9th January 2020

# Property getters and setters

There are two kinds of properties.

The first kind is *data properties*. We already know how to work with them. All properties that we've been using until now were data properties.

The second type of properties is something new. It's *accessor properties*. They are essentially functions that work 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.

For instance, we have a `user` object with `name` and `surname` :

```
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 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 for `user.fullName`:

```
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.

For instance, 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/setters

Getters/setters 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.

## Using 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));
```

Now what to do with the old code that still uses `age` property?

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?

Let’s keep it.

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
```

Now the old code works too and we've got a nice additional property.

Previous lesson Next lesson

Share Tutorial map

## Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)