Meta programming

The `Proxy` and `Reflect` objects allow you to intercept and define custom behavior for fundamental language operations (e.g. property lookup, assignment, enumeration, function invocation, etc.). With the help of these two objects you are able to program at the meta level of JavaScript.

Proxies

Proxy objects allow you to intercept certain operations and to implement custom behaviors.

For example, getting a property on an object:

```
const handler = {
  get(target, name) {
    return name in target ? target[name] : 42;
  },
};

const p = new Proxy({}, handler);
p.a = 1;
console.log(p.a, p.b); // 1, 42
```

The `Proxy` object defines a `target` (an empty object here) and a `handler` object, in which a `get` trap is implemented. Here, an object that is proxied will not return `undefined` when getting undefined properties, but will instead return the number `42`.

Terminology

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

handler
: Placeholder object which contains traps.

traps
: The methods that provide property access. (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 are called *invariants*. If you violate the invariants of a handler, a `TypeError` will be thrown.

Handlers and traps

The following table summarizes the available traps available to `Proxy` objects.

Handler / trap	Interceptions	Invariants

`handler.getPrototypeOf()`	`Object.getPrototypeOf()`
	`Reflect.getPrototypeOf()`
	`__proto__`
	`Object.prototype.isPrototypeOf()`
	`instanceof`
		• `getPrototypeOf` method must return an object or `null`.
		• If `target` is not extensible, `Object.getPrototypeOf(`*`proxy`*`)` method must return the same value as `Object.getPrototypeOf(`*`target`*`).`

`handler.setPrototypeOf()`	`Object.setPrototypeOf()` `Reflect.setPrototypeOf()`	If `target` is not extensible, the prototype parameter must be the same value as `Object.getPrototypeOf(`*`target`*`).`
`handler.isExtensible()`	`Object.isExtensible()` `Reflect.isExtensible()`	`Object.isExtensible(`*`proxy`*`)` must return the same value as `Object.isExtensible(`*`target`*`).`
`handler.preventExtensions()`	`Object.preventExtensions()` `Reflect.preventExtensions()`	`Object.preventExtensions(`*`proxy`*`)` only returns `true` if `Object.isExtensible(`*`proxy`*`)` is `false`.

`handler.getOwnPropertyDescriptor()`	`Object.getOwnPropertyDescriptor()`
	
	`Reflect.getOwnPropertyDescriptor()`
		• `getOwnPropertyDescriptor` must return an object or `undefined`.
		• A property cannot be reported as non-existent if it exists as a non-configurable own property of `target`.
		• A property cannot be reported as non-existent if it exists as an own property of `target` and `target` is not extensible.
		• A property cannot be reported as existent if it does not exists as an own property of `target` and `target` is not extensible.
		• A property cannot be reported as non-configurable if it does not exist as an own property of `target` or if it exists as a configurable own property of `target`.
		• The result of `Object.getOwnPropertyDescriptor(`*`target`*`)` can be applied to `target` using `Object.defineProperty` and will not throw an exception.

`handler.defineProperty()`	`Object.defineProperty()`
	
	`Reflect.defineProperty()`
		• A property cannot be added if `target` is not extensible.
		• A property cannot be added as (or modified to be) non-configurable if it does not exist as a non-configurable own property of `target`.
		• A property may not be non-configurable if a corresponding configurable property of `target` exists.
		• If a property has a corresponding target object property, then `Object.defineProperty(target, prop, descriptor)` will not throw an exception.
		• In strict mode, a `false` value returned from the `defineProperty` handler will throw a `TypeError` exception.

`handler.has()`	Property query:
	`foo in proxy`
	
	Inherited property query:
	`foo in Object.create(`*`proxy`*`)`
	`Reflect.has()`
		• A property cannot be reported as non-existent, if it exists as a non-configurable own property of `target`.
		
		• A property cannot be reported as non-existent if it exists as an own property of `target` and `target` is not extensible.

`handler.get()`	Property access:
	`proxy[foo]`
	`proxy.bar`
	
	Inherited property access:
	`Object.create(proxy)[foo]`
	`Reflect.get()`
		• The value reported for a property must be the same as the value of the corresponding `target` property if `target`'s property is a non-writable, non-configurable data property.
		
