ECMAScript (ES) 2015

a.k.a. ES6

Mark Volkmann
Object Computing, Inc.

slides are at ociweb.com/mark;
search for "MidwestJS"

ECMAScript

- Specification for the JavaScript language
- Defined by ECMA technical committee TC39
- Many ES 2015 features provide **syntactic sugar** for more concise code

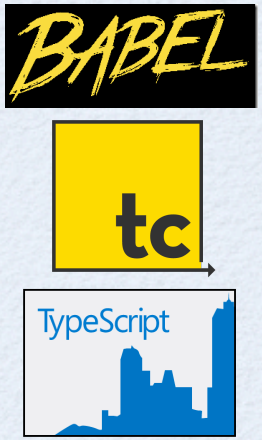
- One goal of ES 2015 and beyond is to make JavaScript a **better target for compiling to from other languages**

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

ES 2015 to ES5 Transpilers

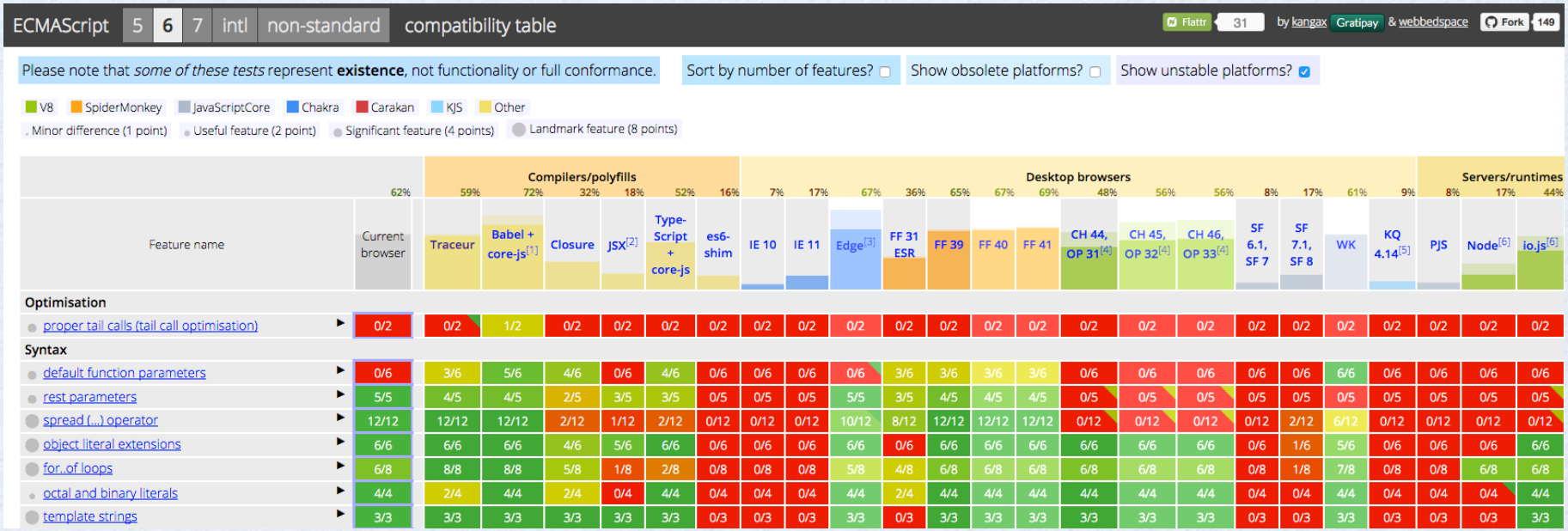
percentages are as of 8/6/15

- The most popular are listed here
- All of these can be installed using **npm**, can be run from **gulp** and **Grunt**, and support **sourcemaps**
- **Babel - 72%**
  - https://babeljs.io
- **Traceur - 59%**
  - from Google; https://github.com/google/traceur-compiler/
- **TypeScript - 52%**
  - from Microsoft; http://www.typescriptlang.org
  - "a typed superset of JavaScript that compiles to plain JavaScript"
  - 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!



Use ES 2015 Today?

