**Object and Object.prototype Methods**

**1.Object.defineProperty()**

/\* \*/

(function() {

// using \_\_proto\_\_

var obj = {};

Object.defineProperty(obj, 'key1', {

\_\_proto\_\_: null, // no inherited properties

value: 'static1' // not enumerable

// not configurable

// not writable

// as defaults

});

// being explicit

Object.defineProperty(obj, 'key2', {

enumerable: false,

configurable: false,

writable: false,

value: 'static2'

});

// recycling same object

function withValue(value) {

var d = withValue.d || (

withValue.d = {

enumerable: false,

writable: false,

configurable: false,

value: null

}

);

d.value = value;

return d;

}

Object.defineProperty(obj, 'key3', withValue('static3'));

}());

**2. Object.defineProperties()**

/\* \*/

(function () {

var obj = {};

Object.defineProperties(obj, {

"property1": {

value: true,

writable: true

},

"property2": {

value: "Hello",

writable: false

}

// etc. etc.

});

}());

**3. Object.create()**

/\* Classical inheritance with Object.create() \*/

(function () {

// Shape - superclass

function Shape() {

this.x = 0;

this.y = 0;

}

// superclass method

Shape.prototype.move = function (x, y) {

this.x += x;

this.y += y;

console.info('Shape moved.');

};

// Rectangle - subclass

function Rectangle() {

Shape.call(this); // call super constructor.

}

// subclass extends superclass

Rectangle.prototype = Object.create(Shape.prototype);

Rectangle.prototype.constructor = Rectangle;

var rect = new Rectangle();

console.log('Is rect an instance of Rectangle? ' + (rect instanceof Rectangle)); // true

console.log('Is rect an instance of Shape? ' + (rect instanceof Shape)); // true

rect.move(1, 1); // Outputs, 'Shape moved.'

}());

/\* Using propertiesObject argument with Object.create() \*/

(function () {

var o;

// create an object with null as prototype

o = Object.create(null);

o = {};

// is equivalent to:

o = Object.create(Object.prototype);

// Example where we create an object with a couple of sample properties.

// (Note that the second parameter maps keys to \*property descriptors\*.)

o = Object.create(Object.prototype, {

// foo is a regular 'value property'

foo: {

writable: true,

configurable: true,

value: 'hello'

},

// bar is a getter-and-setter (accessor) property

bar: {

configurable: false,

get: function () {

return 10;

},

set: function (value) {

console.log('Setting `o.bar` to', value);

}

}

});

function Constructor() {

}

o = new Constructor();

// is equivalent to:

o = Object.create(Constructor.prototype);

// Of course, if there is actual initialization code in the

// Constructor function, the Object.create() cannot reflect it

// create a new object whose prototype is a new, empty object

// and adding single property 'p', with value 42

o = Object.create({}, {p: {value: 42}});

// by default properties ARE NOT writable, enumerable or configurable:

o.p = 24;

console.log(o.p); // 42

o.q = 12;

for (var prop in o) {

console.log(prop); // 'q'

}

delete o.p; // false

}());

/\*\*/

**4. Object.keys()**

/\* \*/

(function () {

var arr = ['a', 'b', 'c'];

console.log(Object.keys(arr)); // console: ['0', '1', '2']

// array like object

var obj = {0: 'a', 1: 'b', 2: 'c'};

console.log(Object.keys(obj)); // console: ['0', '1', '2']

// array like object with random key ordering

var an\_obj = {100: 'a', 2: 'b', 7: 'c'};

console.log(Object.keys(an\_obj)); // console: ['2', '7', '100']

// getFoo is property which isn't enumerable

var my\_obj = Object.create({}, {

getFoo: {

value: function () {

return this.foo;

}

}

});

my\_obj.foo = 1;

console.log(Object.keys(my\_obj)); // console: ['foo']

}());

**6. Object.preventExtensions()**

/\* \*/

(function () {

// Object.preventExtensions returns the object being made non-extensible.

var obj = {};

var obj2 = Object.preventExtensions(obj);

console.log(obj === obj2);

// Objects are extensible by default.

var empty = {};

console.log('is "empty" extensible: ' + Object.isExtensible(empty));

// ...but that can be changed.

