ECMAScript 6 Review

Sperasoft
Your game dev partner

JavaScript and ECMAScript 6

- Name “JavaScript” is a licensed trademark by Sun Microsystems
- JavaScript is described in ECMA-262 specification by the name “ECMAScript”
- Current ECMAScript version is 5.1
- ECMAScript 6 will be released in December 2014

Introduction

JavaScript and ECMAScript 6

- Name “JavaScript” is a licensed trademark by Sun Microsystems
- JavaScript is described in ECMA-262 specification by the name “ECMAScript”
- Current ECMAScript version is 5.1
- ECMAScript 6 will be released in December 2014

JavaScript and ECMAScript 6

```
let fibonacci = {    max: 1000,    *[Symbol.iterator]() {        let pre =
0, cur = 1;        do {            [pre, cur] = [cur, pre + cur];
yield cur;        } while (cur < this.max);    }
}for (let n of fibonacci) {    console.log(n);}
```

New syntax constructions

We are talking about:

- new declarators (let & const)
- arrow functions
- parameters in functions (default + rest + spread)
- object literals
- destructing assignment
- comprehensions
- for..of loop

New syntax constructions

Let declarations

```
/*
* ECMAScript 6: variables declared with a let statement are created as
* bindings on the lexical environment.
* Each block has its own lexical environment.
*/
function foo(param) {
     if (param) {
               let bar = 5; // block scope declaration
     }
     console.log(bar); // ReferenceError: bar is not defined
}

/**
* without closure
*/
for (var i = 0; i < 10; i++) {
     let j = i; // reassign counter with let

     setTimeout(function () {
          console.log(i, j);
     }, 300 * j); // 10 0 10 1 10 2 10 3 .. 10 9
}
```

New syntax constructions

const declarations

```
/*
* const has the same block scoped binding semantics as let,
* but its value is a read-only constant
*/

const z; // SyntaxError: const declarations must have an
initializerconst y = 10; // y === 10y = 20; // SyntaxError:
Assignment to constant variable
```

New syntax constructions

Arrow functions

```
let simple = a => a > 15 ? 15 : a;simple(16); //
15simple(10); // 10let complex = (a, b) => {    if (a > b)
{        return a;    } else {        return b;    }}
```

Arrow functions

- **Not newable -** *cannot be used a constructors.*
- **No arguments object -** *must use named arguments or rest arguments.*
- **Lexical this binding -** *The value of this inside of the function is determined by where the arrow function is defined.*
- **Can’t change this -** *The value of this inside of the function can’t be changed.*

New syntax constructions

Arrow functions (more samples)

```
function Car() {
    this.speed = 0;

    //use an arrow function
    setInterval(() => {
        this.speed += 5; //this is from Car
        console.log(this.speed);
    }, 1000);
}

let car = new Car(); //5 10 15..
```

New syntax constructions

Parameters in function (default arguments)

```
/**
 * ECMAScript 6 makes easier to provide default values for
 * parameters by providing initializations that are used
 * when the parameter isn’t formally passed.
 */

function multiply(x, y = 1) {
    return x * y;
}

(function example(x, y = x * 2) {
    console.log(x, y); // 2, 4
}(2));
```

New syntax constructions

Parameters in function (rest)

```
/**
* Rest parameters are indicated by three dots (...)
* preceding a named parameter. That named parameter
* then becomes an Array containing the rest of the
parameters
*
* Note: no other named arguments can follow in the function
* declaration after rest parameter
*/
function logEach(...things) {
    things.forEach(function (thing) {
            console.log(thing);
    });
}
logEach("a", "b", "c"); //a b c
```

Parameters in function (spread)

```
/**
 * Instead of calling apply(), you can pass in the
 * array and prefix it with the same ... pattern that
 * is used with rest parameters.
 * The JavaScript engine then splits up the array into
 * individual arguments
 */
let example = (a, b, c) => {console.log(a, b, c)};
let arg = 1;
let args = [2, 3];
example(arg, ...args);// 1 2 3

//used with array literals
let parts = ["shoulder", "knees"];
let lyrics = ["head", ...parts, "and", "toes"];
```

Object literals (property & method Initializer)

```
// ECMAScript 5
{
        name: name,
        sayName: function() {
                console.log(this.name);
        }
};

// ECMAScript 6
{
        name,
        sayName() {
                console.log(this.name);
        }
};
```