- 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/



ECMAScript 5 6 7 intl non-standard compatibility table — Flattr 31 — by kangax Gratipay & webbedspace — Fork 149

Please note that *some of these tests* represent **existence**, not functionality or full conformance. — Sort by number of features? — Show obsolete platforms? — Show unstable platforms?

V8 SpiderMonkey JavaScriptCore Chakra Carakan KJS Other

Minor difference (1 point) Useful feature (2 point) Significant feature (4 points) Landmark feature (8 points)

Feature name	Current browser 62%	Compilers/polyfills: Traceur 59%	Babel + core-js[1] 72%	Closure 32%	JSX[2] 18%	TypeScript + core-js 52%	es6-shim 16%	Desktop browsers: IE 10 7%	IE 11 17%	Edge[3] 67%	FF 31 ESR 36%	FF 39 65%	FF 40 67%	FF 41 69%	CH 44, OP 31[4] 48%	CH 45, OP 32[4] 56%	CH 46, OP 33[4] 56%	SF 6.1, SF 7 8%	SF 7.1, SF 8 17%	WK 61%	KQ 4.14[5] 9%	Servers/runtimes: PJS 8%	Node[6] 17%	io.js[6] 44%
Optimisation																								
proper tail calls (tail call optimisation)	0/2	0/2	1/2	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	0/2
Syntax																								
default function parameters	0/6	3/6	5/6	4/6	0/6	4/6	0/6	0/6	0/6	0/6	3/6	3/6	3/6	3/6	0/6	0/6	0/6	0/6	0/6	6/6	0/6	0/6	0/6	0/6
rest parameters	5/5	4/5	4/5	2/5	3/5	3/5	0/5	0/5	0/5	5/5	3/5	4/5	4/5	4/5	0/5	0/5	0/5	0/5	0/5	0/5	0/5	0/5	0/5	0/5
spread (...) operator	12/12	12/12	12/12	2/12	1/12	2/12	0/12	0/12	0/12	10/12	8/12	12/12	12/12	12/12	0/12	0/12	0/12	0/12	2/12	6/12	0/12	0/12	0/12	0/12
object literal extensions	6/6	6/6	6/6	4/6	5/6	6/6	0/6	0/6	0/6	6/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	0/6	1/6	5/6	0/6	0/6	0/6	6/6
for..of loops	6/8	8/8	8/8	5/8	1/8	2/8	0/8	0/8	0/8	5/8	4/8	6/8	6/8	6/8	6/8	6/8	6/8	0/8	1/8	7/8	0/8	0/8	6/8	6/8
octal and binary literals	4/4	2/4	4/4	2/4	0/4	4/4	0/4	0/4	0/4	4/4	2/4	4/4	4/4	4/4	4/4	4/4	4/4	0/4	0/4	4/4	0/4	0/4	0/4	4/4
template strings	3/3	3/3	3/3	3/3	3/3	3/3	0/3	0/3	0/3	3/3	0/3	3/3	3/3	3/3	3/3	3/3	3/3	0/3	0/3	3/3	0/3	0/3	0/3	3/3

ES 2015 Features

- The following slides describe most of the features in ES 2015

Block Scope

- `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)
- `const` declares **constants** with block scope
  - must be initialized
  - reference can't be modified, but object values can
- Function and class definitions are block scoped
- Use a `{}` block in place of an IIFE

```
function demo() {
  console.log(name); // error
  console.log(age); // error
  const name = 'Mark';
  let age = 53;
  age++; // okay
  name = 'Richard'; // error

  if (age >= 18) {
    let favoriteDrink = 'daquiri';
    ...
  }
  console.log(favoriteDrink); // error
}
```

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

```
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
```

- 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

Symbols ...

- Immutable identifiers that are guaranteed to be unique
  - unlike strings
- To create a "local" symbol
  - `let sym = Symbol(description);`
  - `new` keyword is not used
  - description is optional and mainly useful for debugging
- To retrieve description
  - `sym.toString()`
  - returns `'Symbol(description)'`
- 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
  - but can retrieve them with `Object.getOwnPropertySymbols(obj)`, so not private
- 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)
- Well Known Symbols
  - used as method names in custom classes to override how instances are processed by certain operators and built-in class methods
  - see `Symbol.hasInstance`, `Symbol.isConcatSpreadable`, `Symbol.iterator`, `Symbol.match`, `Symbol.replace`, `Symbol.search`, `Symbol.split`, `Symbol.species`, `Symbol.toPrimitive`, `Symbol.toStringTag`, and `Symbol.unscopables`

