Proxies and the Reflection API

ECMAScript 5 and ECMAScript 6 were both developed with demystifying JavaScript functionality in mind. For example, JavaScript environments contained nonenumerable and nonwritable object properties before ECMAScript 5, but developers couldn't define their own nonenumerable or nonwritable properties. ECMAScript 5 included the `Object.defineProperty()` method to allow developers to do what JavaScript engines could do already.

ECMAScript 6 gives developers further access to JavaScript engine capabilities previously available only to built-in objects. The language exposes the inner workings of objects through *proxies*, which are wrappers that can intercept and alter low-level operations of the JavaScript engine. This chapter starts by describing the problem that proxies are meant to address in detail, and then discusses how you can create and use proxies effectively.

The Array Problem

The JavaScript array object behaves in ways that developers couldn't mimic in their own objects before ECMASCript 6. An array's `length` property is affected when you assign values to specific array items, and you can modify array items by modifying the `length` property. For example:

```
let colors = ["red", "green", "blue"];

console.log(colors.length);         // 3

colors[3] = "black";

console.log(colors.length);         // 4
console.log(colors[3]);             // "black"

colors.length = 2;

console.log(colors.length);         // 2
console.log(colors[3]);             // undefined
console.log(colors[2]);             // undefined
console.log(colors[1]);             // "green"
```

The `colors` array starts with three items. Assigning `"black"` to `colors[3]` automatically increments the `length` property to `4`. Setting the `length` property to `2` removes the last two items in the array, leaving only the first two items. Nothing in ECMAScript 5 allows developers to achieve this behavior, but proxies change that.

I> This nonstandard behavior is why arrays are considered exotic objects in ECMAScript 6.

What are Proxies and Reflection?

You can create a proxy to use in place of another object (called the *target*) by calling `new Proxy()`. The proxy *virtualizes* the target so that the proxy and the target appear to be the same object to functionality using the proxy.

Proxies allow you to intercept low-level object operations on the target that are otherwise internal to the JavaScript engine. These low-level operations are intercepted using a *trap*, which is a function that responds to a specific operation.

The reflection API, represented by the `Reflect` object, is a collection of methods that provide the default behavior for the same low-level operations that proxies can override. There is a `Reflect` method for every proxy trap. Those methods have the same name and are passed the same arguments as their respective proxy traps. Table 11-1 summarizes this behavior.

{title="Table 11-1: Proxy traps in JavaScript"}

Proxy Trap	Overrides the Behavior Of	Default Behavior
`get`	Reading a property value	`Reflect.get()`

Proxy Trap	Overrides the Behavior Of	Default Behavior
`set`	Writing to a property	`Reflect.set()`
`has`	The `in` operator	`Reflect.has()`
`deleteProperty`	The `delete` operator	`Reflect.deleteProperty()`
`getPrototypeOf`	`Object.getPrototypeOf()`	`Reflect.getPrototypeOf()`
`setPrototypeOf`	`Object.setPrototypeOf()`	`Reflect.setPrototypeOf()`
`isExtensible`	`Object.isExtensible()`	`Reflect.isExtensible()`
`preventExtensions`	`Object.preventExtensions()`	`Reflect.preventExtensions()`
`getOwnPropertyDescriptor`	`Object.getOwnPropertyDescriptor()`	`Reflect.getOwnPropertyDescriptor()`
`defineProperty`	`Object.defineProperty()`	`Reflect.defineProperty`
`ownKeys`	`Object.keys`, `Object.getOwnPropertyNames()`, `Object.getOwnPropertySymbols()`	`Reflect.ownKeys()`
`apply`	Calling a function	`Reflect.apply()`
`construct`	Calling a function with `new`	`Reflect.construct()`

Each trap overrides some built-in behavior of JavaScript objects, allowing you to intercept and modify the behavior. If you still need to use the built-in behavior, then you can use the corresponding reflection API method. The relationship between proxies and the reflection API becomes clear when you start creating proxies, so it's best to dive in and look at some examples.

I> The original ECMAScript 6 specification had an additional trap called `enumerate` that was designed to alter how `for-in` and `Object.keys()` enumerated properties on an object. However, the `enumerate` trap was removed in ECMAScript 7 (also called ECMAScript 2016) as difficulties were discovered during implementation. The `enumerate` trap no longer exists in any JavaScript environment and is therefore not covered in this chapter.

Creating a Simple Proxy

When you use the `Proxy` constructor to make a proxy, you'll pass it two arguments: the target and a handler. A *handler* is an object that defines one or more traps. The proxy uses the default behavior for all operations except when traps are defined for that operation. To create a simple forwarding proxy, you can use a handler without any traps:

```
let target = {};

let proxy = new Proxy(target, {});

proxy.name = "proxy";
console.log(proxy.name);        // "proxy"
console.log(target.name);       // "proxy"

target.name = "target";
console.log(proxy.name);        // "target"
console.log(target.name);       // "target"
```

In this example, `proxy` forwards all operations directly to `target`. When `"proxy"` is assigned to the `proxy.name` property, `name` is created on `target`. The proxy itself is not storing this property; it's simply forwarding the operation to `target`. Similarly, the values of `proxy.name` and `target.name` are the same because they are both references to `target.name`. That also means setting `target.name` to a new value causes `proxy.name` to reflect the same change. Of course, proxies without traps aren't very interesting, so what happens when you define a trap?

Validating Properties Using the `set` Trap

Suppose you want to create an object whose property values must be numbers. That means every new property added to the object must be validated, and an error must be thrown if the value isn't a number. To accomplish this, you could define a `set` trap that overrides the default behavior of setting a value. The `set` trap receives four arguments:

1. `trapTarget` - the object that will receive the property (the proxy's target)
2. `key` - the property key (string or symbol) to write to
3. `value` - the value being written to the property
4. `receiver` - the object on which the operation took place (usually the proxy)

`Reflect.set()` is the `set` trap's corresponding reflection method, and it's the default behavior for this operation. The `Reflect.set()` method accepts the same four arguments as the `set` proxy trap, making the method easy to use inside of the trap. The trap should return `true` if the property was set or `false` if not. (The `Reflect.set()` method returns the correct value based on whether the operation succeeded.)

To validate the values of properties, you'd use the `set` trap and inspect the `value` that is passed in. Here's an example:

```
let target = {
    name: "target"
};

let proxy = new Proxy(target, {
    set(trapTarget, key, value, receiver) {

        // ignore existing properties so as not to affect them
        if (!trapTarget.hasOwnProperty(key)) {
            if (isNaN(value)) {
                throw new TypeError("Property must be a number.");
            }
        }

        // add the property
        return Reflect.set(trapTarget, key, value, receiver);
    }
});

// adding a new property
proxy.count = 1;
console.log(proxy.count);       // 1
console.log(target.count);      // 1

// you can assign to name because it exists on target already
proxy.name = "proxy";
console.log(proxy.name);        // "proxy"
console.log(target.name);       // "proxy"

// throws an error
proxy.anotherName = "proxy";
```

This code defines a proxy trap that validates the value of any new property added to `target`. When `proxy.count = 1` is executed, the `set` trap is called. The `trapTarget` value is equal to `target`, `key` is `"count"`, `value` is `1`, and `receiver` (not used in this example) is `proxy`. There is no existing property named `count` in `target`, so the proxy validates `value` by passing it to `isNaN()`. If the result is `NaN`, then the property value is not numeric and an error is thrown. Since this code sets `count` to `1`, the proxy calls `Reflect.set()` with the same four arguments that were passed to the trap to add the new property.

When `proxy.name` is assigned a string, the operation completes successfully. Since `target` already has a `name` property, that property is omitted from the validation check by calling the `trapTarget.hasOwnProperty()` method. This ensures that previously-existing non-numeric property values are still supported.

When `proxy.anotherName` is assigned a string, however, an error is thrown. The `anotherName` property doesn't exist on the target, so its value needs to be validated. During validation, the error is thrown because `"proxy"` isn't a numeric value.

Where the `set` proxy trap lets you intercept when properties are being written to, the `get` proxy trap lets you intercept when properties are being read.

Object Shape Validation Using the `get` Trap

One of the interesting, and sometimes confusing, aspects of JavaScript is that reading nonexistent properties doesn't throw an error. Instead, the value `undefined` is used for the property value, as in this example:

```
let target = {};

console.log(target.name);       // undefined
```

In most other languages, attempting to read `target.name` throws an error because the property doesn't exist. But JavaScript just uses `undefined` for the value of the `target.name` property. If you've ever worked on a large code base, you've probably seen how this behavior can cause significant problems, especially when there's a typo in the property name. Proxies can help you save yourself from this problem by having object shape validation.

