Proxy

The **`Proxy`** object enables you to create a proxy for another object, which can intercept and redefine fundamental operations for that object.

Description

The `Proxy` object allows you to create an object that can be used in place of the original object, but which may redefine fundamental `Object` operations like getting, setting, and defining properties. Proxy objects are commonly used to log property accesses, validate, format, or sanitize inputs, and so on.

You create a `Proxy` with two parameters:

- `target`: the original object which you want to proxy
- `handler`: an object that defines which operations will be intercepted and how to redefine intercepted operations.

For example, this code creates a proxy for the `target` object.

```
const target = {
  message1: "hello",
  message2: "everyone",
};

const handler1 = {};

const proxy1 = new Proxy(target, handler1);
```

Because the handler is empty, this proxy behaves just like the original target:

```
console.log(proxy1.message1); // hello
console.log(proxy1.message2); // everyone
```

To customize the proxy, we define functions on the handler object:

```
const target = {
  message1: "hello",
  message2: "everyone",
};

const handler2 = {
  get(target, prop, receiver) {
    return "world";
  },
};

const proxy2 = new Proxy(target, handler2);
```

Here we've provided an implementation of the `get()` handler, which intercepts attempts to access properties in the target.

Handler functions are sometimes called *traps*, presumably because they trap calls to the target object. The very simple trap in `handler2` above redefines all property accessors:

```
console.log(proxy2.message1); // world
console.log(proxy2.message2); // world
```

Proxies are often used with the `Reflect` object, which provides some methods with the same names as the `Proxy` traps. The `Reflect` methods provide the reflective semantics for invoking the corresponding object internal methods. For example, we can call `Reflect.get` if we don't wish to redefine the object's behavior:

```
const target = {
  message1: "hello",
  message2: "everyone",
};

const handler3 = {
  get(target, prop, receiver) {
    if (prop === "message2") {
      return "world";
    }
    return Reflect.get(...arguments);
  },
};

const proxy3 = new Proxy(target, handler3);

console.log(proxy3.message1); // hello
console.log(proxy3.message2); // world
```

The `Reflect` method still interacts with the object through object internal methods — it doesn't "de-proxify" the proxy if it's invoked on a proxy. If you use `Reflect` methods within a proxy trap, and the `Reflect` method call gets intercepted by the trap again, there may be infinite recursion.

Terminology

The following terms are used when talking about the functionality of proxies.

handler

The object passed as the second argument to the `Proxy` constructor. It contains the traps which define the behavior of the proxy.

trap

The function that define the behavior for the corresponding object internal method. (This is analogous to the concept of *traps* in operating systems.)

target

Object which the proxy virtualizes. It is often used as storage backend for the proxy. Invariants (semantics that remain unchanged) regarding object non-extensibility or non-configurable properties are verified against the target.

invariants

Semantics that remain unchanged when implementing custom operations. If your trap implementation violates the invariants of a handler, a `TypeError` will be thrown.

Object internal methods

Objects are collections of properties. However, the language doesn't provide any machinery to *directly* manipulate data stored in the object — rather, the object defines some internal methods specifying how it can be interacted with. For example, when you read `obj.x`, you may expect the following to happen:

- The `x` property is searched up the prototype chain until it is found.
- If `x` is a data property, the property descriptor's `value` attribute is returned.
- If `x` is an accessor property, the getter is invoked, and the return value of the getter is returned.

There isn't anything special about this process in the language — it's just because ordinary objects, by default, have a `[[Get]]` internal method that is defined with this behavior. The `obj.x` property access syntax simply invokes the `[[Get]]` method on the object, and the object uses its own internal method implementation to determine what to return.

As another example, arrays differ from normal objects, because they have a magic `length` property that, when modified, automatically allocates empty slots or removes elements from the array. Similarly, adding array elements automatically changes the `length` property. This is because arrays have a `[[DefineOwnProperty]]` internal method that knows to update `length` when an integer index is written to, or update the array contents when `length` is written to. Such objects whose internal methods have different implementations from ordinary objects are called *exotic objects*. `Proxy` enable developers to define their own exotic objects with full capacity.

