Object Properties

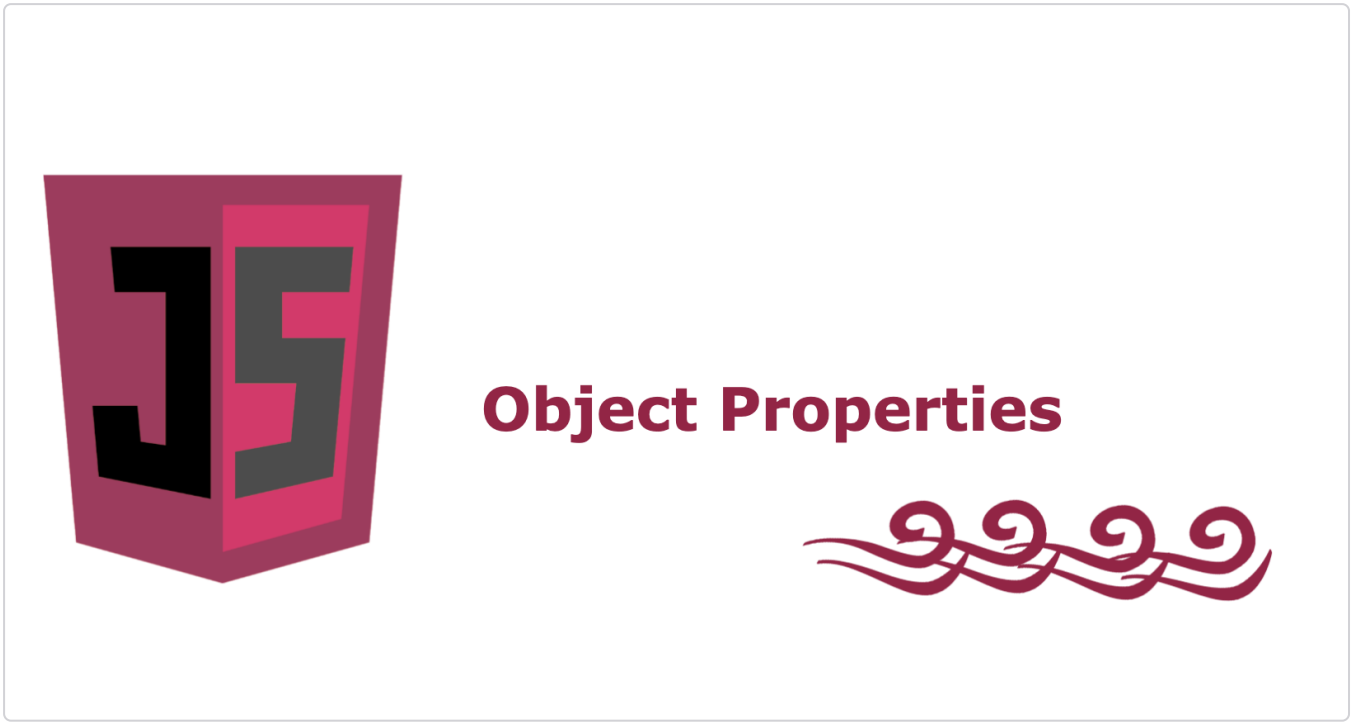
In this lesson, we explore some object properties in detail.
Let's begin!

WE'LL COVER THE FOLLOWING

Object properties may hold any kind of value, including primitive values and object instances. The ECMAScript specification defines two kinds of properties: data properties and accessor properties. Data properties store a data value at a single location. You can directly read or write the location where the data value is stored.

In contrast, accessor properties do not contain a data value, but they are a combination of two methods. The getter method is responsible for obtaining the value of the property; the setter method takes care to write the property value. Internally, JavaScript assigns attributes to properties and those attributes determine the behavior of the property, including the type of operations available on that specific property.

Both data and accessor properties have two common attributes, `Configurable` and `Enumerable`. `Configurable` indicates whether the property may be redefined by removing the property, changing the property's attributes, or changing the property into an accessor property. Enumerable designates whether the property will be returned in a `for-in` loop.



> 📜**NOTE:** By default, `Configurable` and `Enumerable` are true for all properties that are defined directly on an object.

Data properties have two other specific attributes, `Value` and `Writeable`. While `Value` holds the actual data value for the property (this is undefined by default), `Writeable` indicates whether the property's value can be changed (and it is true by default).