An *object shape* is the collection of properties and methods available on the object. JavaScript engines use object shapes to optimize code, often creating classes to represent the objects. If you can safely assume an object will always have the same properties and methods it began with (a behavior you can enforce with the `Object.preventExtensions()` method, the `Object.seal()` method, or the `Object.freeze()` method), then throwing an error on attempts to access nonexistent properties can be helpful. Proxies make object shape validation easy.

Since property validation only has to happen when a property is read, you'd use the `get` trap. The `get` trap is called when a property is read, even if that property doesn't exist on the object, and it takes three arguments:

1. `trapTarget` - the object from which the property is read (the proxy's target)
2. `key` - the property key (a string or symbol) to read
3. `receiver` - the object on which the operation took place (usually the proxy)

These arguments mirror the `set` trap's arguments, with one noticeable difference. There's no `value` argument here because `get` traps don't write values. The `Reflect.get()` method accepts the same three arguments as the `get` trap and returns the property's default value.

You can use the `get` trap and `Reflect.get()` to throw an error when a property doesn't exist on the target, as follows:

```
let proxy = new Proxy({}, {
        get(trapTarget, key, receiver) {
            if (!(key in receiver)) {
                throw new TypeError("Property " + key + " doesn't exist.");
            }

            return Reflect.get(trapTarget, key, receiver);
        }
    });

// adding a property still works
proxy.name = "proxy";
console.log(proxy.name);            // "proxy"

// nonexistent properties throw an error
console.log(proxy.nme);             // throws error
```

In this example, the `get` trap intercepts property read operations. The `in` operator is used to determine if the property already exists on the `receiver`. The `receiver` is used with `in` instead of `trapTarget` in case `receiver` is a proxy with a `has` trap, a type I'll cover in the next section. Using `trapTarget` in this case would sidestep the `has` trap and potentially give you the wrong result. An error is thrown if the property doesn't exist, and otherwise, the default behavior is used.

This code allows new properties like `proxy.name` to be added, written to, and read from with no problems. The last line contains a typo: `proxy.nme` should probably be `proxy.name` instead. This throws an error because `nme` doesn't exist as a property.

Hiding Property Existence Using the `has` Trap

The `in` operator determines whether a property exists on a given object and returns `true` if there is either an own property or a prototype property matching the name or symbol. For example:

```
let target = {
    value: 42;
}

console.log("value" in target);     // true
console.log("toString" in target);  // true
```

Both `value` and `toString` exist on `object`, so in both cases the `in` operator returns `true`. The `value` property is an own property while `toString` is a prototype property (inherited from `Object`). Proxies allow you to intercept this operation and return a different value for `in` with the `has` trap.

The `has` trap is called whenever the `in` operator is used. When called, two arguments are passed to the `has` trap:

1. `trapTarget` - the object the property is read from (the proxy's target)
2. `key` - the property key (string or symbol) to check

The `Reflect.has()` method accepts these same arguments and returns the default response for the `in` operator. Using the `has` trap and `Reflect.has()` allows you to alter the behavior of `in` for some properties while falling back to default behavior for others. For instance, suppose you just want to hide the `value` property. You can do so like this:

```
let target = {
    name: "target",
    value: 42
};

let proxy = new Proxy(target, {
    has(trapTarget, key) {

        if (key === "value") {
            return false;
        } else {
            return Reflect.has(trapTarget, key);
        }
    }
});


console.log("value" in proxy);      // false
console.log("name" in proxy);       // true
console.log("toString" in proxy);   // true
```

The `has` trap for `proxy` checks to see if `key` is `"value"` returns `false` if so. Otherwise, the default behavior is used via a call to the `Reflect.has()` method. As a result, the `in` operator returns `false` for the `value` property even though `value` actually exists on the target. The other properties, `name` and `toString`, correctly return `true` when used with the `in` operator.

Preventing Property Deletion with the `deleteProperty` Trap

The `delete` operator removes a property from an object and returns `true` when successful and `false` when unsuccessful. In strict mode, `delete` throws an error when you attempt to delete a nonconfigurable property; in nonstrict mode, `delete` simply returns `false`. Here's an example:

```
let target = {
    name: "target",
    value: 42
};

Object.defineProperty(target, "name", { configurable: false });

console.log("value" in target);     // true

let result1 = delete target.value;
console.log(result1);               // true

console.log("value" in target);     // false

// Note: The following line throws an error in strict mode
let result2 = delete target.name;
console.log(result2);               // false

console.log("name" in target);      // true
```

The `value` property is deleted using the `delete` operator and, as a result, the `in` operator returns `false` in the third `console.log()` call. The nonconfigurable `name` property can't be deleted so the `delete` operator simply returns `false` (if this code is run in strict mode, an error is thrown instead). You can alter this behavior by using the `deleteProperty` trap in a proxy.

The `deleteProperty` trap is called whenever the `delete` operator is used on an object property. The trap is passed two arguments:

1. `trapTarget` - the object from which the property should be deleted (the proxy's target)
2. `key` - the property key (string or symbol) to delete

The `Reflect.deleteProperty()` method provides the default implementation of the `deleteProperty` trap and accepts the same two arguments. You can combine `Reflect.deleteProperty()` and the `deleteProperty` trap to change how the `delete` operator behaves. For instance, you could ensure that the `value` property can't be deleted:

```
let target = {
    name: "target",
    value: 42
};

let proxy = new Proxy(target, {
    deleteProperty(trapTarget, key) {

        if (key === "value") {
            return false;
        } else {
            return Reflect.deleteProperty(trapTarget, key);
        }
    }
});

// Attempt to delete proxy.value

console.log("value" in proxy);      // true
```

```
let result1 = delete proxy.value;
console.log(result1);               // false

console.log("value" in proxy);      // true

// Attempt to delete proxy.name

console.log("name" in proxy);       // true

let result2 = delete proxy.name;
console.log(result2);               // true

console.log("name" in proxy);       // false
```

This code is very similar to the `has` trap example in that the `deleteProperty` trap checks to see if the `key` is `"value"` and returns `false` if so. Otherwise, the default behavior is used by calling the `Reflect.deleteProperty()` method. The `value` property can't be deleted through `proxy` because the operation is trapped, but the `name` property is deleted as expected. This approach is especially useful when you want to protect properties from deletion without throwing an error in strict mode.

Prototype Proxy Traps

Chapter 4 introduced the `Object.setPrototypeOf()` method that ECMAScript 6 added to complement the ECMAScript 5 `Object.getPrototypeOf()` method. Proxies allow you to intercept execution of both methods through the `setPrototypeOf` and `getPrototypeOf` traps. In both cases, the method on `Object` calls the trap of the corresponding name on the proxy, allowing you to alter the methods' behavior.

Since there are two traps associated with prototype proxies, there's a set of methods associated with each type of trap. The `setPrototypeOf` trap receives these arguments:

1. `trapTarget` - the object for which the prototype should be set (the proxy's target)
2. `proto` - the object to use for as the prototype

These are the same arguments passed to the `Object.setPrototypeOf()` and `Reflect.setPrototypeOf()` methods. The `getPrototypeOf` trap, on the other hand, only receives the `trapTarget` argument, which is the argument passed to the `Object.getPrototypeOf()` and `Reflect.getPrototypeOf()` methods.

How Prototype Proxy Traps Work

There are some restrictions on these traps. First, the `getPrototypeOf` trap must return an object or `null`, and any other return value results in a runtime error. The return value check ensures that `Object.getPrototypeOf()` will always return an expected value. Similarly, the return value of the `setPrototypeOf` trap must be `false` if the operation doesn't succeed. When `setPrototypeOf` returns `false`, `Object.setPrototypeOf()` throws an error. If `setPrototypeOf` returns any value other than `false`, then `Object.setPrototypeOf()` assumes the operation succeeded.

The following example hides the prototype of the proxy by always returning `null` and also doesn't allow the prototype to be changed:

```
let target = {};
let proxy = new Proxy(target, {
    getPrototypeOf(trapTarget) {
        return null;
    },
    setPrototypeOf(trapTarget, proto) {
        return false;
    }
});

let targetProto = Object.getPrototypeOf(target);
let proxyProto = Object.getPrototypeOf(proxy);

console.log(targetProto === Object.prototype);      // true
console.log(proxyProto === Object.prototype);       // false
console.log(proxyProto);                            // null

// succeeds
Object.setPrototypeOf(target, {});

// throws error
Object.setPrototypeOf(proxy, {});
```

This code emphasizes the difference between the behavior of `target` and `proxy`. While `Object.getPrototypeOf()` returns a value for `target`, it returns `null` for `proxy` because the `getPrototypeOf` trap is called. Similarly, `Object.setPrototypeOf()` succeeds when used on `target` but throws an error when used on `proxy` due to the `setPrototypeOf` trap.

