Object.defineProperty

Learn how to enforce immutability through `Object.defineProperty`. Learn how it ties into functional programming.

`Object.defineProperty`

This function allows us to create properties on an object, but specify some rules that the engine will enforce. For example, we can make it so that the property can't be changed, deleted, or printed in a loop.

Usage

`Object.defineProperty` takes in three arguments:

- The object on which we want to define a property
- The name of the property we want to define
- An object containing both the value we'd like to give our property and the rules we would like to enforce on our property

`value`

There are four optional items we can place in our rules object. One of them is `value` and simply allows us to give our property a value.

`writable`, `configurable`, `enumerable`

The remaining three are booleans which, by default, are set to `false`. They remove some piece of functionality from our object.

When discussing them, we'll first manually set them to `true`. As we discuss each one, we'll make it `false` in the examples. Initially setting them all to `true` will help us understand them one by one.

`value`

To provide the value for the property we're adding to our object, we place a `value` property on our options object. It defaults to `undefined` if it's not provided.

```
const obj = {};

Object.defineProperty(obj, 'testValue', {
    value: 17,
    writable: true,
    configurable: true,
    enumerable: true
});

console.log(obj.testValue); // -> 17
```

The previous codeblock is nothing special. It's equivalent to:

```
const obj = {};
obj.test = 17;
console.log(obj.test); // -> 17
```

Let's start diving into the properties that will actually change this object's behavior.

`writable`

This option determines if the property we're assigning can be changed. If `true`, the property will function normally and can be overwritten.

If we attempt to change the property when `writable` is `false`, the operation will fail silently. We can, however, still delete the property.

```
const obj = {};

Object.defineProperty(obj, 'testValue', {
    value: 17,
    writable: false,
    configurable: true,
    enumerable: true
});

obj.testValue = 22; // fails silently
console.log(obj.testValue); // -> 17

delete obj.testValue; // deletion still works
console.log(obj.testValue); // -> undefined
```

`enumerable`

The `enumerable` property on our options object will determine if the property will show up when we iterate through the object. If `false`, it won’t show up in a for-in loop. It is still accessible directly.

```
const obj = { x: 100 };

Object.defineProperty(obj, 'testValue', {
    value: 17,
    writable: true,
    configurable: true,
    enumerable: false
});

// enumerable is set to false, so
// `testValue` won't show up here
for(const key in obj) {
    console.log(key); // -> x
}

console.log(obj.testValue); // -> 17
```

`configurable`

The `configurable` property does two things.

Deletion

First, it determines whether or not the property we’re adding can be deleted.

```
const obj = {};

Object.defineProperty(obj, 'testValue', {
    value: 17,
    writable: true,
    configurable: false,
    enumerable: true
});

delete obj.testValue; // fails silently
console.log(obj.testValue); // -> 17
```

Configuration

Second, it determines whether or not we can later change our configuration of the property. When we use `Object.defineProperty`, as long as `configurable` is `true`, we can later decide to undo our rules.

If we use `Object.definePropery` a second time with the same property argument passed in, it'll simply override our previous rules. In the next example, we set writable to `false` and watch an assignment fail. We then set it to `true` and watch the next assignment succeed.

```
const obj = {};

Object.defineProperty(obj, 'testValue', {
    value: 17,
    writable: false,
    configurable: true,
    enumerable: true
});

obj.testValue = 22; // fails silently
console.log(obj.testValue); // -> 17

Object.defineProperty(obj, 'testValue', {
    value: 17,
    writable: true,
    configurable: true,
    enumerable: true
});

obj.testValue = 35; // assignment works, as
// previous rule was overridden

console.log(obj.testValue); // -> 35
```

If `configurable` is initially to `false`, our attempt to reuse `Object.defineProperty` above will result in an error.

```
const obj = {};

Object.defineProperty(obj, 'testValue', {
    value: 17,
    writable: false,
    configurable: false,
    enumerable: true
});

obj.testValue = 22; // fails silently

console.log(obj.testValue); // -> 17

Object.defineProperty(obj, 'testValue', {
    value: 17,
    writable: true,
    configurable: false,
    enumerable: true
}); // -> TypeError: Cannot redefine property: testValue
```

Once we define a property and set `configurable` to false, there is only one configuration change allowed to the property. We can still change `writable` from `true` to `false`. All other attempts to change configuration will result in an error.

In the following code block, `configurable` is set to `false` and we then try to re-use `Object.defineProperty`. It still works, however, as all we're doing is changing `writable` from `true` to `false`.

```
const obj = {};

Object.defineProperty(obj, 'testValue', {
    value: 17,
    writable: true,
    configurable: false,
    enumerable: true
});

obj.testValue = 22; // assignment works
console.log(obj.testValue); // -> 22

Object.defineProperty(obj, 'testValue', {
    writable: false,
    configurable: false,
    enumerable: true
}); // No error

obj.testValue = 35; // fails silently
console.log(obj.testValue); // -> 22
```

Play around with the code block above. Flipping the booleans on lines 15 and 16 will result in an error.

Once both `configurable` and `writable` are set to `false`, all attemps to change configuration will result in an error.

Conclusion

Using `Object.defineProperty` and setting both `writable` and `configurable` to `false`, we can essentially freeze an object's property. It can't be deleted or changed. This allows us to explicitly implement functional programming by making our data immutable.

Options Summary

- `enumerable`: Determines whether or not the property will show up in iterations.
- `writable`: Determines whether or not the property can be reassigned. It can still be delelted.
- `configurable`: Determines whether or not the property can be deleted, and whether or not the property's confiuration can be changed. If `false`, the only configuration change that can later occur is a change of `writable` from `true` to `false`.
- By default, all of these are false.