Accessor properties have a `Get` and a `Set` attribute that are the function to invoke when the property is read from and the function to call when the property is written to. Both attributes have undefined as their default values.

To query property attributes, you can use the `Object.getOwnPropertyDescriptor()` method. You can create new properties or change existing ones with the `Object.defineProperty()` method or set up multiple properties with `Object.defineProperties()`.

Listing 8-11 demonstrates setting up a new read-only data property.

Listing 8-11: Setting up a new read-only data property

```
<!DOCTYPE html>
```

```
<html>
<head>
  <title>Data property attributes</title>
  <script>
    var car = new Object();
    Object.defineProperty(car, "type",
      {
        writeable: false,
        value: "FR-V"
      });

    console.log(car.type); // FR-V
    // This has no effect
    car.type = "Other";
    console.log(car.type); // FR-V

    var descr = Object.getOwnPropertyDescriptor(
      car, "type");
    console.log(descr.configurable); // false
    console.log(descr.enumerable);   // false
    console.log(descr.writable);     // false
    console.log(descr.value);        // FR-V
  </script>
</head>
<body>
  Listing 8-11: View the console output
</body>
</html>
```

The `Object.defineProperty()` method accepts three arguments. The first is the object instance, the second one names the property, and the third one provides an object that describes the property attributes.

This listing defines a **read-only attribute**, so the third argument specifies the writeable and value attributes. When you try to set a read-only attribute, the JavaScript engine *does not* raise an error, it simply ignores the modification request.

The `Object.getOwnPropertyDescriptor()` method accepts two arguments, the object instance and the property name. It retrieves a descriptor object with property names according to the attributes. As you can see from the output of the example, the unset configurable and enumerable properties are initialized to `false`.

Accessor properties can be great for setting up calculated properties and for values that need checking before setting them.

Listing 8-12 demonstrates how to define and use them.

Listing 8-12: Defining and using accessor

properties

```
<!DOCTYPE html>
<html>
<head>
  <title>Define accessor properties</title>
  <script>
    var person = {
      firstName: "John",
      _lastName: "Doe"
    };
    Object.defineProperty(person, "fullName",
      {
        writeable: false,
        get: function () {
          return this.lastName + ", "
            + this.firstName;
        }
      });
    Object.defineProperty(person, "lastName",
      {
        writeable: true,
        get: function () {
          return this._lastName;
        },
        set: function (value) {
          if (value != "") {
            this._lastName = value.toString();
          }
        }
      });

    console.log(person.fullName); // Doe, John
    person.lastName = "";
    console.log(person.fullName); // Doe, John
    person.lastName = "Smith";
    console.log(person.fullName); // Smith, John
  </script>
</head>
<body>
  Listing 8-12: View the console output
</body>
</html>
```

This code adds two properties to the person object using `Object.defineProperty()` and `fullName`, which is a calculated property, and `lastName`, which accepts only non-empty strings. As you see, the last name of the person is stored in the `_lastName` property, and it is set through the setter function of `lastName`.

The console output, indicated in comments appended to the `console.log()` calls, indicates that `fullName` works as expected and that `lastName` checks for non-empty strings.

With `Object.defineProperties()` you can set up one or more properties in a single step, as Listing 8-13 shows.

This listing sets up the person object just as Listing 8-12 does.

Listing 8-13: Using `Object.defineProperties()`

```
<!DOCTYPE html>
<html>
<head>
  <title>Using Object.defineProperties()</title>
  <script>
    var person = {
      firstName: "John",
      _lastName: "Doe"
    };
    Object.defineProperties(person,
      {
        fullName:
        {
          writeable: false,
          get: function () {
            return this.lastName + ", "
              + this.firstName;
          }
        },
        lastName:
        {
          writeable: true,
          get: function () {
            return this._lastName;
          },
          set: function (value) {
            if (value != "") {
              this._lastName = value.toString();
            }
          }
        }
      });

    console.log(person.fullName); // Doe, John
    person.lastName = "";
    console.log(person.fullName); // Doe, John
    person.lastName = "Smith";
    console.log(person.fullName); // Smith, John
  </script>
</head>
<body>
  Listing 8-13: View the console output
</body>
</html>
```

The first argument of `Object.defineProperties()` is the object to define the

multiple properties for. The second argument is an object that specifies a set

of properties (`fullName` and `lastName` in the example above) and sets each property to a descriptor object similarly to the `Object.defineProperty()` call.

In the *next lesson*, we'll see how to remove some of the object properties that we discussed above.

Stay tuned! :)