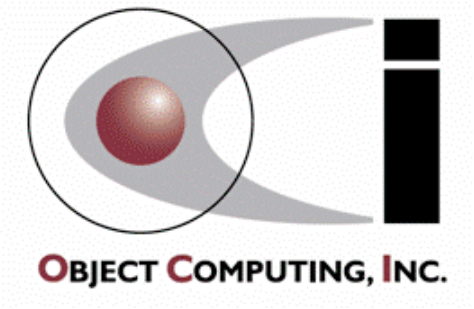
ECMAScript (ES) 5

Mark Volkmann
mark@ociweb.com



History

- **ECMAScript 1** - 6/97
- **ECMAScript 2** - 6/98 - only editorial changes
- **ECMAScript 3** - 12/99
  - regular expressions, string handling improvements
- **ECMAScript 4** - never completed
  - not backward compatible with ECMAScript 3
  - large number of changes and new features
  - very controversial
  - eventually scaled back and renamed ECMAScript 3.1, then became ...
- **ECMAScript 5** - 12/09 - 10 years after last released revision!
  - a.k.a. ECMAScript, 5th Edition
  - compatible with ECMAScript 3
  - adds object properties, "strict mode" subset, JSON support, more reflection, and a few more features
  - spec is at http://www.ecmascript.org/ - see "Fifth Edition of ECMA-262"
- **ECMAScript Harmony** - code name of next edition; work in progress

Object Extensibility

- Refers to the ability to add properties, including functions, to objects

```
obj.p1 = foo;
obj.p2 = function () { ... code ... };
```

- To prevent an object from being extended

```
Object.preventExtensions(obj);
```

  - only works if "strict mode" is enabled

Why not `obj.preventExtensions()`? The rationale is that it would merge the meta and application layers.

- To determine if an object is extensible

```
if (Object.isExtensible(obj)) { ... }
```

- Can't re-enable extensions

Object Properties

- An object "property" has
  - optional getter method, called when value is retrieved
    - can use to compute or lookup value
  - optional setter method, called when value is changed
    - can use to validate value
    - can use to set other related property values
  - "property descriptor" that includes the value and three flags described next
    - these four things are referred to as "property attributes"

Property Descriptor Flags

- **`writable`**
  - if false, the value cannot be changed (a constant)
  - only applies to properties that have a **`value`** attribute and no **`get`** or **`set`** attribute
- **`configurable`**
  - if false, the property cannot be deleted from its object
  - if false, the descriptor flags cannot be changed
    - except **`writable`** can be changed from **`true`** to **`false`**
- **`enumerable`**
  - if false, a **`for`** loop will not see the property when iterating through the properties of its object
- Default values?
  - article by John Resig at says they all default to **`true`**
  - spec says they all default to **`false`** in section 8.6.1, table 7

http://ejohn.org/blog/ecmascript-5-objects-and-properties/

Defining a Property

- To define a property, set its initial value or get/set methods (not both) and set its attribute flags (if non-default value are desired)

```
Object.defineProperty(obj, "temperature", {
  t: 98.2, // descriptor property that holds value
  configurable: false,
  get: function () { return t; },
  set: function (value) { t = value; }
});
```

`t` is not accessible outside get and set methods

`obj.temperature` calls this

`obj.temperature` = calls this

- When **`get`** and **`set`** methods are trivial like above, the following is equivalent

```
Object.defineProperty(obj, "temperature", {
  value: 98.2,
  configurable: false
});
```

Defining Multiple Properties

- To define multiple properties in one call

```
var person = {};
Object.defineProperties(person, {
  "name": {
    value: "Mark Volkmann",
    configurable: false, // can't delete
    writable: false }, // can't change
  "age": {
    value: 49,
    configurable: false // can't delete
    set: function (value) {
      if (value < 0 || value > 110) {
        throw new RangeError("age must be between 0 and 110");
      }
      age = value;
    }
  }});
```

Getting Property Names

- To get names of all enumerable properties of an object

```
var propNames = Object.keys(person); // returns ["name", "age"]

// The following is preferred over the old-style
// for (var prop in obj) {
//   if (obj.hasOwnProperty(prop)) {
//     ...
//   }
// }
Object.keys(person).forEach(function (key) {
  ...
});
```

- To get array of names of all properties of an object, including those that are not enumerable

```
var propNames = Object.getOwnPropertyNames(person);
```

Retrieving a Property Descriptor

- To retrieve the property descriptor of an object property

```
var obj = { p1: "foo", p2: 19 };
var pd = Object.getOwnPropertyDescriptor(obj, "p1");
// pd = {
//   value: "foo",
//   writable: true,
//   enumerable: true,
//   configurable: true
//}
```

It seems Resig is correct, at least in the Node.js implementation.