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

`Math`/`Number`/`String` Additions

- New functions on `Math`
  - `fround`, `sign`, `trunc`, `cbrt`, `expm1`, `hypot`, `imul`, `log1p`, `log10`, `log2`, `asinh`, `acosh`, `atanh`
- New functions on `Number`
  - `isFinite`, `isNumber`, `isNaN`, `isSafeInteger`, `toInteger`, `parseInt`, `parseFloat`

can be represented in 53 bits of a double

- New syntax for hexadecimal, octal, and binary literals
  - `0xa === 10, 0o71 === 57, 0b1101 === 13`
- New functions and methods on `String`
  - methods: `endsWith`, `startsWith`, `includes`, `repeat`
  - handling UTF-16 characters (2 or 4 bytes) - `codePointAt` method, `fromCodePoint` function

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']`

... 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
```

`Array` Additions

- New functions
  - `of`, `from`
- New methods
  - `copyWithin`, `find`(*`predicate`*), `findIndex`(*`predicate`*), `fill`
  - `entries` - returns an iterator over `[`*`index`*`, `*`value`*`]` pairs of *`arr`*
  - `keys` - returns an iterator over indices of *`arr`*
  - `values` - returns an iterator over values in *`arr`*

returns first matching element or index

same API as in `Set` and `Map`

`Object` Additions

- New functions
  - `assign` (see next slide), `is`, `setPrototypeOf`, `getOwnPropertySymbols`

Object.assign

- **`Object.assign(target, src1, ... srcN)`**
  - copies properties from src objects to target (left to right), replacing those already present
  - returns ***`target`***
  - can create 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!

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)

```
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`

New Collection Classes

- **`Set`**
  - instances hold collections of unique values
    - when values are objects, they are compared by reference
  - values can be any type including objects and arrays
- **`Map`**
  - 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
- **`WeakSet`** - similar API to **`Set`**, but
  - 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`** - similar API to **`Map`**, but
  - keys must be objects
  - keys are "weakly held", i.e. a pair can be garbage collected if 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

`Set` Class

- 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()`

- `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

`Map` Class

- 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)`

chain to add/modify multiple values

- 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()`

- `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

Promises ...

- Proxy for a value that may be known in the future after an asynchronous operation completes
  - such as a REST call
- 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
    - `.then(cb1, cb2)` is similar to `.then(cb1).catch(cb2)`, but differs in that `cb2` won't be invoked if `cb1` throws
  - `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
    - ES 2016 will likely add `finally` method
- 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
  - "resolved" state is sometimes called "fullfilled"
  - once state is resolved or rejected, can't return to pending

… Promises …

create with `Promise` constructor, passing it
a function that takes `resolve` and `reject` functions,
and calls one of them

```
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.

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
- 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

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

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
- **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"

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

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`***

    sfx is short for "self executing"
  - 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

the basics plus a little jQuery

```
jspm install jquery
```

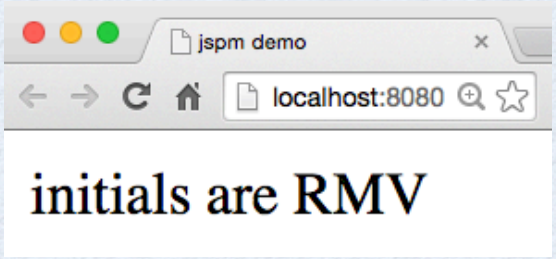
index.html

```
<!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('');
}
```

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
  - `Array`, `Set`, and `Map` classes do this

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`, `Set`, and `Map` return an iterator

  objects returned are both iterators and 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

- **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);
}
```

can get an iterator over keys of 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 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);`
- 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

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 55

stops iterating when `done` is true which never happens in this example

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

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

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 58
  - 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 58
  - 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}`**

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

```
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 48

```
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);
}
```

also see **`yield*`** to yield each value returned by an iterable one at a time; can use to make recursive calls to the same or a different generator function

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 59

... 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!

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
- 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 55-56

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

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 Babel/Traceur and JSHint/ESLint
- 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