Internal method	Corresponding trap
`[[GetPrototypeOf]]`	`getPrototypeOf()`
`[[SetPrototypeOf]]`	`setPrototypeOf()`
`[[IsExtensible]]`	`isExtensible()`
`[[PreventExtensions]]`	`preventExtensions()`
`[[GetOwnProperty]]`	`getOwnPropertyDescriptor()`
`[[DefineOwnProperty]]`	`defineProperty()`
`[[HasProperty]]`	`has()`
`[[Get]]`	`get()`
`[[Set]]`	`set()`
`[[Delete]]`	`deleteProperty()`
`[[OwnPropertyKeys]]`	`ownKeys()`

Function objects also have the following internal methods:

Internal method	Corresponding trap
`[[Call]]`	`apply()`

Internal method	Corresponding trap
`[[Construct]]`	`construct()`

It's important to realize that all interactions with an object eventually boils down to the invocation of one of these internal methods, and that they are all customizable through proxies. This means almost no behavior (except certain critical invariants) is guaranteed in the language — everything is defined by the object itself. When you run `delete obj.x`, there's no guarantee that `"x" in obj` returns `false` afterwards — it depends on the object's implementations of `[[Delete]]` and `[[HasProperty]]`. A `delete obj.x` may log things to the console, modify some global state, or even define a new property instead of deleting the existing one, although these semantics should be avoided in your own code.

All internal methods are called by the language itself, and are not directly accessible in JavaScript code. The `Reflect` namespace offers methods that do little more than call the internal methods, besides some input normalization/validation. In each trap's page, we list several typical situations when the trap is invoked, but these internal methods are called in *a lot* of places. For example, array methods read and write to array through these internal methods, so methods like `push()` would also invoke `get()` and `set()` traps.

Most of the internal methods are straightforward in what they do. The only two that may be confusable are `[[Set]]` and `[[DefineOwnProperty]]`. For normal objects, the former invokes setters; the latter doesn't. (And `[[Set]]` calls `[[DefineOwnProperty]]` internally if there's no existing property or the property is a data property.) While you may know that the `obj.x = 1` syntax uses `[[Set]]`, and `Object.defineProperty()` uses `[[DefineOwnProperty]]`, it's not immediately apparent what semantics other built-in methods and syntaxes use. For example, class fields use the `[[DefineOwnProperty]]` semantic, which is why setters defined in the superclass are not invoked when a field is declared on the derived class.

Constructor

`Proxy()`

Creates a new `Proxy` object.

Note: There's no `Proxy.prototype` property, so `Proxy` instances do not have any special properties or methods.

Static methods

`Proxy.revocable()`

Creates a revocable `Proxy` object.

Examples

Basic example

In this simple example, the number `37` gets returned as the default value when the property name is not in the object. It is using the `get()` handler.

```
const handler = {
  get(obj, prop) {
    return prop in obj ? obj[prop] : 37;
  },
};

const p = new Proxy({}, handler);
p.a = 1;
p.b = undefined;
```

```
console.log(p.a, p.b); // 1, undefined

console.log("c" in p, p.c); // false, 37
```

No-op forwarding proxy

In this example, we are using a native JavaScript object to which our proxy will forward all operations that are applied to it.

```
const target = {};
const p = new Proxy(target, {});

p.a = 37; // Operation forwarded to the target

console.log(target.a); // 37 (The operation has been properly forwarded!)
```

Note that while this "no-op" works for plain JavaScript objects, it does not work for native objects, such as DOM elements, `Map` objects, or anything that has internal slots. See no private property forwarding for more information.

No private property forwarding

A proxy is still another object with a different identity — it's a *proxy* that operates between the wrapped object and the outside. As such, the proxy does not have direct access to the original object's private properties.

```
class Secret {
  #secret;
  constructor(secret) {
    this.#secret = secret;
  }
  get secret() {
    return this.#secret.replace(/\d+/, "[REDACTED]");
  }
}

const aSecret = new Secret("123456");
console.log(aSecret.secret); // [REDACTED]
// Looks like a no-op forwarding...
const proxy = new Proxy(aSecret, {});
console.log(proxy.secret); // TypeError: Cannot read private member #secret from an object whose class did not declare it
```

This is because when the proxy's `get` trap is invoked, the `this` value is the `proxy` instead of the original `secret`, so `#secret` is not accessible. To fix this, use the original `secret` as `this`:

```
const proxy = new Proxy(aSecret, {
  get(target, prop, receiver) {
    // By default, it looks like Reflect.get(target, prop, receiver)
    // which has a different value of `this`
    return target[prop];
  },
});
console.log(proxy.secret);
```

