Catching Up on ES6
(ES 2015)

OCI | WE ARE SOFTWARE ENGINEERS.

Overview

- ES6, the latest version of JavaScript, added many great features
- This talk explains many of them including
  - block scope
  - default parameters
  - rest and spread operators
  - destructuring
  - arrow functions
  - enhanced object literals
  - class syntax
  - template literals
- Most of these features are just syntactic sugar for things that could already be done in ES5
  - marked with sugar packet in upper-right

Block Scope

- **`let`** declares variables like **`var`**, but they have block scope
  - 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
    - to prevent changes to object values, use `Object.freeze(obj)`
- For both
  - not hoisted to beginning of enclosing block, so references before declaration are errors
  - when a file defines a module, top-level uses of **`let`** and **`const`** are file-scoped, unlike **`var`**
  - when a **`let`** or **`const`** variable is accessed out of its scope, a **`ReferenceError`** is thrown with message "***`name` `is not defined`***"

```
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"
  so they are treated as separate arguments to a function
  or elements in a literal array

examples of things that are iterable include arrays and strings

- 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]

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

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

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

arr1 = [...arr1, 'qux', 'foo'];
```

alternative to
`arr1.push('qux', 'foo');`

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
- Can't start statement with `{`, so when assigning to existing variables using object destructuring, surround with parens

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

... Destructuring ...

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

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

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

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

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

... Destructuring ...


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

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

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

extracting array elements by position

extracting object property values by name

creates `name` variable, but not `pitcher`

… Destructuring

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

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

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

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

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

order is irrelevant

Arrow Functions ...

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

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

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

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

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

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

… Arrow Functions

- Inside arrow function, **`this`** has same value as containing scope, not a new value (called "lexical this")
  - so can't use to define constructor functions or prototype methods, only plain functions
- Also provides "lexical super" for use in class constructors and methods
  - can use **`super`** keyword to invoke a superclass method

Enhanced Object Literals ...

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

```
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
```

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!`;
```

- Also see "tagged template literals"

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
  - `typeof Shoe === 'function'`

```
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

The End

- Thanks so much for attending my talk!
- Feel free to find me later and ask questions about anything in the JavaScript world
- **Contact me**

Mark Volkmann, Object Computing, Inc.

Email: mark@ociweb.com

Twitter: @mark_volkmann

GitHub: mvolkmann

Website: http://ociweb.com/mark