Object Methods

25 November 2022 21:02

1. Object.values()

The **Object.values()** method returns an array of a given object's own enumerable property values, in the same order as that provided by a for...in loop. (The only difference is that a for...in loop enumerates properties in the prototype chain as well.)

From <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values>

JavaScript Demo: Object.values()

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false
};

console.log(Object.values(object1));
// expected output: Array ["somestring", 42, false]

```

2. Object.seal()

The **Object.seal()** method *seals* an object. Sealing an object prevents extensions and makes existing properties non-configurable. A sealed object has a fixed set of properties: new properties cannot be added, existing properties cannot be removed, their enumerability and configurability cannot be changed, and its prototype cannot be re-assigned. Values of existing properties can still be changed as long as they are writable. seal() returns the same object that was passed in.

From <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/seal>

JavaScript Demo: Object.seal()

```
const object1 = {
  property1: 42
};

Object.seal(object1);
object1.property1 = 33;
console.log(object1.property1);
// expected output: 33

delete object1.property1; // cannot delete when sealed
console.log(object1.property1);
// expected output: 33

```

3. Object.preventExtensions()

The **Object.preventExtensions()** method prevents new properties from ever being added to an object (i.e. prevents future extensions to the object). It also prevents the object's prototype from being re-assigned.

From <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/preventExtensions>

JavaScript Demo: Object.preventExtensions()

```
const object1 = {};

Object.preventExtensions(object1);

try {
  Object.defineProperty(object1, 'property1', {
    value: 42
  });
} catch (e) {
  console.log(e);
  // expected output: TypeError: Cannot define property property1, object is not extensible
}

```

4. Object.keys()

The **Object.keys()** method returns an array of a given object's own enumerable property **names**, iterated in the same order that a normal loop would.

From <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys>

JavaScript Demo: Object.keys()

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false
};

console.log(Object.keys(object1));
// expected output: Array ["a", "b", "c"]

```

5. Object.isSealed()

The **Object.isSealed()** method determines if an object is sealed.

From <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/isSealed>

JavaScript Demo: Object.isSealed()

```
const object1 = {
  property1: 42
};

console.log(Object.isSealed(object1));
// expected output: false

Object.seal(object1);

console.log(Object.isSealed(object1));
// expected output: true

```

6. Object.isExtensible()

The **Object.isExtensible()** method determines if an object is extensible (whether it can have new properties added to it).

From <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/isExtensible>

JavaScript Demo: Object.isExtensible()

```
1  const object1 = {};
2
3  console.log(Object.isExtensible(object1));
4  // expected output: true
5
6  Object.preventExtensions(object1);
7
8  console.log(Object.isExtensible(object1));
9  // expected output: false
10
```

7. Object.is()

The **Object.is()** method determines whether two values are the same value.

From <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is>

Object.is() determines whether two values are the same value. Two values are the same if one of the following holds:

- both undefined
- both null
- both true or both false
- both strings of the same length with the same characters in the same order
- both the same object (meaning both values reference the same object in memory)
- both BigInts with the same numeric value
- both symbols that reference the same symbol value
- both numbers and
- both +0
- both -0
- both NaN
- or both non-zero, not NaN, and have the same value

Object.is() is not equivalent to the == operator. The == operator applies various coercions to both sides (if they are not the same type) before testing for equality (resulting in such behavior as "" == false being true), but Object.is() doesn't coerce either value.
Object.is() is also *not* equivalent to the === operator. The only difference between Object.is() and === is in their treatment of signed zeros and NaN values.
The === operator (and the == operator) treats the number values -0 and +0 as equal, but treats NaN as not equal to each other.

From <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is>

```
// Case 1: Evaluation result is the same as using ===
Object.is(25, 25); // true
Object.is("foo", "foo"); // true
Object.is("foo", "bar"); // false
Object.is(null, null); // true
Object.is(undefined, undefined); // true
Object.is(window, window); // true
Object.is([], []); // false
const foo = { a: 1 };
const bar = { a: 1 };
const sameFoo = foo;
Object.is(foo, foo); // true
Object.is(foo, bar); // false
Object.is(foo, sameFoo); // true

// Case 2: Signed zero
Object.is(0, -0); // false
Object.is(+0, -0); // false
Object.is(-0, -0); // true

// Case 3: NaN
Object.is(NaN, 0 / 0); // true
Object.is(NaN, Number.NaN); // true
```

8. Object.hasOwn()

The **Object.hasOwn()** static method returns true if the specified object has the indicated property as its *own* property. If the property is inherited, or does not exist, the method returns false.

From <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn>

JavaScript Demo: Object.hasOwn()

```
1  const object1 = {
2    prop: 'exists'
3  };
4
5  console.log(Object.hasOwn(object1, 'prop'));
6  // expected output: true
7
8  console.log(Object.hasOwn(object1, 'toString'));
9  // expected output: false
10
11 console.log(Object.hasOwn(object1, 'undeclaredPropertyValue'));
12 // expected output: false
13
```

9. Object.entries()

The **Object.entries()** method returns an array of a given object's own enumerable string-keyed property [key, value] pairs. This is the same as iterating with a for...in loop, except that a for...in loop enumerates properties in the prototype chain as well.
The order of the array returned by Object.entries() is the same as that provided by

a for...in loop. If there is a need for different ordering, then the array should be sorted first, like Object.entries(obj).sort((a, b) => a[0].localeCompare(b[0]));.

From <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries>

JavaScript Demo: Object.entries()

```
const object1 = {
  a: 'somestring',
  b: 42
};

for (const [key, value] of Object.entries(object1)) {
  console.log(`${key}: ${value}`);
}

// expected output:
// "a: somestring"
// "b: 42"

```

10. Object.fromEntries()

The **Object.fromEntries()** method transforms a list of key-value pairs into an object.

From <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/fromEntries>

JavaScript Demo: Object.fromEntries()

```
const entries = new Map([
  ['foo', 'bar'],
  ['baz', 42]
]);

const obj = Object.fromEntries(entries);

console.log(obj);
// expected output: Object { foo: "bar", baz: 42 }

```