Object.preventExtensions(empty);

console.log('is "empty" extensible: ' + Object.isExtensible(empty));

// Object.defineProperty throws when adding a new property to a non-extensible object.

var nonExtensible = {removable: true};

Object.preventExtensions(nonExtensible);

//Object.defineProperty(nonExtensible, 'new', { value: 8675309 }); // throws a TypeError

// In strict mode, attempting to add new properties to a non-extensible object throws a TypeError.

function fail() {

'use strict';

nonExtensible.newProperty = 'FAIL'; // throws a TypeError

}

//fail();

console.log('is "nonExtensible" extensiblxe: ' + Object.isExtensible(nonExtensible));

}());

**7. Object.isExtensible()**

/\* \*/

(function() {

// New objects are extensible.

var empty = {};

console.log('is "empty" extensible: ' + Object.isExtensible(empty));

// ...but that can be changed.

Object.preventExtensions(empty);

console.log('is "empty" extensible: ' + Object.isExtensible(empty));

// Sealed objects are by definition non-extensible.

var sealed = Object.seal({});

console.log('is "sealed" extensible: ' + Object.isExtensible(sealed));

// Frozen objects are also by definition non-extensible.

var frozen = Object.freeze({});

console.log('is "frozen" extensible: ' + Object.isExtensible(frozen));

}());

**8.Object.seal()**

/\* \*/

(function () {

var o, obj = {

prop: function () {

},

foo: 'bar'

};

// New properties may be added, existing properties may be changed or removed.

obj.foo = 'baz';

obj.lumpy = 'woof';

delete obj.prop;

o = Object.seal(obj);

console.log(o === obj);

console.log(Object.isSealed(obj) === true);

// Changing property values on a sealed object still works.

obj.foo = 'quux';

console.log('obj.foo = ' + obj.foo);

// But you can't convert data properties to accessors, or vice versa.

//Object.defineProperty(obj, 'foo', { get: function() { return 'g'; } }); // throws a TypeError

// Now any changes, other than to property values, will fail.

obj.quaxxor = 'the friendly duck'; // silently doesn't add the property

console.log('obj.quaxxor = ' + obj.quaxxor);

delete obj.foo; // silently doesn't delete the property

console.log('obj.foo = ' + obj.foo);

// ...and in strict mode such attempts will throw TypeErrors.

function fail() {

'use strict';

delete obj.foo; // throws a TypeError

obj.sparky = 'arf'; // throws a TypeError

}

fail();

// Attempted additions through Object.defineProperty will also throw.

Object.defineProperty(obj, 'ohai', {value: 17}); // throws a TypeError

Object.defineProperty(obj, 'foo', {value: 'eit'}); // changes existing property value

}());

**9.Object.isSealed()**

/\* \*/

(function() {

// Objects aren't sealed by default.

var empty = {};

console.log('is "empty" sealed: ' + Object.isSealed(empty));

// If you make an empty object non-extensible, it is vacuously sealed.

Object.preventExtensions(empty);

console.log('is "empty" sealed: ' + Object.isSealed(empty));

// The same is not true of a non-empty object, unless its properties are all non-configurable.

var hasProp = {fee: 'fie foe fum'};

Object.preventExtensions(hasProp);

console.log('is "hasProp" sealed: ' + Object.isSealed(hasProp));

// But make them all non-configurable and the object becomes sealed.

Object.defineProperty(hasProp, 'fee', {configurable: false});

console.log('is "hasProp" sealed: ' + Object.isSealed(hasProp));

// The easiest way to seal an object, of course, is Object.seal.

var sealed = {};

Object.seal(sealed);

console.log('is "sealed" sealed: ' + Object.isSealed(sealed));

// A sealed object is, by definition, non-extensible.

console.log('is "sealed" extensible: ' + Object.isExtensible(sealed));

// A sealed object might be frozen, but it doesn't have to be.

console.log('is "sealed" frozen: ' + Object.isFrozen(sealed)); // all properties also non-writable

var s2 = Object.seal({p: 3});

console.log('is "s2" frozen: ' + Object.isFrozen(s2)); // 'p' is still writable

var s3 = Object.seal({

get p() {

return 0;

}

});

// only configurability matters for accessor properties

console.log('is "sealed" frozen: ' + Object.isFrozen(s3));

}());

**10.Object.freeze()**

/\* Object.freeze() \*/

(function() {

var o, obj = {

prop: function() {},

foo: 'bar'

};

// New properties may be added, existing properties may be changed or removed

obj.foo = 'baz';

obj.lumpy = 'woof';

delete obj.prop;

o = Object.freeze(obj);

console.log(Object.isFrozen(obj) === true);