For methods, this means you have to redirect the method's `this` value to the original object as well:

```
class Secret {
  #x = 1;
  x() {
    return this.#x;
  }
}

const aSecret = new Secret();
const proxy = new Proxy(aSecret, {
  get(target, prop, receiver) {
    const value = target[prop];
    if (value instanceof Function) {
      return function (...args) {
        return value.apply(this === receiver ? target : this, args);
      };
    }
    return value;
  },
});
console.log(proxy.x());
```

Some native JavaScript objects have properties called *internal slots*, which are not accessible from JavaScript code. For example, `Map` objects have an internal slot called `[[MapData]]`, which stores the key-value pairs of the map. As such, you cannot trivially create a forwarding proxy for a map:

```
const proxy = new Proxy(new Map(), {});
console.log(proxy.size); // TypeError: get size method called on incompatible Proxy
```

You have to use the "`this`-recovering" proxy illustrated above to work around this.

Validation

With a `Proxy`, you can easily validate the passed value for an object. This example uses the `set()` handler.

```
const validator = {
  set(obj, prop, value) {
    if (prop === "age") {
      if (!Number.isInteger(value)) {
        throw new TypeError("The age is not an integer");
      }
      if (value > 200) {
        throw new RangeError("The age seems invalid");
      }
    }

    // The default behavior to store the value
    obj[prop] = value;

    // Indicate success
    return true;
  },
};

const person = new Proxy({}, validator);

person.age = 100;
console.log(person.age); // 100
person.age = "young"; // Throws an exception
person.age = 300; // Throws an exception
```

Manipulating DOM nodes

In this example we use `Proxy` to toggle an attribute of two different elements: so when we set the attribute on one element, the attribute is unset on the other one.

We create a `view` object which is a proxy for an object with a `selected` property. The proxy handler defines the `set()` handler.

When we assign an HTML element to `view.selected`, the element's `'aria-selected'` attribute is set to `true`. If we then assign a different element to `view.selected`, this element's `'aria-selected'` attribute is set to `true` and the previous element's `'aria-selected'` attribute is automatically set to `false`.

```
const view = new Proxy(
  {
    selected: null,
  },
  {
    set(obj, prop, newval) {
      const oldval = obj[prop];

      if (prop === "selected") {
        if (oldval) {
          oldval.setAttribute("aria-selected", "false");
        }
        if (newval) {
          newval.setAttribute("aria-selected", "true");
        }
      }

      // The default behavior to store the value
      obj[prop] = newval;

      // Indicate success
      return true;
    },
  },
);

const item1 = document.getElementById("item-1");
const item2 = document.getElementById("item-2");

// select item1:
view.selected = item1;

console.log(`item1: ${item1.getAttribute("aria-selected")}`);
// item1: true
// selecting item2 de-selects item1:
view.selected = item2;

console.log(`item1: ${item1.getAttribute("aria-selected")}`);
// item1: false

console.log(`item2: ${item2.getAttribute("aria-selected")}`);
// item2: true
```

Value correction and an extra property

The `products` proxy object evaluates the passed value and converts it to an array if needed. The object also supports an extra property called `latestBrowser` both as a getter and a setter.

```
const products = new Proxy(
  {
    browsers: ["Firefox", "Chrome"],
  },
  {
    get(obj, prop) {
      // An extra property
      if (prop === "latestBrowser") {
        return obj.browsers[obj.browsers.length - 1];
      }

      // The default behavior to return the value
      return obj[prop];
    },
    set(obj, prop, value) {
      // An extra property
      if (prop === "latestBrowser") {
        obj.browsers.push(value);
        return true;
      }

      // Convert the value if it is not an array
      if (typeof value === "string") {
        value = [value];
      }

      // The default behavior to store the value
      obj[prop] = value;

      // Indicate success
      return true;
    },
  },
);

console.log(products.browsers);
//  ['Firefox', 'Chrome']

products.browsers = "Safari";
//  pass a string (by mistake)
```

```
console.log(products.browsers);
//  ['Safari'] <- no problem, the value is an array

products.latestBrowser = "Edge";

console.log(products.browsers);
//  ['Safari', 'Edge']

console.log(products.latestBrowser);
//  'Edge'
```