Sealing

- Prevents property addition, property deletion and descriptor changes for an object
- To seal an object

```
Object.seal(obj);
```

  - sets **`configurable`** property attribute to **`false`** for each property in the object and calls **`Object.preventExtension(obj);`**
  - can still access and modify the existing properties
- To determine if an object is sealed

```
if (Object.isSealed(obj)) { ... }
```

- Cannot unseal an object

Freezing

- Same as sealing, but properties cannot be modified
- To freeze an object

```
Object.freeze(obj);
```

- To determine if an object is frozen

```
if (Object.isFrozen(obj)) { ... }
```

- Cannot unfreeze an object

Object Creation

- To create an object with
  - a specific prototype object
  - set of properties specified in the same way as when defining multiple properties

```
var obj = Object.create(prototypeObject, properties);
```

- To get prototype of an object

```
var prototypeObject = Object.getPrototypeOf(obj);
```

Strict Mode ...

- Helps avoid common coding problems
- Enabled with the directive "`use strict`";
  - opt-in model
  - include the quotes! - single or double
    - planning to drop quotes in a future version
  - just a string, so no new syntax required
  - to affect entire source file, include as first executable statement
    - doesn't affect subsequently parsed files
  - to affect a single function, include as first line in function
  - to affect a set of functions, wrap functions in an anonymous function that includes the directive and executes itself
    - `();` at the end
  - to affect a string of code passed to `eval`, include as first statement in string

... Strict Mode ...

- Has no effect on JavaScript engines that don't support it
  - but code that is tested that way may not run in an engine that does
- See "Annex C" in the spec for a summary of strict mode

... Strict Mode ...

- Variables must be declared before first use
  - either setting or getting
- Object literals cannot contain duplicate property names
- Octal literals are not allowed
  - numbers with a leading zero
- **`with`** statement cannot be used
- **`delete`**
  - can only be used on properties, not variables, functions or parameters
  - cannot be called on properties whose "**`configurable`**" attribute is **`false`**

... Strict Mode ...

- Functions cannot have parameters with duplicate names
- When code executed by **`eval`** declares new variables (with **`var`**) or defines new functions, they exist in a new environment, not in the environment of the caller
- Inside functions (not methods)
  - **`this`** is **`null`** rather than the global object
  - can use to test whether environment supports strict mode

```
var supportsStrict = (function () {
  'use strict';
  return !this;
})();
```

... Strict Mode

- **`arguments`** special variable is immutable
- "**`arguments`**" and "**`eval`**" are reserved
  - cannot be used for the name of a variable, property, function, parameter or **`catch`** identifier
- "**`arguments`**" and "**`caller`**" are reserved
  - cannot create or modify properties with these names on function objects
- **`caller`** property of **`Function`** objects and **`callee`** property of **`Arguments`** objects cannot be accessed

New String Method

- To trim leading and trailing whitespace

```
var s2 = s.trim();
or
s = s.trim();
```

New Date Methods

- Create ISO string from Date
  - example - `'2010-11-04T00:17:15.177Z'`

```
var iso = date.toISOString();
```

- Create Date from ISO string

```
var millis = Date.parse(isoString);
var date = new Date(isoString);
```

milliseconds are since since midnight 01 January, 1970 UTC

- Create Date representing current time

```
var millis = Date.now();
```

New Array Methods ...

- **`isArray(obj)`**

```
if (Array.isArray(obj)) { ... }
```

- **`indexOf(element[, fromIndex])`**

```
var index = arr.indexOf('yellow');
```

- **`lastIndexOf(element[, fromIndex])`**

```
var index = arr.lastIndexOf('yellow');
```

Existing Array methods

concat
join
pop
push
reverse
shift
slice
sort
splice
toString
toLocaleString
unshift

... New Array Methods ...

- **forEach(fn[, thisInFn])**

```
arr.forEach(function (element) { print(element); });
```

  - fn is passed the current element, its index, and the array, but like all JS functions, it only needs to accept those it uses
  - if thisInFn is specified, it is the value of this in fn
    - otherwise this is null

... New Array Methods ...

- **map(fn[, thisInFn])**
  - returns a new Array created from the results of applying fn to each element
  - fn takes same arguments as in forEach

```
var newArr = arr.map(function (element) { return element * 2; });
```

- **filter**
  - returns a new Array containing all the elements for which fn returns true
  - fn takes same arguments as in forEach
  - in addition, fn must return a value that can be coerced to a boolean

```
var isEven = function (x) { return x % 2 === 0; }
var evens = arr.filter(function (element) { return isEven(element); });
```

To use new Arrray methods in ECMAScript 3 see http://erik.eae.net/playground/arrayextras/

... New Array Methods ...