If you want to use the default behavior for these two traps, you can use the corresponding methods on `Reflect`. For instance, this code implements the default behavior for the `getPrototypeOf` and `setPrototypeOf` traps:

```
let target = {};
let proxy = new Proxy(target, {
    getPrototypeOf(trapTarget) {
        return Reflect.getPrototypeOf(trapTarget);
    },
    setPrototypeOf(trapTarget, proto) {
        return Reflect.setPrototypeOf(trapTarget, proto);
    }
});

let targetProto = Object.getPrototypeOf(target);
let proxyProto = Object.getPrototypeOf(proxy);

console.log(targetProto === Object.prototype);      // true
console.log(proxyProto === Object.prototype);       // true

// succeeds
Object.setPrototypeOf(target, {});

// also succeeds
Object.setPrototypeOf(proxy, {});
```

In this example, you can use `target` and `proxy` interchangeably and get the same results because the `getPrototypeOf` and `setPrototypeOf` traps are just passing through to use the default implementation. It's important that this example use the `Reflect.getPrototypeOf()` and `Reflect.setPrototypeOf()` methods rather than the methods of the same name on `Object` due to some important differences.

Why Two Sets of Methods?

The confusing aspect of `Reflect.getPrototypeOf()` and `Reflect.setPrototypeOf()` is that they look suspiciously similar to the `Object.getPrototypeOf()` and `Object.setPrototypeOf()` methods. While both sets of methods perform similar operations, there are some distinct differences between the two.

To begin, `Object.getPrototypeOf()` and `Object.setPrototypeOf()` are higher-level operations that were created for developer use from the start. The `Reflect.getPrototypeOf()` and `Reflect.setPrototypeOf()` methods are lower-level operations that give developers access to the previously internal-only `[[GetPrototypeOf]]` and `[[SetPrototypeOf]]` operations. The `Reflect.getPrototypeOf()` method is the wrapper for the internal `[[GetPrototypeOf]]` operation (with some input validation). The `Reflect.setPrototypeOf()` method and `[[SetPrototypeOf]]` have the same relationship. The corresponding methods on `Object` also call `[[GetPrototypeOf]]` and `[[SetPrototypeOf]]` but perform a few steps before the call and inspect the return value to determine how to behave.

The `Reflect.getPrototypeOf()` method throws an error if its argument is not an object, while `Object.getPrototypeOf()` first coerces the value into an object before performing the operation. If you were to pass a number into each method, you'd get a different result:

```
let result1 = Object.getPrototypeOf(1);
console.log(result1 === Number.prototype);  // true

// throws an error
Reflect.getPrototypeOf(1);
```

The `Object.getPrototypeOf()` method allows you to retrieve a prototype for the number `1` because it first coerces the value into a `Number` object and then returns `Number.prototype`. The `Reflect.getPrototypeOf()` method doesn't coerce the value, and since `1` isn't an object, it throws an error.

The `Reflect.setPrototypeOf()` method also has a few more differences from the `Object.setPrototypeOf()` method. First, `Reflect.setPrototypeOf()` returns a boolean value indicating whether the operation was successful. A `true` value is returned for success, and `false` is returned for failure. If `Object.setPrototypeOf()` fails, it throws an error.

As the first example under "How Prototype Proxy Traps Work" showed, when the `setPrototypeOf` proxy trap returns `false`, it causes `Object.setPrototypeOf()` to throw an error. The `Object.setPrototypeOf()` method returns the first argument as its value and therefore isn't suitable for implementing the default behavior of the `setPrototypeOf` proxy trap. The following code demonstrates these differences:

```
let target1 = {};
let result1 = Object.setPrototypeOf(target1, {});
console.log(result1 === target1);                   // true

let target2 = {};
let result2 = Reflect.setPrototypeOf(target2, {});
console.log(result2 === target2);                   // false
console.log(result2);                               // true
```

In this example, `Object.setPrototypeOf()` returns `target1` as its value, but `Reflect.setPrototypeOf()` returns `true`. This subtle difference is very important. You'll see more seemingly duplicate methods on `Object` and `Reflect`, but always be sure to use the method on `Reflect` inside any proxy traps.

I> Both sets of methods will call the `getPrototypeOf` and `setPrototypeOf` proxy traps when used on a proxy.

Object Extensibility Traps

ECMAScript 5 added object extensibility modification through the `Object.preventExtensions()` and `Object.isExtensible()` methods, and ECMAScript 6 allows proxies to intercept those method calls to the underlying objects through the `preventExtensions` and `isExtensible` traps. Both traps receive a single argument called `trapTarget` that is the object on which the method was called. The `isExtensible` trap must return a boolean value indicating whether the object is extensible while the `preventExtensions` trap must return a boolean value indicating if the operation succeeded.

There are also `Reflect.preventExtensions()` and `Reflect.isExtensible()` methods to implement the default behavior. Both return boolean values, so they can be used directly in their corresponding traps.

Two Basic Examples

To see object extensibility traps in action, consider the following code, which implements the default behavior for the `isExtensible` and `preventExtensions` traps:

```
let target = {};
let proxy = new Proxy(target, {
    isExtensible(trapTarget) {
        return Reflect.isExtensible(trapTarget);
    },
    preventExtensions(trapTarget) {
        return Reflect.preventExtensions(trapTarget);
    }
});


console.log(Object.isExtensible(target));       // true
console.log(Object.isExtensible(proxy));        // true

Object.preventExtensions(proxy);

console.log(Object.isExtensible(target));       // false
console.log(Object.isExtensible(proxy));        // false
```

This example shows that both `Object.preventExtensions()` and `Object.isExtensible()` correctly pass through from `proxy` to `target`. You can, of course, also change the behavior. For example, if you don't want to allow `Object.preventExtensions()` to succeed on your proxy, you could return `false` from the `preventExtensions` trap:

```
let target = {};
let proxy = new Proxy(target, {
    isExtensible(trapTarget) {
        return Reflect.isExtensible(trapTarget);
    },
    preventExtensions(trapTarget) {
        return false
    }
});


console.log(Object.isExtensible(target));       // true
console.log(Object.isExtensible(proxy));        // true

Object.preventExtensions(proxy);

console.log(Object.isExtensible(target));       // true
console.log(Object.isExtensible(proxy));        // true
```

Here, the call to `Object.preventExtensions(proxy)` is effectively ignored because the `preventExtensions` trap returns `false`. The operation isn't forwarded to the underlying `target`, so `Object.isExtensible()` returns `true`.

Duplicate Extensibility Methods

You may have noticed that, once again, there are seemingly duplicate methods on `Object` and `Reflect`. In this case, they're more similar than not. The methods `Object.isExtensible()` and `Reflect.isExtensible()` are similar except when passed a non-object value. In that case, `Object.isExtensible()` always returns `false` while `Reflect.isExtensible()` throws an error. Here's an example of that behavior:

```
let result1 = Object.isExtensible(2);
console.log(result1);                       // false

// throws error
let result2 = Reflect.isExtensible(2);
```

This restriction is similar to the difference between the `Object.getPrototypeOf()` and `Reflect.getPrototypeOf()` methods, as the method with lower-level functionality has stricter error checks than its higher-level counterpart.

The `Object.preventExtensions()` and `Reflect.preventExtensions()` methods are also very similar. The `Object.preventExtensions()` method always returns the value that was passed to it as an argument even if the value isn't an object. The `Reflect.preventExtensions()` method, on the other hand, throws an error if the argument isn't an object; if the argument is an object, then `Reflect.preventExtensions()` returns `true` when the operation succeeds or `false` if not. For example:

```
let result1 = Object.preventExtensions(2);
console.log(result1);                               // 2

let target = {};
let result2 = Reflect.preventExtensions(target);
console.log(result2);                               // true

// throws error
let result3 = Reflect.preventExtensions(2);
```

Here, `Object.preventExtensions()` passes through the value `2` as its return value even though `2` isn't an object. The `Reflect.preventExtensions()` method returns `true` when an object is passed to it and throws an error when `2` is passed to it.

Property Descriptor Traps

One of the most important features of ECMAScript 5 was the ability to define property attributes using the `Object.defineProperty()` method. In previous versions of JavaScript, there was no way to define an accessor property, make a property read-only, or make a property nonenumerable. All of these are possible with the `Object.defineProperty()` method, and you can retrieve those attributes with the `Object.getOwnPropertyDescriptor()` method.