Object literals (computed property names)

```
/**
 * The square brackets inside of the object literal
 * indicate that the property name is computed, so
 * its contents are evaluated as a string.
 * That means you can also include expressions
 */
let lastName = "last name";
let suffix = " name";

let person = {
    ["first" + suffix]: "Nicholas",
    [lastName]: "Zakas"
};

console.log(person["first name"]);      // "Nicholas"
console.log(person[lastName]);          // "Zakas"
```

Destructing assignment (objects)

```
let options = {
     repeat: true,
     save: false,
     rules: {
               custom: 10
     }
};

let { repeat, save, rules: { custom }} = options;

console.log(repeat);            // true
console.log(save);              // false
console.log(custom);            // 10

// syntax error without let, var, const
{ repeat, save, rules: { custom }} = options;

// works fine
({ repeat, save, rules: { custom }}) = options;
```

Destructing assignment (array)

```
let colors = [ "red", [ "green", "lightgreen" ], "blue" ];

let [ firstColor, [ secondColor ] ] = colors;

console.log(firstColor);          // "red"
console.log(secondColor);         // "green"

// mixed
let options = {
     repeat: true,
     save: false,
     colors: [ "red", "green", "blue" ]
};

let { repeat, save, colors: [ firstColor, secondColor ]} = options;

console.log(repeat);              // true
console.log(save);                // false
console.log(firstColor);          // "red"
console.log(secondColor);         // "green"
```

New syntax constructions

for...of loop

```
let arr = [ 3, 5, 7 ];arr.foo = "hello";// ECMAScript 5for (var i in arr) {
console.log(i); // logs "0", "1", "2", "foo"}// ECMAScript 6for (let i of
arr) {   console.log(i); // logs "3", "5", "7"}

/**
 * for...of uses iterators for iteration, thus it doesn't * iterate through
regular object
 */
```

New features

- Iterators
- Symbol
- Map
- WeakMap
- Set
- Generators

New features

Iterators

```
var iterator = function () {
  var base = 2, count = 3, current = 1;
  return {
          next: function () {
                   return {
                            done: (count--) <= 0,
                            value: current *= base
                   }
          }
  };
};
var i = iterator();
i.next(); // { done: false, value: 2 }
i.next(); // { done: false, value: 4 }
i.next(); // { done: false, value: 8 }
i.next(); // { done: true, value: 16 }
```

Iterators (ECMAScript 5)

```
collection.iterator = function () {    var that = this, currentIndex = 0;
return {        next: function () {            return {
value: that.models[currentIndex++],                  done: currentIndex >
that.models.length
            };         }    }
}; for (var iterator = collection.iterator(), next = iterator.next();
!next.done; next = iterator.next()) {    console.log(next.value);}
```

Have a question?
Like this deck?

Just follow us on twitter
@Sperasoft

Iterators (ECMAScript 6)

```
collection[Symbol.iterator] = function () {    var that = this, currentIndex
= 0;    return {        next: function () {            return {
value: that.models[currentIndex++],                done: currentIndex >
that.models.length
            };        }    }
};for (let model of collection) {    console.log(model);}
```

New features

Generators

```
collection[Symbol.iterator] = function* () {     for (let modelKey in
this.models) {         yield this.models[modelKey];     }
}; for (let model of collection) {     console.log(model);}
```

New features

Generators

```
me.hunt = function* (dragons) {     let fails = 0;     for (let dragon of
dragons) {         let result = dragon.hunt();         yield [dragon, result];
        if (!result) { fails++; }         if (fails >= 3) { return; }     }
};let dragons = [                  ,              ,          ,
];for (let [dragon, result] of me.hunt(dragons)) {     console.log("Hunt on "
+ dragon + " was " + (result ? "successful!" : "failed."));}
```

Map

```
let map = new Map();map.set('key', 'Primitive string key');map.set(NaN,
'Watman');map.get('key'); // 'Primitive string key'map.get(Number('foo'));
// 'Watman'let a1 = [], a2 = [], a3 = function () { };map.set(a1,
'array');map.set(a2, 'yet another array');map.set(a3, 'not an
array');map.get([]); // undefinedmap.get(a1); // 'array'map.get(a3); // 'not
an array'
```