- **reduce(fn[, initialValue])**
  - fn is passed the current result, the current element, its index, and the array
  - for the first call to fn
    - if initialValue is specfied, it is the current result
    - if initialValue is not specified,
      the first element is the current result and the second element is the current element
  - for subsequent calls to fn
    - the current result is the value returned by the previous call to fn

```
var sum = arr.reduce(function (x, y) { return x + y; });
```

- **reduceRight(fn[, initialValue])**
  - same as reduce, but elements are processed
    from right to left instead of left to right

... New Array Methods

- **every(fn[, thisInFn])**

```
if (arr.every(isEven)) { ... }
```

isEven is defined on slide 21

  - fn takes same arguments as in forEach
  - in addition, fn must return a value that can be coerced to a boolean
  - stops and returns false the first time fn returns false;
    otherwise returns true

can use these to avoid writing loops that break out before evaluating all the elements in an array

```
var names =
  'Mark Tami Amanda Jeremy'.split(' '),
  picked;
names.some(function (name) {
  console.log('evaluating ' + name);
  var pick = name.length > 4;
  if (pick) picked = name;
  return pick;
});
console.log(picked); // Amanda
```

- **some(fn[, thisInFn])**

```
if (arr.some(isEven)) { ... }
```

  - same as every, but
    stops and returns true the first time fn returns true;
    otherwise returns false

New Function Method

- **`bind(thisInFn[, initialArgs])`**
  - returns a new function that invokes a given function with the value of **`this`** bound to a given object and initial arguments bound
  - **`thisInFn`** is the value of **`this`** in **`fn`**
  - can perform partial application
    - means creating a new function that invokes a given function with predefined values for some or all of the parameters starting at the beginning

```
var product = function (x, y) { return x * y; }
var arr1 = [1, 2, 3];
var arr2 = arr1.map(product.bind(null, 5));
// arr2 = [5, 10, 15]

var times5 = product.bind(null, 5);
arr2 = arr1.map(times5);
// same result
```

```
// Suppose f1 is a function
// that takes a callback,
// f2 is the callback,
// and it takes two arguments.
// The following are equivalent.
f1(function () { f2(a, b); });
f1(f2.bind(null, a, b));
```

  - useful for callbacks that take arguments
    - without bind, an anonymous function must be used

New JSON Object ...

- Improves security
  - old way of creating JavaScript objects from JSON text simply executes the text as code and creates objects by treating JSON as an object literal
  - new way verifies that text being parsed is valid JSON and doesn't execute arbitrary JavaScript code

- Creating an object from a JSON string

```
var obj = JSON.parse(json[, reviverFunction]);
```

  - optional reviver function
    - has **`key`** and **`value`** parameters
    - return **`undefined`** to delete the property
    - return some other value to transform it (ex. transform date strings to **`Date`** objects)

... New JSON Object

- Creating a JSON string from an object

```
var json = JSON.stringify(object[, replacer[, space]]);
```

  - objects with a **toJSON** method are stringified using that
  - optional **replacer** argument can be
    - a function that is passed each value to be stringified
      - return value is stringified instead of the original value
    - an array of names of properties to be included in result
  - optional **space** argument can be
    - a string or number of spaces to be used in indented output for human readability
    - maximum indentation increment is ten spaces or characters

JSON doesn't support cycles in object relationships

- Can use in ES3 by downloading **json2.js**
  - from Douglas Crockford
  - see link at bottom of http://www.json.org/js.html

JSON Example

```
function Address(street, city, state, zip) {
  this.street = street;
  this.city = city;
  this.state = state;
  this.zip = zip;
}

function Person(name, address) {
  this.name = name;
  this.address = address;
}

var a = new Address(
  '644 Glen Summit', 'St. Charles', 'MO', 63304);
var p = new Person('Mark', a);
var json = JSON.stringify(p);
console.log(json);
var newP = JSON.parse(json);
console.log(newP.name + ' ' + newP.address.zip);
// Mark 63304
```

Output on one line:

```
{
  "name":"Mark",
  "address":{
    "street":"644 Glen Summit",
    "city":"St. Charles",
    "state":"MO",
    "zip":63304
  }
}
```

Other Changes

- Constructors
  - cannot be called without **`new`**
    - ex. `Foo()` instead of `new Foo()`
    - when not in strict mode, `this` is undefined and setting properties in the constructor may throw an error

Is any function that begins with an uppercase letter considered to be a constructor function???

- Objects
  - trailing commas in object literals are allowed
    - ex. `{ foo: "hello", bar: "world", }`
  - the global object cannot be accessed ???

Resources

- ECMAScript 5 Objects and Properties
  - John Resig, http://ejohn.org/blog/ecmascript-5-objects-and-properties/
- ECMAScript 5 Strict Mode, JSON, and More
  - John Resig, http://ejohn.org/blog/ecmascript-5-strict-mode-json-and-more/
- ECMAScript 5: The Definitive Slides
  - David Flanagan, http://davidflanagan.com/Talks/es5/slides.html
- ECMAScript 5 Compatibility Table
  - http://kangax.github.com/es5-compat-table/
  - thanks to Bill Edney for telling me about this!