Proxies let you intercept calls to `Object.defineProperty()` and `Object.getOwnPropertyDescriptor()` using the `defineProperty` and `getOwnPropertyDescriptor` traps, respectively. The `defineProperty` trap receives the following arguments:

1. `trapTarget` - the object on which the property should be defined (the proxy's target)
2. `key` - the string or symbol for the property
3. `descriptor` - the descriptor object for the property

The `defineProperty` trap requires you to return `true` if the operation is successful and `false` if not. The `getOwnPropertyDescriptor` traps receives only `trapTarget` and `key`, and you are expected to return the descriptor. The corresponding `Reflect.defineProperty()` and `Reflect.getOwnPropertyDescriptor()` methods accept the same arguments as their proxy trap counterparts. Here's an example that just implements the default behavior for each trap:

```
let proxy = new Proxy({}, {
    defineProperty(trapTarget, key, descriptor) {
        return Reflect.defineProperty(trapTarget, key, descriptor);
    },
    getOwnPropertyDescriptor(trapTarget, key) {
        return Reflect.getOwnPropertyDescriptor(trapTarget, key);
    }
});


Object.defineProperty(proxy, "name", {
    value: "proxy"
});
```

```
console.log(proxy.name);            // "proxy"

let descriptor = Object.getOwnPropertyDescriptor(proxy, "name");

console.log(descriptor.value);      // "proxy"
```

This code defines a property called `"name"` on the proxy with the `Object.defineProperty()` method. The property descriptor for that property is then retrieved by the `Object.getOwnPropertyDescriptor()` method.

Blocking Object.defineProperty()

The `defineProperty` trap requires you to return a boolean value to indicate whether the operation was successful. When `true` is returned, `Object.defineProperty()` succeeds as usual; when `false` is returned, `Object.defineProperty()` throws an error. You can use this functionality to restrict the kinds of properties that the `Object.defineProperty()` method can define. For instance, if you want to prevent symbol properties from being defined, you could check that the key is a string and return `false` if not, like this:

```
let proxy = new Proxy({}, {
    defineProperty(trapTarget, key, descriptor) {

        if (typeof key === "symbol") {
            return false;
        }

        return Reflect.defineProperty(trapTarget, key, descriptor);
    }
});


Object.defineProperty(proxy, "name", {
    value: "proxy"
});

console.log(proxy.name);                    // "proxy"

let nameSymbol = Symbol("name");

// throws error
Object.defineProperty(proxy, nameSymbol, {
    value: "proxy"
});
```

The `defineProperty` proxy trap returns `false` when `key` is a symbol and otherwise proceeds with the default behavior. When `Object.defineProperty()` is called with `"name"` as the key, the method succeeds because the key is a string. When `Object.defineProperty()` is called with `nameSymbol`, it throws an error because the `defineProperty` trap returns `false`.

I> You can also have `Object.defineProperty()` silently fail by returning `true` and not calling the `Reflect.defineProperty()` method. That will suppress the error while not actually defining the property.

Descriptor Object Restrictions

To ensure consistent behavior when using the `Object.defineProperty()` and `Object.getOwnPropertyDescriptor()` methods, descriptor objects passed to the `defineProperty` trap are normalized. Objects returned from `getOwnPropertyDescriptor` trap are always validated for the same reason.

No matter what object is passed as the third argument to the `Object.defineProperty()` method, only the properties `enumerable`, `configurable`, `value`, `writable`, `get`, and `set` will be on the descriptor object passed to the `defineProperty` trap. For example:

```
let proxy = new Proxy({}, {
    defineProperty(trapTarget, key, descriptor) {
        console.log(descriptor.value);              // "proxy"
        console.log(descriptor.name);               // undefined

        return Reflect.defineProperty(trapTarget, key, descriptor);
    }
});


Object.defineProperty(proxy, "name", {
    value: "proxy",
    name: "custom"
});
```

Here, `Object.defineProperty()` is called with a nonstandard `name` property on the third argument. When the `defineProperty` trap is called, the `descriptor` object doesn't have a `name` property but does have a `value` property. That's because `descriptor` isn't a reference to the actual third argument passed to the `Object.defineProperty()` method, but rather a new object that contains only the allowable properties. The `Reflect.defineProperty()` method also ignores any nonstandard properties on the descriptor.

The `getOwnPropertyDescriptor` trap has a slightly different restriction that requires the return value to be `null`, `undefined`, or an object. If an object is returned, only `enumerable`, `configurable`, `value`, `writable`, `get`, and `set` are allowed as own properties of the object. An error is thrown if you return an object with an own property that isn't allowed, as this code shows:

```
let proxy = new Proxy({}, {
    getOwnPropertyDescriptor(trapTarget, key) {
        return {
            name: "proxy"
        };
    }
});

// throws error
let descriptor = Object.getOwnPropertyDescriptor(proxy, "name");
```

The property `name` isn't allowable on property descriptors, so when `Object.getOwnPropertyDescriptor()` is called, the `getOwnPropertyDescriptor` return value triggers an error. This restriction ensures that the value returned by `Object.getOwnPropertyDescriptor()` always has a reliable structure regardless of use on proxies.

Duplicate Descriptor Methods

Once again, ECMAScript 6 has some confusingly similar methods, as the `Object.defineProperty()` and `Object.getOwnPropertyDescriptor()` methods appear to do the same thing as the `Reflect.defineProperty()` and `Reflect.getOwnPropertyDescriptor()` methods, respectively. Like other method pairs discussed earlier in this chapter, these have some subtle but important differences.

defineProperty() Methods

The `Object.defineProperty()` and `Reflect.defineProperty()` methods are exactly the same except for their return values. The `Object.defineProperty()` method returns the first argument, while `Reflect.defineProperty()` returns `true` if the operation succeeded and `false` if not. For example:

```
let target = {};

let result1 = Object.defineProperty(target, "name", { value: "target "});

console.log(target === result1);        // true

let result2 = Reflect.defineProperty(target, "name", { value: "reflect" });

console.log(result2);                   // true
```

When `Object.defineProperty()` is called on `target`, the return value is `target`. When `Reflect.defineProperty()` is called on `target`, the return value is `true`, indicating that the operation succeeded. Since the `defineProperty` proxy trap requires a boolean value to be returned, it's better to use `Reflect.defineProperty()` to implement the default behavior when necessary.

getOwnPropertyDescriptor() Methods

The `Object.getOwnPropertyDescriptor()` method coerces its first argument into an object when a primitive value is passed and then continues the operation. On the other hand, the `Reflect.getOwnPropertyDescriptor()` method throws an error if the first argument is a primitive value. Here's an example showing both:

```
let descriptor1 = Object.getOwnPropertyDescriptor(2, "name");
console.log(descriptor1);       // undefined

// throws an error
let descriptor2 = Reflect.getOwnPropertyDescriptor(2, "name");
```

The `Object.getOwnPropertyDescriptor()` method returns `undefined` because it coerces `2` into an object, and that object has no `name` property. This is the standard behavior of the method when a property with the given name isn't found on an object. When `Reflect.getOwnPropertyDescriptor()` is called, however, an error is thrown immediately because that method doesn't accept primitive values for the first argument.

The `ownKeys` Trap

The `ownKeys` proxy trap intercepts the internal method `[[OwnPropertyKeys]]` and allows you to override that behavior by returning an array of values. This array is used in four methods: the `Object.keys()` method, the `Object.getOwnPropertyNames()` method, the `Object.getOwnPropertySymbols()` method, and the `Object.assign()` method. (The `Object.assign()` method uses the array to determine which properties to copy.)

The default behavior for the `ownKeys` trap is implemented by the `Reflect.ownKeys()` method and returns an array of all own property keys, including both strings and symbols. The `Object.getOwnProperyNames()` method and the `Object.keys()` method filter symbols out of the array and returns the result while `Object.getOwnPropertySymbols()` filters the strings out of the array and returns the result. The `Object.assign()` method uses the array with both strings and symbols.