// Now any changes will fail

obj.foo = 'quux'; // silently does nothing

obj.quaxxor = 'the friendly duck'; // silently doesn't add the property

// ...and in strict mode such attempts will throw TypeErrors

function fail(){

'use strict';

obj.foo = 'sparky'; // throws a TypeError

delete obj.quaxxor; // throws a TypeError

obj.sparky = 'arf'; // throws a TypeError

}

fail();

// Attempted changes through Object.defineProperty will also throw

Object.defineProperty(obj, 'ohai', { value: 17 }); // throws a TypeError

Object.defineProperty(obj, 'foo', { value: 'eit' }); // throws a TypeError

}());

**11.Object.deepFreeze()**

/\* object values in a frozen object can be mutated (freeze is shallow) \*/

(function() {

obj = {

internal: {}

};

Object.freeze(obj);

obj.internal.a = 'aValue';

console.log(obj.internal.a); // 'aValue'

// To make obj fully immutable, freeze each object in obj.

// To do so, we use this function.

function deepFreeze(o) {

var prop, propKey;

Object.freeze(o); // First freeze the object.

for (propKey in o) {

prop = o[propKey];

if (!o.hasOwnProperty(propKey) || !(typeof prop === 'object') || Object.isFrozen(prop)) {

// If the object is on the prototype, not an object, or is already frozen,

// skip it. Note that this might leave an unfrozen reference somewhere in the

// object if there is an already frozen object containing an unfrozen object.

continue;

}

deepFreeze(prop); // Recursively call deepFreeze.

}

}

obj2 = {

internal: {}

};

deepFreeze(obj2);

obj2.internal.a = 'anotherValue';

console.log(obj2.internal.a); // undefined

}());

**12.Object.isFrozen()**

/\* Object.isFrozen() \*/

(function() {

// A new object is extensible, so it is not frozen.

console.log('is "{}" frozen: ' + Object.isFrozen({}));

// An empty object which is not extensible is vacuously frozen.

var vacuouslyFrozen = Object.preventExtensions({});

console.log('is "vacuouslyFrozen" frozen: ' + Object.isFrozen(vacuouslyFrozen));

// A new object with one property is also extensible, ergo not frozen.

var oneProp = {p: 42};

console.log('is "oneProp" frozen: ' + Object.isFrozen(oneProp));

// Preventing extensions to the object still doesn't make it frozen,

// because the property is still configurable (and writable).

Object.preventExtensions(oneProp);

console.log('is "oneProp" frozen after preventing extensions: ' + Object.isFrozen(oneProp));

// ...but then deleting that property makes the object vacuously frozen.

delete oneProp.p;

console.log('is "oneProp" frozen after deleting "p": ' + Object.isFrozen(oneProp));

// A non-extensible object with a non-writable but still configurable property is not frozen.

var nonWritable = {e: 'plep'};

Object.preventExtensions(nonWritable);

Object.defineProperty(nonWritable, 'e', {writable: false}); // make non-writable

console.log('is "nonWritable" frozen with e non-writable: ' + Object.isFrozen(nonWritable));

// Changing that property to non-configurable then makes the object frozen.

Object.defineProperty(nonWritable, 'e', {configurable: false}); // make non-configurable

console.log('is "nonWritable" frozen with e non-configurable: ' + Object.isFrozen(nonWritable));

// A non-extensible object with a non-configurable but still writable property also isn't frozen.

var nonConfigurable = {release: 'the kraken!'};

Object.preventExtensions(nonConfigurable);

Object.defineProperty(nonConfigurable, 'release', {configurable: false});

console.log('is "nonConfigurable" frozen with "release" non-configurable: ' + Object.isFrozen(nonConfigurable));

// Changing that property to non-writable then makes the object frozen.

Object.defineProperty(nonConfigurable, 'release', {writable: false});

console.log('is "nonConfigurable" frozen with "release" non-writable: ' + Object.isFrozen(nonConfigurable));

// A non-extensible object with a configurable accessor property isn't frozen.

var accessor = {

get food() {

return 'yum';

}

};

Object.preventExtensions(accessor);

console.log('is "accessor" frozen: ' + Object.isFrozen(accessor));

// ...but make that property non-configurable and it becomes frozen.

