5 - Object Properties Configuration

We have seen properties as a simple "key-value" pair, but they are actually more flexible and powerful.

Property flags

Objects beside a value has three special attributes (flags):

- **writable** - if true, value can be changed;
- **enumerable** - if true, than listed in loops, otherwise not listed;
- **configurable** - if ture, property can be deleted and these attributes can be modified, otherwise not.

We didn't see them yet, because generally they do not show up. When we create a property "the usual way", all of them are true. But we also can change them anytime. To get those flags: `Object.getOwnPropertyDescriptor` allows to query the full information about a property.

```
let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);
```

The returned value is a so-called "property descriptor" object: it contains the value and all the flags.

```
let user = {
  name: "John"
};

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/* property descriptor:
{
  "value": "John",
  "writable": true,
  "enumerable": true,
  "configurable": true
}
*/
```

To change those flags, we can use `Object.defineProperty`:

```
Object.defineProperty(obj, propertyName, descriptor)
```

If the property exists, defineProperty updates its flags. Otherwise, it creates the property with the given value and flags; in that case, if a flag is not supplied, it is assumed false.

For instance, here a property name is created with all falsy flags:

```
let user = {};

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

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
  "value": "John",
  "writable": false,
  "enumerable": false,
  "configurable": false
}
 */
```

Compare it with “normally created” user.name above: now all flags are falsy. If that’s not what we want then we’d better set them to true in descriptor.

Now let’s see effects of the flags by example.

Non-writable

```
let user = {
  name: "John"
};

Object.defineProperty(user, "name", {
  writable: false
});

user.name = "Pete"; // Error: Cannot assign to read only property 'name'

// Same example but creating property from scratch:
let user = { };

Object.defineProperty(user, "name", {
  value: "John",
  // for new properties we need to explicitly list what's true
  enumerable: true,
  configurable: true
});
```

```
alert(user.name); // John
user.name = "Pete"; // Error
```

Now no one can change the name of our user, unless they apply their own defineProperty to override ours.

Non-enumerable

Now let's add a custom toString to user.

Normally, a built-in toString for objects is non-enumerable, it does not show up in for..in. But if we add a toString of our own, then by default it shows up in for..in, like this:

```
let user = {
  name: "John",
  toString() {
    return this.name;
  }
};

// By default, both our properties are listed:
for (let key in user) alert(key); // name, toString
```

If we don't like it, then we can set enumerable:false. Then it won't appear in a for..in loop, just like the built-in one:

```
let user = {
  name: "John",
  toString() {
    return this.name;
  }
};

Object.defineProperty(user, "toString", {
  enumerable: false
});

// Now our toString disappears:
for (let key in user) alert(key); // name

// Non enumerable properties are also excluded from Object.keys:

alert(Object.keys(user)); // name
```

Non-configurable

The non-configurable flag (configurable:false) is sometimes preset for built-in objects and properties.

A non-configurable property can't be deleted, its attributes can't be modified.

For instance, Math.PI is non-writable, non-enumerable and non-configurable:

```
let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
  "value": 3.141592653589793,
  "writable": false,
  "enumerable": false,
  "configurable": false
}
*/

Math.PI = 3; // Error, because it has writable: false

// delete Math.PI won't work either

// We can't even change the value of Math.PI or overwrite it:

// Error, because of configurable: false
Object.defineProperty(Math, "PI", { writable: true });
```

There's absolutely nothing we can do with Math.PI.

Making a property non-configurable is a one-way road. We cannot change it back with defineProperty.

Please note: configurable: false prevents changes of property flags and its deletion, while allowing to change its value.

Here user.name is non-configurable, but we can still change it (as it's writable):

```
let user = {
  name: "John"
};

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

user.name = "Pete"; // works fine
delete user.name; // Error

// Here we will make usern.name a forever sealed constant, like Math.PI:

let user = {
  name: "John"
```

```
};

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

// won't be able to change user.name or its flags
// all this won't work:
user.name = "Pete";
delete user.name;
Object.defineProperty(user, "name", { value: "Pete" });
```

We can change writable: true to false for a non-configurable property, thus preventing its value modification (to add another layer of protection). Not the other way around though.

Object.defineProperties

There is a method that allows to define many properties at once:

```
Object.defineProperties(obj, {
  prop1: descriptor1,
  prop2: descriptor2
  // ...
});

// This way we can set many properties at once

// To get all property descriptors at once, we can use the method:

Object.getOwnPropertyDescriptors(obj)

// Together with Object.defineProperties this can be used as a flags-aware
way of cloning an object:

let clone = Object.defineProperties({},
Object.getOwnPropertyDescriptors(obj));

// The traditional way of cloning an object would be:
for (let key in user) {
  clone[key] = user[key]
}
```

```
// But this does not copy flags, so if we want a deeper clone we can use these methods instead.
```

Another difference is that for..in ignores symbolic and non-enumerable properties, but Object.getOwnPropertyDescriptors returns all property descriptors including symbolic and non-enumerable ones.

There are also methods that limit access to the whole object:

Property descriptors work at the level of individual properties.