The `ownKeys` trap receives a single argument, the target, and must always return an array or array-like object; otherwise, an error is thrown. You can use the `ownKeys` trap to, for example, filter out certain property keys that you don't want used when the `Object.keys()`, the `Object.getOwnPropertyNames()` method, the `Object.getOwnPropertySymbols()` method, or the `Object.assign()` method is used. Suppose you don't want to include any property names that begin with an underscore character, a common notation in JavaScript indicating that a field is private. You can use the `ownKeys` trap to filter out those keys as follows:

```
let proxy = new Proxy({}, {
    ownKeys(trapTarget) {
        return Reflect.ownKeys(trapTarget).filter(key => {
            return typeof key !== "string" || key[0] !== "_";
        });
    }
});

let nameSymbol = Symbol("name");

proxy.name = "proxy";
proxy._name = "private";
proxy[nameSymbol] = "symbol";

let names = Object.getOwnPropertyNames(proxy),
    keys = Object.keys(proxy);
    symbols = Object.getOwnPropertySymbols(proxy);

console.log(names.length);      // 1
console.log(names[0]);          // "name"

console.log(keys.length);      // 1
console.log(keys[0]);          // "name"

console.log(symbols.length);    // 1
console.log(symbols[0]);        // "Symbol(name)"
```

This example uses an `ownKeys` trap that first calls `Reflect.ownKeys()` to get the default list of keys for the target. Then, the `filter()` method is used to filter out keys that are strings and begin with an underscore character. Then, three properties are added to the `proxy` object: `name`, `_name`, and `nameSymbol`. When `Object.getOwnPropertyNames()` and `Object.keys()` is called on `proxy`, only the `name` property is returned. Similarly, only `nameSymbol` is returned when `Object.getOwnPropertySymbols()` is called on `proxy`. The `_name` property doesn't appear in either result because it is filtered out.

I> The `ownKeys` trap also affects the `for-in` loop, which calls the trap to determine which keys to use inside of the loop.

Function Proxies with the `apply` and `construct` Traps

Of all the proxy traps, only `apply` and `construct` require the proxy target to be a function. Recall from Chapter 3 that functions have two internal methods called `[[Call]]` and `[[Construct]]` that are executed when a function is called without and with the `new` operator, respectively. The `apply` and `construct` traps correspond to and let you override those internal methods. When a function is called without `new`, the `apply` trap receives, and `Reflect.apply()` expects, the following arguments:

1. `trapTarget` - the function being executed (the proxy's target)
2. `thisArg` - the value of `this` inside of the function during the call
3. `argumentsList` - an array of arguments passed to the function

The `construct` trap, which is called when the function is executed using `new`, receives the following arguments:

1. `trapTarget` - the function being executed (the proxy's target)
2. `argumentsList` - an array of arguments passed to the function

The `Reflect.construct()` method also accepts these two arguments and has an optional third argument called `newTarget`. When given, the `newTarget` argument specifies the value of `new.target` inside of the function.

Together, the `apply` and `construct` traps completely control the behavior of any proxy target function. To mimic the default behavior of a function, you can do this:

```
let target = function() { return 42 },
    proxy = new Proxy(target, {
        apply: function(trapTarget, thisArg, argumentList) {
            return Reflect.apply(trapTarget, thisArg, argumentList);
        },
        construct: function(trapTarget, argumentList) {
            return Reflect.construct(trapTarget, argumentList);
        }
    });

// a proxy with a function as its target looks like a function
console.log(typeof proxy);                    // "function"

console.log(proxy());                         // 42

var instance = new proxy();
console.log(instance instanceof proxy);       // true
console.log(instance instanceof target);      // true
```

This example has a function that returns the number 42. The proxy for that function uses the `apply` and `construct` traps to delegate those behaviors to the `Reflect.apply()` and `Reflect.construct()` methods, respectively. The end result is that the proxy function works exactly like the target function, including identifying itself as a function when `typeof` is used. The proxy is called without `new` to return 42 and then is called with `new` to create an object called `instance`. The `instance` object is considered an instance of both `proxy` and `target` because `instanceof` uses the prototype chain to determine this information. Prototype chain lookup is not affected by this proxy, which is why `proxy` and `target` appear to have the same prototype to the JavaScript engine.

Validating Function Parameters

The `apply` and `construct` traps open up a lot of possibilities for altering the way a function is executed. For instance, suppose you want to validate that all arguments are of a specific type. You can check the arguments in the `apply` trap:

```
// adds together all arguments
function sum(...values) {
    return values.reduce((previous, current) => previous + current, 0);
}

let sumProxy = new Proxy(sum, {
        apply: function(trapTarget, thisArg, argumentList) {

            argumentList.forEach((arg) => {
                if (typeof arg !== "number") {
                    throw new TypeError("All arguments must be numbers.");
                }
            });

            return Reflect.apply(trapTarget, thisArg, argumentList);
        },
        construct: function(trapTarget, argumentList) {
            throw new TypeError("This function can't be called with new.");
        }
    });

console.log(sumProxy(1, 2, 3, 4));          // 10

// throws error
console.log(sumProxy(1, "2", 3, 4));

// also throws error
let result = new sumProxy();
```

This example uses the `apply` trap to ensure that all arguments are numbers. The `sum()` function adds up all of the arguments that are passed. If a non-number value is passed, the function will still attempt the operation, which can cause unexpected results. By wrapping `sum()` inside the `sumProxy()` proxy, this code intercepts function calls and ensures that each argument is a number before allowing the call to proceed. To be safe, the code also uses the `construct` trap to ensure that the function can't be called with `new`.

You can also do the opposite, ensuring that a function must be called with `new` and validating its arguments to be numbers:

```
function Numbers(...values) {
    this.values = values;
}

let NumbersProxy = new Proxy(Numbers, {

        apply: function(trapTarget, thisArg, argumentList) {
            throw new TypeError("This function must be called with new.");
        },

        construct: function(trapTarget, argumentList) {
            argumentList.forEach((arg) => {
                if (typeof arg !== "number") {
                    throw new TypeError("All arguments must be numbers.");
                }
            });

            return Reflect.construct(trapTarget, argumentList);
        }
    });

let instance = new NumbersProxy(1, 2, 3, 4);
console.log(instance.values);               // [1,2,3,4]

// throws error
NumbersProxy(1, 2, 3, 4);
```

Here, the `apply` trap throws an error while the `construct` trap uses the `Reflect.construct()` method to validate input and return a new instance. Of course, you can accomplish the same thing without proxies using `new.target` instead.

Calling Constructors Without new

Chapter 3 introduced the `new.target` metaproperty. To review, `new.target` is a reference to the function on which `new` is called, meaning that you can tell if a function was called using `new` or not by checking the value of `new.target` like this:

```
function Numbers(...values) {

    if (typeof new.target === "undefined") {
        throw new TypeError("This function must be called with new.");
    }

    this.values = values;
}

let instance = new Numbers(1, 2, 3, 4);
console.log(instance.values);               // [1,2,3,4]

// throws error
Numbers(1, 2, 3, 4);
```

This example throws an error when `Numbers` is called without using `new`, which is similar to the example in the "Validating Function Parameters" section but doesn't use a proxy. Writing code like this is much simpler than using a proxy and is preferable if your only goal is to prevent calling the function without `new`. But sometimes you aren't in control of the function whose behavior needs to be modified. In that case, using a proxy makes sense.

Suppose the `Numbers` function is defined in code you can't modify. You know that the code relies on `new.target` and want to avoid that check while still calling the function. The behavior when using `new` is already set, so you can just use the `apply` trap:

```
function Numbers(...values) {

    if (typeof new.target === "undefined") {
        throw new TypeError("This function must be called with new.");
    }

    this.values = values;
}


let NumbersProxy = new Proxy(Numbers, {
        apply: function(trapTarget, thisArg, argumentsList) {
            return Reflect.construct(trapTarget, argumentsList);
        }
    });


let instance = NumbersProxy(1, 2, 3, 4);
console.log(instance.values);               // [1,2,3,4]
```

The `NumbersProxy` function allows you to call `Numbers` without using `new` and have it behave as if `new` were used. To do so, the `apply` trap calls `Reflect.construct()` with the arguments passed into `apply`. The `new.target` inside of `Numbers` is equal to `Numbers` itself, and no error is thrown. While this is a simple example of modifying `new.target`, you can also do so more directly.