		• The value reported for a property must be `undefined` if the corresponding `target` property is non-configurable accessor property that has undefined as its `[[Get]]` attribute.

`handler.set()`	Property assignment:
	`proxy[foo] = bar`
	`proxy.foo = bar`
	
	Inherited property assignment:
	`Object.create(`*`proxy`*`)[foo] = bar`
	`Reflect.set()`
		• Cannot change the value of a property to be different from the value of the corresponding `target` property if the corresponding `target` property is a non-writable, non-configurable data property.
		
		• Cannot set the value of a property if the corresponding `target` property is a non-configurable accessor property that has `undefined` as its `[[Set]]` attribute.
		
		• In strict mode, a `false` return value from the set handler will throw a `TypeError` exception.

`handler.deleteProperty()`	Property deletion:
	`delete proxy[foo]`
	`delete proxy.foo`
	`Reflect.deleteProperty()`
		A property cannot be deleted if it exists as a non-configurable own property of `target`.

`handler.ownKeys()`	`Object.getOwnPropertyNames()`
	
	`Object.getOwnPropertySymbols()`
	
	`Object.keys()`
	`Reflect.ownKeys()`
		• The result of `ownKeys` is a List.
		• The Type of each result List element is either `String` or `Symbol`.
		• The result List must contain the keys of all non-configurable own properties of `target`.
		• If the `target` object is not extensible, then the result List must contain all the keys of the own properties of `target` and no other values.

`handler.apply()`	`proxy(..args)`
	`Function.prototype.apply()` and `Function.prototype.call()`
	`Reflect.apply()`
		There are no invariants for the `handler.apply` method.

`handler.construct()`	`new proxy(...args)` `Reflect.construct()`	The result must be an `Object`.

Revocable Proxy

The `Proxy.revocable()` method is used to create a revocable `Proxy` object. This means that the proxy can be revoked via the function `revoke` and switches the proxy off.

Afterwards, any operation on the proxy leads to a `TypeError`.

```
const revocable = Proxy.revocable(
  {},
  {
    get(target, name) {
      return `[[${name}]]`;
    },
  },
);
const proxy = revocable.proxy;
console.log(proxy.foo); // "[[foo]]"

revocable.revoke();

console.log(proxy.foo); // TypeError: Cannot perform 'get' on a proxy that has been
                                    revoked
proxy.foo = 1; // TypeError: Cannot perform 'set' on a proxy that has been revoked
delete proxy.foo; // TypeError: Cannot perform 'deleteProperty' on a proxy that has
                                  been revoked
```

```
console.log(typeof proxy); // "object", typeof doesn't trigger any trap
```

Reflection

`Reflect` is a built-in object that provides methods for interceptable JavaScript operations. The methods are the same as those of the proxy handlers.

`Reflect` is not a function object.

`Reflect` helps with forwarding default operations from the handler to the `target`.

With `Reflect.has()` for example, you get the `in` operator as a function:

```
Reflect.has(Object, "assign"); // true
```

A better apply() function

Before `Reflect`, you typically use the `Function.prototype.apply()` method to call a function with a given `this` value and `arguments` provided as an array (or an array-like object).

```
Function.prototype.apply.call(Math.floor, undefined, [1.75]);
```

With `Reflect.apply` this becomes less verbose and easier to understand:

```
Reflect.apply(Math.floor, undefined, [1.75]);
// 1

Reflect.apply(String.fromCharCode, undefined, [104, 101, 108, 108, 111]);
// "hello"

Reflect.apply(RegExp.prototype.exec, /ab/, ["confabulation"]).index;
// 4

Reflect.apply("".charAt, "ponies", [3]);
// "i"
```

Checking if property definition has been successful

With `Object.defineProperty`, which returns an object if successful, or throws a `TypeError` otherwise, you would use a `try...catch` block to catch any error that occurred while defining a property. Because `Reflect.defineProperty` returns a Boolean success status, you can just use an `if...else` block here:

```
if (Reflect.defineProperty(target, property, attributes)) {
  // success
} else {
  // failure
}
```