Object.defineProperty(accessor, 'food', {configurable: false});

console.log('is "accessor" frozen with "food" non-configurable: ' + Object.isFrozen(accessor));

// But the easiest way for an object to be frozen is if Object.freeze has been called on it.

var frozen = {1: 81};

console.log('is "frozen" frozen: ' + Object.isFrozen(frozen));

Object.freeze(frozen);

console.log('is "frozen" frozen: ' + Object.isFrozen(frozen));

// By definition, a frozen object is non-extensible.

console.log('is "frozen" extensible: ' + Object.isExtensible(frozen));

// Also by definition, a frozen object is sealed.

console.log('is "frozen" sealed: ' + Object.isSealed(frozen));

});

**13.Object.assign()**

/\* Object.assign() \*/

// Polyfill

if (!Object.assign) {

Object.defineProperty(Object, 'assign', {

enumerable: false,

configurable: true,

writable: true,

value: function(target) {

'use strict';

if (target === undefined || target === null) {

throw new TypeError('Cannot convert first argument to object');

}

var to = Object(target);

for (var i = 1; i < arguments.length; i++) {

var nextSource = arguments[i];

if (nextSource === undefined || nextSource === null) {

continue;

}

nextSource = Object(nextSource);

var keysArray = Object.keys(Object(nextSource));

for (var nextIndex = 0, len = keysArray.length; nextIndex < len; nextIndex++) {

var nextKey = keysArray[nextIndex];

var desc = Object.getOwnPropertyDescriptor(nextSource, nextKey);

if (desc !== undefined && desc.enumerable) {

to[nextKey] = nextSource[nextKey];

}

}

}

return to;

}

});

}

// Cloning an object

(function() {

var obj = {a: 1},

copy = Object.assign({}, obj);

console.log(copy); // { a: 1 }

}());

// Merging objects

(function() {

var obj,

o1 = {a: 1},

o2 = {b: 2},

o3 = {c: 3};

obj = Object.assign(o1, o2, o3);

console.log(obj); // { a: 1, b: 2, c: 3 }

console.log(o1); // { a: 1, b: 2, c: 3 }, target object itself is changed.

}());

// Inherit properties and non-enumerable properties cannot be copied

(function() {

var obj, copy;

obj = Object.create({ foo: 1 }, { // foo is an inherit property.

bar: {

value: 2 // bar is a non-enumerable property.

},

baz: {

value: 3,

enumerable: true // baz is an own enumerable property.

}

});

copy = Object.assign({}, obj);

console.log(copy);

console.log(obj);

console.log('copy.bar = ' + copy.bar);

console.log('obj.bar = ' + obj.bar);

}());

**14.Object.is()**

/\* Object.is() \*/

(function() {

console.log(Object.is('foo', 'foo')); // true

console.log(Object.is(window, window)); // true

console.log(Object.is('foo', 'bar')); // false

console.log(Object.is([], [])); // false

var test = {a: 1};

console.log(Object.is(test, test)); // true

console.log(console.log(Object.is(null, null))); // true

// Special Cases

console.log(Object.is(0, -0)); // false

console.log(Object.is(-0, -0)); // true

console.log(Object.is(NaN, 0 / 0)); // true

}());

**15.Object.prototype.hasOwnProperty()**

/\* Object.prototype.hasOwnProperty() \*/

// Using hasOwnProperty to test for a property's existence

(function() {

o = new Object();

o.prop = 'exists';

function changeO() {

o.newprop = o.prop;

delete o.prop;

}

console.log('"o" has property "prop": ' + o.hasOwnProperty('prop')); // returns true

changeO();

console.log('"o" has property "prop": ' + o.hasOwnProperty('prop')); // returns false

}());

// Direct versus inherited properties

(function() {

var o = new Object();

o.prop = 'exists';

console.log('"o" has property "prop": ' + o.hasOwnProperty('prop')); // returns true

console.log('"o" has property "toString": ' + o.hasOwnProperty('toString')); // returns false

console.log('"o" has property "hasOwnProperty": ' + o.hasOwnProperty('hasOwnProperty')); // returns false

}());

// Iterating over the properties of an object

(function() {

var buz = {

fog: 'stack'

};

for (var name in buz) {

if (buz.hasOwnProperty(name)) {

console.log('this is fog (' + name + ') for sure. Value: ' + buz[name]);

}

else {

console.log(name); // toString or something else

}

}

}());

// hasOwnProperty as a property

(function() {

var foo = {

hasOwnProperty: function() {

return false;

},

bar: 'Here be dragons'

};

console.log('"foo" has property "bar": ' + foo.hasOwnProperty('bar')); // always returns false

// Use another Object's hasOwnProperty and call it with 'this' set to foo

console.log('"foo" has property "bar": ' + ({}).hasOwnProperty.call(foo, 'bar')); // true

// It's also possible to use the hasOwnProperty property from the Object prototype for this purpose

console.log('"foo" has property "bar": ' + Object.prototype.hasOwnProperty.call(foo, 'bar')); // true

}());