Overriding Abstract Base Class Constructors

You can go one step further and specify the third argument to `Reflect.construct()` as the specific value to assign to `new.target`. This is useful when a function is checking `new.target` against a known value, such as when creating an abstract base class constructor (discussed in Chapter 9). In an abstract base class constructor, `new.target` is expected to be something other than the class constructor itself, as in this example:

```
class AbstractNumbers {

    constructor(...values) {
        if (new.target === AbstractNumbers) {
            throw new TypeError("This function must be inherited from.");
        }

        this.values = values;
    }
}

class Numbers extends AbstractNumbers {}

let instance = new Numbers(1, 2, 3, 4);
console.log(instance.values);           // [1,2,3,4]

// throws error
new AbstractNumbers(1, 2, 3, 4);
```

When `new AbstractNumbers()` is called, `new.target` is equal to `AbstractNumbers` and an error is thrown. Calling `new Numbers()` still works because `new.target` is equal to `Numbers`. You can bypass this restriction by manually assigning `new.target` with a proxy:

```
class AbstractNumbers {

    constructor(...values) {
        if (new.target === AbstractNumbers) {
            throw new TypeError("This function must be inherited from.");
        }

        this.values = values;
    }
}

let AbstractNumbersProxy = new Proxy(AbstractNumbers, {
        construct: function(trapTarget, argumentList) {
            return Reflect.construct(trapTarget, argumentList, function() {});
        }
    });


let instance = new AbstractNumbersProxy(1, 2, 3, 4);
console.log(instance.values);               // [1,2,3,4]
```

The `AbstractNumbersProxy` uses the `construct` trap to intercept the call to the `new AbstractNumbersProxy()` method. Then, the `Reflect.construct()` method is called with arguments from the trap and adds an empty function as the third argument. That empty function is used as the value of `new.target` inside of the constructor. Because `new.target` is not equal to `AbstractNumbers`, no error is thrown and the constructor executes completely.

Callable Class Constructors

Chapter 9 explained that class constructors must always be called with `new`. That happens because the internal `[[Call]]` method for class constructors is specified to throw an error. But proxies can intercept calls to the `[[Call]]` method, meaning you can effectively create callable class constructors by using a proxy. For instance, if you want a class constructor to work without using `new`, you can use the `apply` trap to create a new instance. Here's some sample code:

```
class Person {
    constructor(name) {
        this.name = name;
    }
}

let PersonProxy = new Proxy(Person, {
        apply: function(trapTarget, thisArg, argumentList) {
            return new trapTarget(...argumentList);
        }
    });


let me = PersonProxy("Nicholas");
console.log(me.name);                   // "Nicholas"
console.log(me instanceof Person);      // true
console.log(me instanceof PersonProxy); // true
```

The `PersonProxy` object is a proxy of the `Person` class constructor. Class constructors are just functions, so they behave like functions when used in proxies. The `apply` trap overrides the default behavior and instead returns a new instance of `trapTarget` that's equal to `Person`. (I used `trapTarget` in this example to show that you don't need to manually specify the class.) The `argumentList` is passed to `trapTarget` using the spread operator to pass each argument separately. Calling `PersonProxy()`

without using new returns an instance of Person; if you attempt to call Person() without new, the constructor will still throw an error. Creating callable class constructors is something that is only possible using proxies.

Revocable Proxies

Normally, a proxy can't be unbound from its target once the proxy has been created. All of the examples to this point in this chapter have used nonrevocable proxies. But there may be situations when you want to revoke a proxy so that it can no longer be used. You'll find it most helpful to revoke proxies when you want to provide an object through an API for security purposes and maintain the ability to cut off access to some functionality at any point in time.

You can create revocable proxies with the Proxy.revocable() method, which takes the same arguments as the Proxy constructor--a target object and the proxy handler. The return value is an object with the following properties:

1. proxy - the proxy object that can be revoked
2. revoke - the function to call to revoke the proxy

When the revoke() function is called, no further operations can be performed through the proxy. Any attempt to interact with the proxy object in a way that would trigger a proxy trap throws an error. For example:

```
let target = {
    name: "target"
};

let { proxy, revoke } = Proxy.revocable(target, {});

console.log(proxy.name);        // "target"

revoke();

// throws error
console.log(proxy.name);
```

This example creates a revocable proxy. It uses destructuring to assign the proxy and revoke variables to the properties of the same name on the object returned by the Proxy.revocable() method. After that, the proxy object can be used just like a nonrevocable proxy object, so proxy.name returns "target" because it passes through to target.name. Once the revoke() function is called, however, proxy no longer functions. Attempting to access proxy.name throws an error, as will any other operation that would trigger a trap on proxy.

Solving the Array Problem

At the beginning of this chapter, I explained how developers couldn't mimic the behavior of an array accurately in JavaScript prior to ECMAScript 6. Proxies and the reflection API allow you to create an object that behaves in the same manner as the built-in Array type when properties are added and removed. To refresh your memory, here's an example showing the behavior that proxies help to mimick:

```
let colors = ["red", "green", "blue"];

console.log(colors.length);         // 3

colors[3] = "black";

console.log(colors.length);         // 4
console.log(colors[3]);             // "black"

colors.length = 2;

console.log(colors.length);         // 2
console.log(colors[3]);             // undefined
console.log(colors[2]);             // undefined
console.log(colors[1]);             // "green"
```

There are two particularly important behaviors to notice in this example:

1. The `length` property is increased to 4 when `colors[3]` is assigned a value.
2. The last two items in the array are deleted when the `length` property is set to 2.

These two behaviors are the only ones that need to be mimicked to accurately recreate how built-in arrays work. The next few sections describe how to make an object that correctly mimics them.

Detecting Array Indices

Keep in mind that assigning to an integer property key is a special case for arrays, as those are treated differently from non-integer keys. The ECMAScript 6 specification gives these instructions on how to determine if a property key is an array index:

> A String property name `P` is an array index if and only if `ToString(ToUint32(P))` is equal to `P` and `ToUint32(P)` is not equal to 2^32^-1.

This operation can be implemented in JavaScript as follows:

```
function toUint32(value) {
    return Math.floor(Math.abs(Number(value))) % Math.pow(2, 32);
}

function isArrayIndex(key) {
    let numericKey = toUint32(key);
    return String(numericKey) == key && numericKey < (Math.pow(2, 32) - 1);
}
```

The `toUint32()` function converts a given value into an unsigned 32-bit integer using an algorithm described in the specification. The `isArrayIndex()` function first converts the key into a uint32 and then performs the comparisons to determine if the key is an array index or not. With these utility functions available, you can start to implement an object that will mimic a built-in array.

Increasing length when Adding New Elements

You might have noticed that both array behaviors I described rely on the assignment of a property. That means you really only need to use the `set` proxy trap to accomplish both behaviors. To get started, here's an example that implements the first of the two behaviors by incrementing the `length` property when an array index larger than `length - 1` is used:

```
function toUint32(value) {
    return Math.floor(Math.abs(Number(value))) % Math.pow(2, 32);
}

function isArrayIndex(key) {
    let numericKey = toUint32(key);
    return String(numericKey) == key && numericKey < (Math.pow(2, 32) - 1);
}

function createMyArray(length=0) {
    return new Proxy({ length }, {
        set(trapTarget, key, value) {

            let currentLength = Reflect.get(trapTarget, "length");

            // the special case
            if (isArrayIndex(key)) {
                let numericKey = Number(key);
                if (numericKey >= currentLength) {
                    Reflect.set(trapTarget, "length", numericKey + 1);
                }
            }

            // always do this regardless of key type
            return Reflect.set(trapTarget, key, value);
        }
    });
}

let colors = createMyArray(3);
console.log(colors.length);         // 3

colors[0] = "red";
colors[1] = "green";
colors[2] = "blue";

console.log(colors.length);         // 3

colors[3] = "black";

console.log(colors.length);         // 4
console.log(colors[3]);             // "black"
```

This example uses the `set` proxy trap to intercept the setting of an array index. If the key is an array index, then it is converted into a number because keys are always passed as strings. Next, if that numeric value is greater than or equal to the current `length` property, then the `length` property is updated to be one more than the numeric key (setting an item in position 3 means the `length` must be 4). After that, the default behavior for setting a property is used via `Reflect.set()`, since you do want the property to receive the value as specified.

The initial custom array is created by calling `createMyArray()` with a `length` of 3 and the values for those three items are added immediately afterward. The `length` property correctly remains 3 until the value `"black"` is assigned to position 3. At that point, `length` is set to 4.

With the first behavior working, it's time to move on to the second.