Sealing an object globally

There are also methods that limit access to the whole object (these are rarely used in practice):

`Object.preventExtensions(obj)`
Forbids the addition of new properties to the object.
`Object.seal(obj)`
Forbids adding/removing of properties. Sets configurable: false for all existing properties.
`Object.freeze(obj)`
Forbids adding/removing/changing of properties. Sets configurable: false, writable: false for all existing properties.
And also there are tests for them:

`Object.isExtensible(obj)`
Returns false if adding properties is forbidden, otherwise true.
`Object.isSealed(obj)`
Returns true if adding/removing properties is forbidden, and all existing properties have configurable: false.
`Object.isFrozen(obj)`
Returns true if adding/removing/changing properties is forbidden, and all current properties are configurable: false, writable: false.

Property getters and setters

There are two types of properties: data properties that we have used until now, and **accessor properties**. They are functions that execute on getting and setting a value, but look like regular properties to an external code.

Getters and setters

Accessor properties are represented by **getter** and **setter** methods. In an object literal they are denoted by get and set:

```
let obj = {
  get propName() {
    // getter, the code executed on getting obj.propName
  },

  set propName(value) {
    // setter, the code executed on setting obj.propName = value
  }
};

// The getter works when obj.propName is read, the setter when it is assigned

let user = {
  name: "John",
  surname: "Smith"
};
```

Now we want to add a fullName property, that should be "John Smith". Of course, we don't want to copy-paste existing information, so we can implement it as an accessor:

```
let user = {
  name: "John",
  surname: "Smith",

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

alert(user.fullName); // John Smith
```

From the outside, an accessor property looks like a regular one. That's the idea of accessor properties. We don't call user.fullName as a function, we read it normally: the getter runs behind the scenes. As of

now, fullName has only a getter. If we attempt to assign user.fullName=, there will be an error.

```
let user = {
  get fullName() {
    return `...`;
  }
};

user.fullName = "Test"; // Error (property has only a getter)

// Let's fix it by adding a setter:
let user = {
  name: "John",
  surname: "Smith",

  get fullName() {
    return `${this.name} ${this.surname}`;
  },

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  }
};

// set fullName is executed with the given value.
user.fullName = "Alice Cooper";

alert(user.name); // Alice
alert(user.surname); // Cooper
```

As the result, we have a “virtual” property fullName. It is readable and writable.

Accessor descriptors

Descriptors for accessor properties are different from those for data properties.
For accessor properties, there is no value or writable, but instead there are get and set functions.

That is, an accessor descriptor may have:

- **get** – a function without arguments, that works when a property is read,
- **set** – a function with one argument, that is called when the property is set,
- **enumerable** – same as for data properties,
- **configurable** – same as for data properties.

To create an accessor fullName with defineProperty, we can pass a descriptor with **get** and **set**:

```
let user = {
  name: "John",
  surname: "Smith"
};

Object.defineProperty(user, 'fullName', {
  get() {
    return `${this.name} ${this.surname}`;
  },

  set(value) {
    [this.name, this.surname] = value.split(" ");
  }
});

alert(user.fullName); // John Smith

for(let key in user) alert(key); // name, surname
```

Please note that a property can be either an accessor (has get/set methods) or a data property (has a value), not both.
If we try to supply both get and value in the same descriptor, there will be an error:

```
// Error: Invalid property descriptor.
Object.defineProperty({}, 'prop', {
  get() {
    return 1
  },

  value: 2
});
```

Smarter getters and setters

These can be used as wrappers over real property values to gain more control over operations with them.
For instance, if we want to forbid too short names for user, we can have a setter name and keep the value in a separate property _name:

```
let user = {
  get name() {
    return this._name;
  },
```

```
  set name(value) {
    if (value.length < 4) {
      alert("Name is too short, need at least 4 characters");
      return;
    }
    this._name = value;
  }
};

user.name = "Pete";
alert(user.name); // Pete

user.name = ""; // Name is too short...
```

So, the name is stored in _name property, and the access is done via getter and setter.
Technically, external code is able to access the name directly by using user.*name. **But there is a widely known convention that properties starting with an underscore "*" are internal and should not be touched from outside the object.**

Used for compatibility

One of the great uses of accessors is that they allow to take control over a “regular” data property at any moment by replacing it with a getter and a setter and tweak its behavior.

Imagine we started implementing user objects using data properties name and age:

```
function User(name, age) {
  this.name = name;
  this.age = age;
}

let john = new User("John", 25);

alert( john.age ); // 25
```

But sooner or later, things may change. Instead of age we may decide to store birthday, because it’s more precise and convenient:

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;
}

let john = new User("John", new Date(1992, 6, 1));
```

We can try to find all such places and fix them, but that takes time and can be hard to do if that code is used by many other people. And besides, age is a nice thing to have in user, right? Adding a getter for age solves the problem:

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;

  // age is calculated from the current date and birthday
  Object.defineProperty(this, "age", {
    get() {
      let todayYear = new Date().getFullYear();
      return todayYear - this.birthday.getFullYear();
    }
  });
}

let john = new User("John", new Date(1992, 6, 1));

alert( john.birthday ); // birthday is available
alert( john.age );      // ...as well as the age
```