, 4, 5, 6]
}

giveItARest(1, 2, 3, 4, 5, 6);
```

- Spread Operator

  Place ... before an array to spread the array into individual elements.

```
const numbers = [2, 3, 4];
console.log([1, numbers]); // nested array: [1, [2, 3, 4]]
console.log([1, ...numbers]); // normal array: [1, 2, 3, 4]
```

Cheatsheet

Object Syntax

- Concise Methods

  Object methods no longer need : function, for example:

```
const dog = {
  name: 'Boo',
  bark() {
    console.log('yip!');
  }
};
dog.bark(); // yip!
```

Cheatsheet

Destructuring

- Objects

```
const person = {
  name: 'Whitney',
  age: 38
};

const { name, age } = person;
console.log(name, age); // Whitney 38
```

In the event a value is undefined, set a default value with an equal sign (=) after that value, for example:

```
const person = {
  name: 'Whitney'
};

const { name, age = 0 } = person;
console.log(name, age); // Whitney 0
```

Cheatsheet

Destructuring can also be used in function parameters when an object (e.g. an options object) is passed in:

```
function register( { email, username, password, dateOfBirth, city }) {
  // code to create a new user
}

register(user); // will be destructured within register() parameters
```

To assign a different variable name to the destructured value, use a colon after the key name, for example:

```
const { name : localName, age } = person;
// now I can reference localName going forward!
```

Cheatsheet

Destructuring

- Arrays

  Use destructuring on arrays to pluck values by position, for example:

```
const colors = ['red', 'green', 'blue'];
const [ first, second, third ] = colors;
console.log(first, second, third); // red green blue
```

  In the event a value is undefined, set a default value with an equal sign (=) after that value, for example:

```
const scores = [ 94, 89 ];
const [ first, second, third = 0 ] = scores;
console.log(first, second, third); // 94 89 0
```

Cheatsheet

Promises

- Promises are placeholders for results that will eventually process. They're a great alternative to 'callback hell' and give developers more control over determining when an event is resolved or rejected. Be sure to always include a **catch()** handler.

- To create a promise, create a **Promise** object with an executor callback function that accepts **resolve** and **reject** handlers. Attach a **catch()** block as a rejection handler and a **then()** block as your fulfillment handler, for example:

Cheatsheet

```
const promise = new Promise((resolve, reject) => {
        if (true) {
                resolve(12);
        } else {
                reject(13);
        }
});

promise.then(res => {
                console.log(res); // 12
        })
        .catch(err => {
                console.log(err); // would output 13 if there were an error
        });
```

Promise.all() accepts an iterable (e.g. an array) of promises, and returns an array of result values. If any of the promises in the array fail, the returned promise fails too.

Promise.race() accepts an iterable (e.g. an array) and returns the results of the

Cheatsheet

Fetch

The Fetch helper accepts a URL and returns a promise. Use **.json()** on the initially-returned **Response** promise to get the **JSON** you need via chaining **this()**, for example:

```
fetch('https://api.github.com/users/octocat/repos')
  .then(response => response.json())
  .then(data => console.log(data)) // the JSON data
  .catch(err => console.log(err));
```

Cheatsheet

Generator Functions

Generator functions are pausable functions that, when called, return an *iterator*. They contain *yield* statements. After a *yield* statement is returned, the function pauses execution at that point. To construct a generator function, use a * between **function** and the function name.

Call **next()** on the iterator to return an object containing the yielded **value** and **done** state of the generator function (e.g. **{ value: yieldValue, done: trueOrFalse}**).

```
function *createIterator() {
    yield 30;
    yield 31;
    yield 32;
}
const iterator = createIterator();
console.log(iterator.next()); // { value: 30, done: false }
console.log(iterator.next()); // { value: 31, done: false }
console.log(iterator.next()); // { value: 32, done: false }
// subsequent calls
console.log(iterator.next()); // { value: undefined, done: true }
```

Code.Hub

Cheatsheet

Iterables
To make an object you've created iterable, add a ***[Symbol.iterator]()** method with yielded values:

```
const team = {
  members: [],
  *[Symbol.iterator](){
    for (let member of this.members) {
      yield member;
    }
  }
};
```

Sets
A set is an ordered list of **unique** values.

```
const set = new Set([1, 2, 3, 4, 5, 5, 5, 5]);
console.log(set.size); // 5
set.add(6);
console.log(set.size); // 6
```

Cheatsheet

Maps
A map is an ordered list of key-value pairs, where the key and value can be any type.

```
const map = new Map();
map.set('title', 'JS206: Intro to ES6');
map.set('group', 'GDI');

console.log(map.get('title')); // JS206: Intro to ES6
console.log(map.get('group')); // GDI
```

Built-in Iterators
You can loop over collections, including arrays, objects (using **Object.theIterator(objectName)**), maps, and sets using the following functions:

- **keys()** returns an iterator whose values are the keys contained in the collection
- **entries()** returns an iterator whose values are key-value pairs
- **values()** returns an iterator whose values are the values of the collection

Cheatsheet

Classes

Declare **classes** with the class keyword, and make derived classes with the **extends** keyword. A derived class can access the base class constructor by calling **super()**:

```
class Rectangle {
  constructor(length, width) {
    this.length = length;
    this.width = width;
  }
  getArea() {
    return this.length * this.width;
  }
}
class Square extends Rectangle {
  constructor(length) {
    super(length, length); // calls the base class constructor
  }
}
const square = new Square(5);
console.log(square.getArea()); // 25
```

Cheatsheet

Exporting and Importing Modules

Use the **export** keyword before variables, functions, and classes that you'd like to expose in a file, for example:

```
// export variables
export let description = 'GDI Fan';
export const threshold = 7;
// export functions
export function multiply(a, b) {
  return a * b;
}
// export classes
export class Rectangle {
  constructor(length, width) {
    this.length = length;
    this.width = width;
  }
}
```

Cheatsheet

Import variables, functions, and classes by using the **import** keyword and placing all of your imports at the top of your file, for example:

```
import { Rectangle, description, multiply } from './myExports.js';

const shape = new Rectangle(6, 8);
console.log(description); // 'GDI Fan'
console.log(multiply(3, 5)); // 15
```

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
- enhanced object literals
- for-of loops
- iterators
- generators
- modules
- promises
- rest parameters
- spread operator
- template strings
- new methods in String and Object classes