Deleting Elements on Reducing length

The first array behavior to mimic is used only when an array index is greater than or equal to the `length` property. The second behavior does the opposite and removes array items when the `length` property is set to a smaller value than it previously contained. That involves not only changing the `length` property, but also deleting all items that might otherwise exist. For instance, if an array with a `length` of 4 has `length` set to 2, the items in positions 2 and 3 are deleted. You can accomplish this inside the `set` proxy trap alongside the first behavior. Here's the previous example again, with an updated `createMyArray` method:

```
function toUint32(value) {
    return Math.floor(Math.abs(Number(value))) % Math.pow(2, 32);
}

function isArrayIndex(key) {
    let numericKey = toUint32(key);
    return String(numericKey) == key && numericKey < (Math.pow(2, 32) - 1);
}

function createMyArray(length=0) {
    return new Proxy({ length }, {
        set(trapTarget, key, value) {

            let currentLength = Reflect.get(trapTarget, "length");
            // the special case
            if (isArrayIndex(key)) {
                let numericKey = Number(key);

                if (numericKey >= currentLength) {
                    Reflect.set(trapTarget, "length", numericKey + 1);
                }
            } else if (key === "length") {

                if (value < currentLength) {
                    for (let index = currentLength - 1; index >= value; index--) {
                        Reflect.deleteProperty(trapTarget, index);
                    }
                }

            }

            // always do this regardless of key type
            return Reflect.set(trapTarget, key, value);
        }
    });
}

let colors = createMyArray(3);
console.log(colors.length);         // 3

colors[0] = "red";
colors[1] = "green";
colors[2] = "blue";
colors[3] = "black";

console.log(colors.length);         // 4

colors.length = 2;

console.log(colors.length);         // 2
console.log(colors[3]);             // undefined
console.log(colors[2]);             // undefined
console.log(colors[1]);             // "green"
console.log(colors[0]);             // "red"
```

The `set` proxy trap in this code checks to see if `key` is `"length"` in order to adjust the rest of the object correctly. When that happens, the current length is first retrieved using `Reflect.get()` and compared against the new value. If the new value is less than the current length, then a `for` loop deletes all properties on the target that should no longer be available. The `for` loop goes backward from the current array length (`currentLength`) and deletes each property until it reaches the new array length (`value`).

This example adds four colors to `colors` and then sets the `length` property to 2. That effectively removes the items in positions 2 and 3, so they now return `undefined` when you attempt to access them. The `length` property is correctly set to 2 and the items in positions 0 and 1 are still accessible.

With both behaviors implemented, you can easily create an object that mimics the behavior of built-in arrays. But doing so with a function isn't as desirable as creating a class to encapsulate this behavior, so the next step is to implement this functionality as a class.

Implementing the MyArray Class

The simplest way to create a class that uses a proxy is to define the class as usual and then return a proxy from the constructor. That way, the object returned when a class is instantiated will be the proxy instead of the instance. (The instance is the value of `this` inside the constructor.) The instance becomes the target of the proxy and the proxy is returned as if it were the instance. The instance will be completely private and you won't be able to access it directly, though you'll be able to access it indirectly through the proxy.

Here's a simple example of returning a proxy from a class constructor:

```
class Thing {
    constructor() {
        return new Proxy(this, {});
    }
}

let myThing = new Thing();
console.log(myThing instanceof Thing);      // true
```

In this example, the class `Thing` returns a proxy from its constructor. The proxy target is `this` and the proxy is returned from the constructor. That means `myThing` is actually a proxy even though it was created by calling the `Thing` constructor. Because proxies pass through their behavior to their targets, `myThing` is still considered an instance of `Thing`, making the proxy completely transparent to anyone using the `Thing` class.

With that in mind, creating a custom array class using a proxy in relatively straightforward. The code is mostly the same as the code in the "Deleting Elements on Reducing Length" section. The same proxy code is used, but this time, it's inside a class constructor. Here's the complete example:

```
function toUint32(value) {
    return Math.floor(Math.abs(Number(value))) % Math.pow(2, 32);
}

function isArrayIndex(key) {
    let numericKey = toUint32(key);
    return String(numericKey) == key && numericKey < (Math.pow(2, 32) - 1);
}

class MyArray {
    constructor(length=0) {
        this.length = length;

        return new Proxy(this, {
            set(trapTarget, key, value) {

                let currentLength = Reflect.get(trapTarget, "length");

                // the special case
                if (isArrayIndex(key)) {
                    let numericKey = Number(key);

                    if (numericKey >= currentLength) {
                        Reflect.set(trapTarget, "length", numericKey + 1);
                    }
                } else if (key === "length") {

                    if (value < currentLength) {
                        for (let index = currentLength - 1; index >= value; index--) {
                            Reflect.deleteProperty(trapTarget, index);
                        }
                    }

                }

                // always do this regardless of key type
                return Reflect.set(trapTarget, key, value);
            }
        });

    }
}


let colors = new MyArray(3);
console.log(colors instanceof MyArray);     // true

console.log(colors.length);         // 3

colors[0] = "red";
colors[1] = "green";
colors[2] = "blue";
colors[3] = "black";

console.log(colors.length);         // 4

colors.length = 2;

console.log(colors.length);         // 2
console.log(colors[3]);             // undefined
console.log(colors[2]);             // undefined
console.log(colors[1]);             // "green"
console.log(colors[0]);             // "red"
```

This code creates a `MyArray` class that returns a proxy from its constructor. The `length` property is added in the constructor (initialized to either the value that is passed in or to a default value of 0) and then a proxy is created and returned. This gives the `colors` variable the appearance of being just an instance of `MyArray` and implements both of the key array behaviors.

Although returning a proxy from a class constructor is easy, it does mean that a new proxy is created for every instance. There is, however, a way to have all instances share one proxy: you can use the proxy as a prototype.

Using a Proxy as a Prototype

Proxies can be used as prototypes, but doing so is a bit more involved than the previous examples in this chapter. When a proxy is a prototype, the proxy traps are only called when the default operation would normally continue on to the prototype, which does limit a proxy's capabilities as a prototype. Consider this example:

```
let target = {};
let newTarget = Object.create(new Proxy(target, {

    // never called
    defineProperty(trapTarget, name, descriptor) {

        // would cause an error if called
        return false;
    }
}));

Object.defineProperty(newTarget, "name", {
    value: "newTarget"
});

console.log(newTarget.name);                    // "newTarget"
console.log(newTarget.hasOwnProperty("name"));  // true
```

The `newTarget` object is created with a proxy as the prototype. Making `target` the proxy target effectively makes `target` the prototype of `newTarget` because the proxy is transparent. Now, proxy traps will only be called if an operation on `newTarget` would pass the operation through to happen on `target`.

The `Object.defineProperty()` method is called on `newTarget` to create an own property called `name`. Defining a property on an object isn't an operation that normally continues to the object's prototype, so the `defineProperty` trap on the proxy is never called and the `name` property is added to `newTarget` as an own property.

While proxies are severely limited when used as prototypes, there are a few traps that are still useful.

Using the `get` Trap on a Prototype

When the internal `[[Get]]` method is called to read a property, the operation looks for own properties first. If an own property with the given name isn't found, then the operation continues to the prototype and looks for a property there. The process continues until there are no further prototypes to check.

Thanks to that process, if you set up a `get` proxy trap, the trap will be called on a prototype whenever an own property of the given name doesn't exist. You can use the `get` trap to prevent unexpected behavior when accessing properties that you can't guarantee will exist. Just create an object that throws an error whenever you try to access a property that doesn't exist:

```
let target = {};
let thing = Object.create(new Proxy(target, {
    get(trapTarget, key, receiver) {
        throw new ReferenceError(`${key} doesn't exist`);
    }
}));

thing.name = "thing";

console.log(thing.name);        // "thing"

// throw an error
let unknown = thing.unknown;
```

In this code, the `thing` object is created with a proxy as its prototype. The `get` trap throws an error when called to indicate that the given key doesn't exist on the `thing` object. When `thing.name` is read, the operation never calls the `get` trap on the prototype because the property exists on `thing`. The `get` trap is called only when the `thing.unknown` property, which doesn't exist, is accessed.

When the last line executes, `unknown` isn't an own property of `thing`, so the operation continues to the prototype. The `get` trap then throws an error. This type of behavior can be very useful in JavaScript, where unknown properties silently return `undefined` instead of throwing an error (as happens in other languages).

It's important to understand that in this example, `trapTarget` and `receiver` are different objects. When a proxy is used as a prototype, the `trapTarget` is the prototype object itself while the `receiver` is the instance object. In this case, that means `trapTarget` is equal to `target` and `receiver` is equal to `thing`. That allows you access both to the original target of the proxy and the object on which the operation is meant to take place.

Using the `set` Trap on a Prototype

The internal `[[Set]]` method also checks for own properties and then continues to the prototype if needed. When you assign a value to an object property, the value is assigned to the own property with the same name if it exists. If no own property with the given name exists, then the operation continues to the prototype. The tricky part is that even though the assignment operation continues to the prototype, assigning a value to that property will create a property on the instance (not the prototype) by default, regardless of whether a property of that name exists on the prototype.