Map (continue)

```
let map = new Map([['key1', 'value1'], ['key2',
'value2']]);console.log([...map]); // [['key1', 'value1'], ['key2',
'value2']]
console.log([...map.keys()]); // ['key1', 'key2']
console.log([...map.values()]); // ['value1', 'value2']for (let [key, value]
of map) {    console.log('map[' + key + '] = ' + value);
}for (let key of map.keys()) {    console.log('map[' + key + '] = ' +
map.get(key)); // The same as above}
```

New features

WeakMap

```
/**
 * WeakMap is a version of Map with improved memory leak control.
 * It doesn't support primitive keys or enumerators.
 */let map = new WeakMap();map.set('key', 'Primitive string key'); //
TypeError - WeakMap doesn't support primitive keysmap.set(NaN, 'Watman'); //
TypeErrorlet div  = document.createElement('div');map.set(div, 'dom
element');map.get(div); // 'dom element'for (let key of map); // TypeError -
WeakMap doesn't support enumeration
```

New features

Set

```
let set = new Set([1, 1, 2, 3, 5]);set.has(1); //
trueset.delete(1);set.has(1); // false
set.has(8); // falseset.add(8);

set.add(document.querySelector('body'));

let a1 = [], a2 = [];set.add(a1);set.add(a2);set.has(a1); //
trueset.delete(a1);set.has(a2); // true
```

Private properties (without Symbol)

```
let Publisher = (function () {    function Publisher () {
this._callbacks = new Set();    };    Publisher.prototype.subscribe =
function (callback) {        this._callbacks.add(callback);    };
Publisher.prototype.publish = function (data) {        for (let callback of
this._callbacks)            callback(data);    };    return Publisher;
})();
```

New features

Private properties (with Symbol)

```
let Publisher = (function () {    let callbacks = Symbol('callbacks');
function Publisher () {        this[callbacks] = new Set();    };
Publisher.prototype.subscribe = function (callback) {
this[callbacks].add(callback);    };    Publisher.prototype.publish =
function (data) {        for (let callback of this[callbacks])
callback(data);    };    return Publisher;
})();
```

Symbols

```
let callbacks = Symbol('callbacks');Symbol('callbacks') ===
Symbol('callbacks'); // falseString('callbacks') === String('callbacks'); //
truecallbacks.toString(); // Symbol(callbacks)typeof callbacks; // symbol
```

Well-known symbols:

- Symbol.create
- Symbol.iterator
- Symbol.toStringTag
- etc.

Even more new features

Long story short:

- Proxy
- Classes
- Modules
- Template strings
- Array comprehension
- New methods (Object.assign, Array.from, String, Math.*)

Even more new features

Proxy

```
getOwnPropertyDescriptor   // Object.getOwnPropertyDescriptor(proxy,name)
getOwnPropertyNames        // Object.getOwnPropertyNames(proxy)
getPrototypeOf                 // Object.getPrototypeOf(proxy)
defineProperty                 //Object.defineProperty(proxy,name,desc)
deleteProperty                 // delete proxy[name]
freeze                         // Object.freeze(proxy)
seal                           // Object.seal(proxy)
preventExtensions              // Object.preventExtensions(proxy)
isFrozen                       // Object.isFrozen(proxy)
isSealed                       // Object.isSealed(proxy)
isExtensible                   // Object.isExtensible(proxy)
has                            // name in proxy
hasOwn                         // ({}).hasOwnProperty.call(proxy,name)
get                            // receiver[name]
set                            // receiver[name] = val
enumerate                      // for (name in proxy)
keys                           // Object.keys(proxy)
apply                          // proxy(...args)
construct                      // new proxy(...args)
```

Even more new features

Proxy

```
let p = Proxy.create({
         get: function (proxy, name) {
                  return (name in this) ? this[name] : 'Unknown property '
+ name;
         },
         set: function(proxy, name, value) {
         if (name === 'age') {
                  if (!Number.isInteger(value)) {
                  throw new TypeError('The age is not an integer');
                  }
                  if (value > 150) {
                  throw new RangeError('The age seems invalid');
                  }
         }
         //provide default behaviour
         this[name] = value;
         }
});
p.age = 100;
p.age = 300; //RangeError: The age seems invalid
console.log(p.age); //100
console.log(p.height); //Unknown property height
```

