The page you requested has been retired, so we've sent you to the English equivalent.

Object.assign()

The `Object.assign()` static method copies all enumerable own properties from one or more *source objects* to a *target object*. It returns the modified target object.

Try it

JavaScript Demo: Object.assign()

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source)

console.log(target);
// Expected output: Object { a: 1, b: 4, c: 5 }

console.log(returnedTarget === target);
// Expected output: true

```

Syntax

```
Object.assign(target, ...sources)
```

Parameters

`target`

The target object — what to apply the sources' properties to, which is returned after it is modified.

`sources`

The source object(s) — objects containing the properties you want to apply.

Return value

The target object.

Description

Properties in the target object are overwritten by properties in the sources if they have the same key. Later sources' properties overwrite earlier ones.

The `Object.assign()` method only copies *enumerable* and *own* properties from a source object to a target object. It uses `[[Get]]` on the source and `[[Set]]` on the target, so it w
getters and setters. Therefore it *assigns* properties, versus copying or defining new properties. This may make it unsuitable for merging new properties into a prototype if the m
sources contain getters.

For copying property definitions (including their enumerability) into prototypes, use `Object.getOwnPropertyDescriptor()` and `Object.defineProperty()` instead.

Both `String` and `Symbol` properties are copied.

In case of an error, for example if a property is non-writable, a `TypeError` is raised, and the `target` object is changed if any properties are added before the error is raised.

> **Note:** `Object.assign()` does not throw on `null` or `undefined` sources.

Examples

Cloning an object

```
const obj = { a: 1 };
const copy = Object.assign({}, obj);
console.log(copy); // { a: 1 }
```

Warning for Deep Clone

For deep cloning, we need to use alternatives, because `Object.assign()` copies property values.

If the source value is a reference to an object, it only copies the reference value.

```
const obj1 = { a: 0, b: { c: 0 } };
const obj2 = Object.assign({}, obj1);
console.log(obj2); // { a: 0, b: { c: 0 } }

obj1.a = 1;
console.log(obj1); // { a: 1, b: { c: 0 } }
console.log(obj2); // { a: 0, b: { c: 0 } }

obj2.a = 2;
console.log(obj1); // { a: 1, b: { c: 0 } }
console.log(obj2); // { a: 2, b: { c: 0 } }

obj2.b.c = 3;
console.log(obj1); // { a: 1, b: { c: 3 } }
console.log(obj2); // { a: 2, b: { c: 3 } }

// Deep Clone
const obj3 = { a: 0, b: { c: 0 } };
const obj4 = JSON.parse(JSON.stringify(obj3));
obj3.a = 4;
obj3.b.c = 4;
console.log(obj4); // { a: 0, b: { c: 0 } }
```

Merging objects

```
const o1 = { a: 1 };
const o2 = { b: 2 };
const o3 = { c: 3 };

const 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.
```

Merging objects with same properties

```
const o1 = { a: 1, b: 1, c: 1 };
const o2 = { b: 2, c: 2 };
const o3 = { c: 3 };

const obj = Object.assign({}, o1, o2, o3);
console.log(obj); // { a: 1, b: 2, c: 3 }
```

The properties are overwritten by other objects that have the same properties later in the parameters order.

Copying symbol-typed properties

```
const o1 = { a: 1 };
const o2 = { [Symbol("foo")]: 2 };

const obj = Object.assign({}, o1, o2);
console.log(obj); // { a : 1, [Symbol("foo")]: 2 } (cf. bug 1207182 on Firefox)
Object.getOwnPropertySymbols(obj); // [Symbol(foo)]
```

Properties on the prototype chain and non-enumerable properties cannot be copied

```
const obj = Object.create(
  // foo is on obj's prototype chain.
  { foo: 1 },
  {
    bar: {
      value: 2, // bar is a non-enumerable property.
    },
    baz: {
      value: 3,
      enumerable: true, // baz is an own enumerable property.
    },
  },
);

const copy = Object.assign({}, obj);
console.log(copy); // { baz: 3 }
```

Primitives will be wrapped to objects

```
const v1 = "abc";
const v2 = true;
const v3 = 10;
const v4 = Symbol("foo");
const obj = Object.assign({}, v1, null, v2, undefined, v3, v4);
// Primitives will be wrapped, null and undefined will be ignored.
// Note, only string wrappers can have own enumerable properties.
console.log(obj); // { "0": "a", "1": "b", "2": "c" }
```

Exceptions will interrupt the ongoing copying task

```
const target = Object.defineProperty({}, "foo", {
  value: 1,
  writable: false,
}); // target.foo is a read-only property

Object.assign(target, { bar: 2 }, { foo2: 3, foo: 3, foo3: 3 }, { baz: 4 });
// TypeError: "foo" is read-only
// The Exception is thrown when assigning target.foo

console.log(target.bar); // 2, the first source was copied successfully.
console.log(target.foo2); // 3, the first property of the second source was copied successfully.
console.log(target.foo); // 1, exception is thrown here.
console.log(target.foo3); // undefined, assign method has finished, foo3 will not be copied.
console.log(target.baz); // undefined, the third source will not be copied either.
```

Copying accessors

```
const obj = {
  foo: 1,
  get bar() {
    return 2;
  },
};

let copy = Object.assign({}, obj);
console.log(copy);
// { foo: 1, bar: 2 }
// The value of copy.bar is obj.bar's getter's return value.

// This is an assign function that copies full descriptors
function completeAssign(target, ...sources) {
  sources.forEach((source) => {
    const descriptors = Object.keys(source).reduce((descriptors, key) => {
      descriptors[key] = Object.getOwnPropertyDescriptor(source, key);
      return descriptors;
    }, {});
    // By default, Object.assign copies enumerable Symbols, too
    Object.getOwnPropertySymbols(source).forEach((sym) => {
      const descriptor = Object.getOwnPropertyDescriptor(source, sym);
      if (descriptor.enumerable) {
        descriptors[sym] = descriptor;
      }
    });
    Object.defineProperties(target, descriptors);
  });
  return target;
}

copy = completeAssign({}, obj);
console.log(copy);
// { foo:1, get bar() { return 2 } }
```

Specifications

Specification
[ECMAScript Language Specification](#) [# sec-object.assign](#)

Browser compatibility

[Report problems with this compatibility data on GitHub](#)

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android
`assign`	Chrome 45	Edge 12	Firefox 34	Opera 32	Safari 9	Chrome 45 Android	Firefox 34 for Android	Opera 32 Android	Safari 9 on iOS	Samsung 5.0 Internet	WebVie Androi

Tip: you can click/tap on a cell for more information.

Full support

See also

- Polyfill of `Object.assign` in `core-js`
- `Object.defineProperties()`
- Enumerability and ownership of properties
- Spread in object literals

This page was last modified on Feb 21, 2023 by MDN contributors.