To get a better idea of when the `set` trap will be called on a prototype and when it won't, consider the following example showing the default behavior:

```
let target = {};
let thing = Object.create(new Proxy(target, {
    set(trapTarget, key, value, receiver) {
        return Reflect.set(trapTarget, key, value, receiver);
    }
}));

console.log(thing.hasOwnProperty("name"));      // false

// triggers the `set` proxy trap
thing.name = "thing";

console.log(thing.name);                        // "thing"
console.log(thing.hasOwnProperty("name"));      // true

// does not trigger the `set` proxy trap
thing.name = "boo";

console.log(thing.name);                        // "boo"
```

In this example, `target` starts with no own properties. The `thing` object has a proxy as its prototype that defines a `set` trap to catch the creation of any new properties. When `thing.name` is assigned `"thing"` as its value, the `set` proxy trap is called because `thing` doesn't have an own property called `name`. Inside the `set` trap, `trapTarget` is equal to `target` and `receiver` is equal to `thing`. The operation should ultimately create a new property on `thing`, and fortunately `Reflect.set()` implements this default behavior for you if you pass in `receiver` as the fourth argument.

Once the `name` property is created on `thing`, setting `thing.name` to a different value will no longer call the `set` proxy trap. At that point, `name` is an own property so the `[[Set]]` operation never continues on to the prototype.

Using the `has` Trap on a Prototype

Recall that the `has` trap intercepts the use of the `in` operator on objects. The `in` operator searches first for an object's own property with the given name. If an own property with that name doesn't exist, the operation continues to the prototype. If there's no own property on the prototype, then the search continues through the prototype chain until the own property is found or there are no more prototypes to search.

The `has` trap is therefore only called when the search reaches the proxy object in the prototype chain. When using a proxy as a prototype, that only happens when there's no own property of the given name. For example:

```
let target = {};
let thing = Object.create(new Proxy(target, {
    has(trapTarget, key) {
        return Reflect.has(trapTarget, key);
    }
}));

// triggers the `has` proxy trap
console.log("name" in thing);                   // false

thing.name = "thing";

// does not trigger the `has` proxy trap
console.log("name" in thing);                   // true
```

This code creates a `has` proxy trap on the prototype of `thing`. The `has` trap isn't passed a `receiver` object like the `get` and `set` traps are because searching the prototype happens automatically when the `in` operator is used. Instead, the `has` trap must operate only on `trapTarget`, which is equal to `target`. The first time the `in` operator is used in this example, the `has` trap is called because the property `name` doesn't exist as an own property of `thing`. When `thing.name` is given a value and then the `in` operator is used again, the `has` trap isn't called because the operation stops after finding the own property `name` on `thing`.

The prototype examples to this point have centered around objects created using the `Object.create()` method. But if you want to create a class that has a proxy as a prototype, the process is a bit more involved.

Proxies as Prototypes on Classes

Classes cannot be directly modified to use a proxy as a prototype because their `prototype` property is non-writable. You can, however, use a bit of misdirection to create a class that has a proxy as its prototype by using inheritance. To start, you need to create an ECMAScript 5-style type definition using a constructor function. You can then overwrite the prototype to be a proxy. Here's an example:

```
function NoSuchProperty() {
    // empty
}

NoSuchProperty.prototype = new Proxy({}, {
    get(trapTarget, key, receiver) {
        throw new ReferenceError(`${key} doesn't exist`);
    }
});

let thing = new NoSuchProperty();

// throws error due to `get` proxy trap
let result = thing.name;
```

The `NoSuchProperty` function represents the base from which the class will inherit. There are no restrictions on the `prototype` property of functions, so you can overwrite it with a proxy. The `get` trap is used to throw an error when the property doesn't exist. The `thing` object is created as an instance of `NoSuchProperty` and throws an error when the nonexistent `name` property is accessed.

The next step is to create a class that inherits from `NoSuchProperty`. You can simply use the `extends` syntax discussed in Chapter 9 to introduce the proxy into the class' prototype chain, like this:

```
function NoSuchProperty() {
    // empty
}

NoSuchProperty.prototype = new Proxy({}, {
    get(trapTarget, key, receiver) {
        throw new ReferenceError(`${key} doesn't exist`);
    }
});

class Square extends NoSuchProperty {
    constructor(length, width) {
        super();
        this.length = length;
        this.width = width;
    }
}

let shape = new Square(2, 6);

let area1 = shape.length * shape.width;
console.log(area1);                         // 12
```

```
// throws an error because "wdth" doesn't exist
let area2 = shape.length * shape.wdth;
```

The `Square` class inherits from `NoSuchProperty` so the proxy is in the `Square` class' prototype chain. The `shape` object is then created as a new instance of `Square` and has two own properties: `length` and `width`. Reading the values of those properties succeeds because the `get` proxy trap is never called. Only when a property that doesn't exist on `shape` is accessed (`shape.wdth`, an obvious typo) does the `get` proxy trap trigger and throw an error.

That proves the proxy is in the prototype chain of `shape`, but it might not be obvious that the proxy is not the direct prototype of `shape`. In fact, the proxy is a couple of steps up the prototype chain from `shape`. You can see this more clearly by slightly altering the preceding example:

```
function NoSuchProperty() {
    // empty
}

// store a reference to the proxy that will be the prototype
let proxy = new Proxy({}, {
    get(trapTarget, key, receiver) {
        throw new ReferenceError(`${key} doesn't exist`);
    }
});

NoSuchProperty.prototype = proxy;

class Square extends NoSuchProperty {
    constructor(length, width) {
        super();
        this.length = length;
        this.width = width;
    }
}

let shape = new Square(2, 6);

let shapeProto = Object.getPrototypeOf(shape);

console.log(shapeProto === proxy);                  // false

let secondLevelProto = Object.getPrototypeOf(shapeProto);

console.log(secondLevelProto === proxy);            // true
```

This version of the code stores the proxy in a variable called `proxy` so it's easy to identify later. The prototype of `shape` is `Square.prototype`, which is not a proxy. But the prototype of `Square.prototype` is the proxy that was inherited from `NoSuchProperty`.

The inheritance adds another step in the prototype chain, and that matters because operations that might result in calling the `get` trap on `proxy` need to go through one extra step before getting there. If there's a property on `Square.prototype`, then that will prevent the `get` proxy trap from being called, as in this example:

```
function NoSuchProperty() {
    // empty
}

NoSuchProperty.prototype = new Proxy({}, {
    get(trapTarget, key, receiver) {
        throw new ReferenceError(`${key} doesn't exist`);
    }
});

class Square extends NoSuchProperty {
    constructor(length, width) {
        super();
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }
}

let shape = new Square(2, 6);

let area1 = shape.length * shape.width;
console.log(area1);                         // 12

let area2 = shape.getArea();
console.log(area2);                         // 12

// throws an error because "wdth" doesn't exist
let area3 = shape.length * shape.wdth;
```

Here, the `Square` class has a `getArea()` method. The `getArea()` method is automatically added to `Square.prototype` so when `shape.getArea()` is called, the search for the method `getArea()` starts on the `shape` instance and then proceeds to its prototype. Because `getArea()` is found on the prototype, the search stops and the proxy is never called. That is actually the behavior you want in this situation, as you wouldn't want to incorrectly throw an error when `getArea()` was called.

Even though it takes a little bit of extra code to create a class with a proxy in its prototype chain, it can be worth the effort if you need such functionality.

Summary

Prior to ECMAScript 6, certain objects (such as arrays) displayed nonstandard behavior that developers couldn't replicate. Proxies change that. They let you define your own nonstandard behavior for several low-level JavaScript operations, so you can replicate all behaviors of built-in JavaScript objects through proxy traps. These traps are called behind the scenes when various operations take place, like a use of the `in` operator.

A reflection API was also introduced in ECMAScript 6 to allow developers to implement the default behavior for each proxy trap. Each proxy trap has a corresponding method of the same name on the `Reflect` object, another ECMAScript 6 addition. Using a combination of proxy traps and reflection API methods, it's possible to filter some operations to behave differently only in certain conditions while defaulting to the built-in behavior.

Revocable proxies are a special proxies that can be effectively disabled by using a `revoke()` function. The `revoke()` function terminates all functionality on the proxy, so any attempt to interact with the proxy's properties throws an error after `revoke()` is called. Revocable proxies are important for application security where third-party developers may need access to certain objects for a specified amount of time.

While using proxies directly is the most powerful use case, you can also use a proxy as the prototype for another object. In that case, you are severely limited in the number of proxy traps you can effectively use. Only the `get`, `set`, and `has` proxy traps will ever be called on a proxy when it's used as a prototype, making the set of use cases much smaller.