**16.Object.prototype.isPrototypeOf()**

/\* Object.prototype.hasOwnProperty() \*/

// Using hasOwnProperty to test for a property's existence

(function() {

o = new Object();

o.prop = 'exists';

function changeO() {

o.newprop = o.prop;

delete o.prop;

}

console.log('"o" has property "prop": ' + o.hasOwnProperty('prop')); // returns true

changeO();

console.log('"o" has property "prop": ' + o.hasOwnProperty('prop')); // returns false

}());

// Direct versus inherited properties

(function() {

var o = new Object();

o.prop = 'exists';

console.log('"o" has property "prop": ' + o.hasOwnProperty('prop')); // returns true

console.log('"o" has property "toString": ' + o.hasOwnProperty('toString')); // returns false

console.log('"o" has property "hasOwnProperty": ' + o.hasOwnProperty('hasOwnProperty')); // returns false

}());

// Iterating over the properties of an object

(function() {

var buz = {

fog: 'stack'

};

for (var name in buz) {

if (buz.hasOwnProperty(name)) {

console.log('this is fog (' + name + ') for sure. Value: ' + buz[name]);

}

else {

console.log(name); // toString or something else

}

}

}());

// hasOwnProperty as a property

(function() {

var foo = {

hasOwnProperty: function() {

return false;

},

bar: 'Here be dragons'

};

console.log('"foo" has property "bar": ' + foo.hasOwnProperty('bar')); // always returns false

// Use another Object's hasOwnProperty and call it with 'this' set to foo

console.log('"foo" has property "bar": ' + ({}).hasOwnProperty.call(foo, 'bar')); // true

// It's also possible to use the hasOwnProperty property from the Object prototype for this purpose

console.log('"foo" has property "bar": ' + Object.prototype.hasOwnProperty.call(foo, 'bar')); // true

}());

**17.Object.prtotype.propertyIsEnumerable()**

/\* Object.prototype.propertyIsEnumerable() \*/

// A basic use of propertyIsEnumerable

(function() {

var o = {};

var a = [];

o.prop = 'is enumerable';

a[0] = 'is enumerable';

console.log('is "prop" enumerable: ' + o.propertyIsEnumerable('prop')); // returns true

console.log('is "0" enumerable: ' + a.propertyIsEnumerable(0)); // returns true

}());

// User-defined versus built-in objects

(function() {

var a = ['is enumerable'];

console.log('is "0" enumerable: ' + a.propertyIsEnumerable(0)); // returns true

console.log('is "length" enumerable: ' + a.propertyIsEnumerable('length')); // returns false

console.log('is "Math.random" enumerable: ' + Math.propertyIsEnumerable('random')); // returns false

console.log('is "this.Math" enumerable: ' + this.propertyIsEnumerable('Math')); // returns false

}());

// Direct versus inherited properties

(function() {

var a = [];

console.log('is "[].constructor" enumerable: ' + a.propertyIsEnumerable('constructor')); // returns false

function FirstConstructor() {

this.property = 'is not enumerable';

}

FirstConstructor.prototype.firstMethod = function() {};

function SecondConstructor() {

this.method = function method() { return 'is enumerable'; };

}

SecondConstructor.prototype = new FirstConstructor;

SecondConstructor.prototype.constructor = SecondConstructor;

var o = new SecondConstructor();

o.arbitraryProperty = 'is enumerable';

console.log('is "o.arbitraryProo.perty" enumerable: ' + o.propertyIsEnumerable('arbitraryProperty')); // returns true

console.log('is "o.method" enumerable: ' + o.propertyIsEnumerable('method')); // returns true

console.log('is "o.property" enumerable: ' + o.propertyIsEnumerable('property')); // returns false

o.property = 'is enumerable';

console.log('is "o.property" enumerable: ' + o.propertyIsEnumerable('property')); // returns true

// These return false as they are on the prototype which

// propertyIsEnumerable does not consider (even though the last two are iteratable with for-in)

console.log('is "o.prototype" enumerable: ' + o.propertyIsEnumerable('prototype')); // returns false (as of JS 1.8.1/FF3.6)

console.log('is "o.constructor" enumerable: ' + o.propertyIsEnumerable('constructor')); // returns false

console.log('is "o.firstMethod" enumerable: ' + o.propertyIsEnumerable('firstMethod')); // returns false

}());

**18. Object.prototype.ValueOf()**

/\* Object.prototype.valueOf() \*/

(function () {

var o = new Object(),

myVar = o.valueOf();

console.log('' + myVar); // [object Object]

}());