mdn web docs

Classes in JavaScript

In the last article, we introduced some basic concepts of object-oriented programming (OOP), and discussed an example where we used OOP principles to model professors and students in a school.

We also talked about how it's possible to use prototypes and constructors to implement a model like this, and that JavaScript also provides features that map more closely to classical OOP concepts.

In this article, we'll go through these features. It's worth keeping in mind that the features described here are not a new way of combining objects: under the hood, they still use prototypes. They're just a way to make it easier to set up a prototype chain.

Prerequisites:	A basic understanding of HTML and CSS, familiarity with JavaScript basics (see First steps and Building blocks) and OOJS basics (see Introduction to objects, Object prototypes, and Object-oriented programming).
Objective:	To understand how to use the features JavaScript provides to implement "classical" object-oriented

	programs.

Classes and constructors

You can declare a class using the `class` keyword. Here's a class declaration for our `Person` from the previous article:

JS

```js
class Person {
  name;

  constructor(name) {
    this.name = name;
  }

  introduceSelf() {
    console.log(`Hi! I'm ${this.name}`);
  }
}
```

This declares a class called `Person`, with:

- a `name` property.
- a constructor that takes a `name` parameter that is used to initialize the new object's `name` property
- an `introduceSelf()` method that can refer to the object's properties using `this`.

The `name;` declaration is optional: you could omit it, and the line `this.name = name;` in the constructor will create the `name` property before initializing it. However, listing properties explicitly in the class declaration might make it easier for people reading your code to see which properties are part of this class.

You could also initialize the property to a default value when you declare it, with a line like `name = '';`.

The constructor is defined using the `constructor` keyword. Just like a constructor outside a class definition, it will:

- create a new object
- bind `this` to the new object, so you can refer to `this` in your constructor code
- run the code in the constructor
- return the new object.

Given the class declaration code above, you can create and use a new `Person` instance like this:

JS

```js
const giles = new Person("Giles");

giles.introduceSelf(); // Hi! I'm Giles
```

Note that we call the constructor using the name of the class, `Person` in this example.

Omitting constructors

If you don't need to do any special initialization, you can omit the constructor, and a default constructor will be generated for you:

JS

```js
class Animal {
  sleep() {
    console.log("zzzzzzz");
  }
}

const spot = new Animal();

spot.sleep(); // 'zzzzzzz'
```

Inheritance

Given our `Person` class above, let's define the `Professor` subclass.

JS

```js
class Professor extends Person {
  teaches;

  constructor(name, teaches) {
    super(name);
    this.teaches = teaches;
  }

  introduceSelf() {
    console.log(
      `My name is ${this.name}, and I will be your ${this.teaches}
professor.`,
    );
  }

  grade(paper) {
    const grade = Math.floor(Math.random() * (5 - 1) + 1);
    console.log(grade);
  }
}
```

We use the `extends` keyword to say that this class inherits from another class.

The `Professor` class adds a new property `teaches`, so we declare that.

Since we want to set `teaches` when a new `Professor` is created, we define a constructor, which takes the `name` and `teaches` as arguments. The first thing this constructor does is call the superclass constructor using `super()`, passing up the `name` parameter. The superclass constructor takes care of setting `name`. After that, the `Professor` constructor sets the `teaches` property.

Note: If a subclass has any of its own initialization to do, it **must** first call the superclass constructor using `super()`, passing up any parameters that the superclass constructor is expecting.

We've also overridden the `introduceSelf()` method from the superclass, and added a new method `grade()`, to grade a paper (our professor isn't very good, and just assigns random grades to papers).

With this declaration we can now create and use professors:

JS

```js
const walsh = new Professor("Walsh", "Psychology");
walsh.introduceSelf(); // 'My name is Walsh, and I will be your Psychology professor'

walsh.grade("my paper"); // some random grade
```

Encapsulation

Finally, let's see how to implement encapsulation in JavaScript. In the last article we discussed how we would like to make the `year` property of `Student` private, so we could change the rules about archery classes without breaking any code that uses the `Student` class.

Here's a declaration of the `Student` class that does just that:

JS

```js
class Student extends Person {
  #year;
  constructor(name, year) {
    super(name);
    this.#year = year;
  }

  introduceSelf() {
    console.log(`Hi! I'm ${this.name}, and I'm in year ${this.#year}.`);
  }

  canStudyArchery() {
    return this.#year > 1;
  }
}
```

In this class declaration, `#year` is a private data property. We can construct a `Student` object, and it can use `#year` internally, but if code outside the object tries to access `#year` the browser throws an error:

JS

```
const summers = new Student("Summers", 2);

summers.introduceSelf(); // Hi! I'm Summers, and I'm in year 2.
summers.canStudyArchery(); // true

summers.#year; // SyntaxError
```

Note: Code run in the Chrome console can access private properties outside the class. This is a DevTools-only relaxation of the JavaScript syntax restriction.

Private data properties must be declared in the class declaration, and their names start with `#`.

Private methods

You can have private methods as well as private data properties. Just like private data properties, their names start with `#`, and they can only be called by the object's own methods:

JS

```js
class Example {
  somePublicMethod() {
    this.#somePrivateMethod();
  }

  #somePrivateMethod() {
    console.log("You called me?");
  }
}

const myExample = new Example();

myExample.somePublicMethod(); // 'You called me?'

myExample.#somePrivateMethod(); // SyntaxError
```

Test your skills!

You've reached the end of this article, but can you remember the most important information? You can find some further tests to verify that you've retained this information before you move on — see Test your skills: Object-oriented JavaScript.

Summary

In this article, we've gone through the main tools available in JavaScript for writing object-oriented programs. We haven't covered everything here, but this should be enough to get you started. Our article on Classes is a good place to learn more.