Finding an array item object by its property

This proxy extends an array with some utility features. As you see, you can flexibly "define" properties without using `Object.defineProperties()`. This example can be adapted to find a table row by its cell. In that case, the target will be `table.rows`.

```
const products = new Proxy(
  [
    { name: "Firefox", type: "browser" },
    { name: "SeaMonkey", type: "browser" },
    { name: "Thunderbird", type: "mailer" },
  ],
  {
    get(obj, prop) {
      // The default behavior to return the value; prop is usually an integer
      if (prop in obj) {
        return obj[prop];
      }

      // Get the number of products; an alias of products.length
      if (prop === "number") {
        return obj.length;
      }

      let result;
      const types = {};

      for (const product of obj) {
        if (product.name === prop) {
          result = product;
        }
        if (types[product.type]) {
          types[product.type].push(product);
        } else {
          types[product.type] = [product];
        }
      }

      // Get a product by name
      if (result) {
        return result;
      }
      // Get products by type
      if (prop in types) {
        return types[prop];
      }

      // Get product types
      if (prop === "types") {
        return Object.keys(types);
      }

      return undefined;
    },
  },
);

console.log(products[0]); // { name: 'Firefox', type: 'browser' }
console.log(products["Firefox"]); // { name: 'Firefox', type: 'browser' }
console.log(products["Chrome"]); // undefined
console.log(products.browser); // [{ name: 'Firefox', type: 'browser' }, { name: 'SeaMonkey', type: 'browser' }]
console.log(products.types); // ['browser', 'mailer']
console.log(products.number); // 3
```

A complete traps list example

Now in order to create a complete sample `traps` list, for didactic purposes, we will try to proxify a *non-native* object that is particularly suited to this type of operation: the `docCookies` global object created by a simple cookie framework.

```
/*
  const docCookies = ... get the "docCookies" object here:
  https://reference.codeproject.com/dom/document/cookie/simple_document.cookie_framework
*/

const docCookies = new Proxy(docCookies, {
  get(target, key) {
    return target[key] ?? target.getItem(key) ?? undefined;
  },
  set(target, key, value) {
    if (key in target) {
      return false;
    }
    return target.setItem(key, value);
  },
  deleteProperty(target, key) {
    if (!(key in target)) {
      return false;
    }
    return target.removeItem(key);
  },
  ownKeys(target) {
    return target.keys();
  },
  has(target, key) {
    return key in target || target.hasItem(key);
  },
  defineProperty(target, key, descriptor) {
    if (descriptor && "value" in descriptor) {
      target.setItem(key, descriptor.value);
    }
    return target;
  },
  getOwnPropertyDescriptor(target, key) {
    const value = target.getItem(key);
    return value
      ? {
          value,
          writable: true,
          enumerable: true,
          configurable: false,
        }
      : undefined;
  },
});

/* Cookies test */

console.log((docCookies.myCookie1 = "First value"));
console.log(docCookies.getItem("myCookie1"));

docCookies.setItem("myCookie1", "Changed value");
console.log(docCookies.myCookie1);
```

Specifications

Specification
ECMAScript Language Specification # sec-proxy-objects

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	Deno
`Proxy`	✓ Chrome 49	✓ Edge 12	✓ Firefox 18	✓ Opera 36	✓ Safari 10	✓ Chrome 49 Android	✓ Firefox 18 for Android	✓ Opera 36 Android	✓ Safari 10 on iOS	✓ Samsung 5.0 Internet	✓ WebView 49 Android	✓ Den
`Proxy()` constructor	✓ Chrome 49	✓ Edge 12	✓ Firefox 18	✓ Opera 36	✓ Safari 10	✓ Chrome 49 Android	✓ Firefox 18 for	✓ Opera 36 Android	✓ Safari 10 on	✓ Samsung 5.0 Internet	✓ WebView 49 Android	✓ Den

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android	Deno
`revocable`	✓ Chrome 63	✓ Edge 12	✓ Firefox 34	✓ Opera 50	✓ Safari 10	✓ Chrome 63 Android	✓ Firefox 34 for Android	✓ Opera 46 Android	✓ Safari 10 on iOS	✓ Samsung 8.0 Internet	✓ WebView 63 Android	✓ Den

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

✓ Full support

See also

- "Proxies are awesome" Brendan Eich presentation at JSConf (slides)
- Tutorial on proxies

This page was last modified on Feb 23, 2023 by MDN contributors.