Even more new features

Classes

```
class Polygon {
   constructor(height, width) { //class constructor
          this.name = 'Polygon';
          this.height = height;
          this.width = width;
   }
   sayName() { //class method
          console.log('Hi, I am a', this.name + '.');
   }
}
class Square extends Polygon {
   constructor(length) {
          super(length, length); //call the parent method with super
          this.name = 'Square';
   }
   get area() { //calculated attribute getter
          return this.height * this.width;
   }
}
let s = new Square(5);
s.sayName(); //Hi, I am a Square.
console.log(s.area); //25
```

Even more new features

Modules

Modules has two main advantages:

- ***You could get compile time errors*** *if you try to import something that has not been exported.*
- ***You can easily load ES6 modules asynchronously.***

The ES6 module standard has two parts:

- ***Declarative syntax*** *(for importing and exporting).*
- ***Programmatic loader API:*** *to configure how modules are loaded and to conditionally load modules.*

Even more new features

Modules

```
// point.js
module "point" {
export class Point {
    constructor (x, y) {
            public x = x;
            public y = y;
            }
    }
}

// myapp.js
module point from "/point.js";
import Point from "point";

var origin = new Point(0, 0);
```

Even more new features

Template strings

```
let name = "John", surname = "Doe";
let template1 = `Hello! My name is ${name} ${surname}!`;
console.log(template1); // Hello! My name is John Doe!

let salutation = 'Hello';
let greeting = `
    ${salutation},
       this
           crazy
               world!`;
console.log(greeting);
/*
Hello,
       this
               crazy
                      world!
*/
```

Even more new features

Array comprehension

```
//ECMAScript 5
[1, 2, 3].map(function (i) { return i * i }); // [1, 4, 9]

[1,4,2,3,-8].filter(function(i) { return i < 3 }); // [1, 2, -8]

//ECMAScript 6
[for (i of [1, 2, 3]) i * i]; // [1, 4, 9]

[for (i of [1,4,2,3,-8]) if (i < 3) i]; // [1, 2, -8]

// generator
function* range(start, end) {
    while (start < end) {
            yield start;
            start++;
    }
}

var ten_squares = [i * i for each (i in range(0, 10))];
```

Even more new features

New methods (Object.assign, Array.from, String, Math.*)

```
Number.EPSILON
Number.isInteger(Infinity) // false
Number.isNaN("NaN") // false

Math.acosh(3) // 1.762747174039086
Math.hypot(3, 4) // 5
Math.imul(Math.pow(2, 32) - 1, Math.pow(2, 32) - 2) // 2

"abcde".contains("cd") // true
"abc".repeat(3) // "abcabcabc"

Array.from(document.querySelectorAll('*')) // Returns a real Array
Array.of(1, 2, 3) // Similar to new Array(...), but without special one-arg
behavior
[0, 0, 0].fill(7, 1) // [0,7,7]
[1,2,3].findIndex(x => x == 2) // 1
["a", "b", "c"].entries() // iterator [0, "a"], [1,"b"], [2,"c"]
["a", "b", "c"].keys() // iterator 0, 1, 2
["a", "b", "c"].values() // iterator "a", "b", "c"

Object.assign(Point, { origin: new Point(0,0) }) //extend
```

Can you read it now?

Block scoped declarator

JavaScript and ECMAScript 6

Iterator property

```
let fibonacci = {     max: 1000,     *[Symbol.iterator]() {          let pre =
0, cur = 1;          do {               [pre, cur] = [cur, pre + cur];
yield cur;          } while (cur < this.max);     }
}for (let n of fibonacci) {    console.log(n);}
```

Destructing assignment

Generator

Iterate through values

How to try right now?

Browser support

- Almost everything supports consts
- IE11 also supports let, Map, Set and WeakMap
- Chrome supports some functions (Number.isNaN, Number.isInteger, Object.is, etc.)
- Firefox doesn't support classes, template strings, computed properties and Object.assign

How to try right now?

Polyfills

- **harmony-collections** provides implementations of Map, Set and WeakMap (https://github.com/Benvie/harmony-collections)
- **es6-promise** provides implementation of Promise (https://github.com/jakearchibald/es6-promise)
- **es6-shim** provides a lot of stuff (https://github.com/paulmillr/es6-shim/)

How to try right now?

Compilers

- **Google Traceur** (https://github.com/google/traceur-compiler)
- **TypeScript** - supports classes, modules, some syntax stuff (http://www.typescriptlang.org/)

Follow us on Twitter @Sperasoft

Visit our site: sperasoft.com