An Introduction to JavaScript Classes

What are Classes?

A *class* is a template function that makes objects with initial values for variables and methods. They're useful because they allow us to make a lot of similar objects with slight differences. What they really do is create a function with the same name as the one provided to it, with function code that comes from the provided constructor method. And then it stores class methods in that function's prototype.

Once it creates an object, that object has access to the prototype that connects all of the objects it creates. They're all plugged into the same prototype database! We could technically do all of this without actually using a class, but there are differences between "normally" setting up a group of functions and an actual class. Class-made functions have an internal property **[[IsClassConstructor]]: true** that identifies them from normal functions. JavaScript will check for that property at some points, such as when trying to call a function (class functions must be called with **new**). And stringified class constructors (in most JavaScript engines) will start with "class."

Additionally, class methods are intentionally non-enumerable because we don't typically want an object's class methods when we **for…in** loop over it. Finally, all of the code in a class is in strict mode automatically. Here's the syntax for a class:

```
1  class MyClass {
2      constructor() {console.log}
3      method1() {console.log}
4      method2() {console.log}
5      method3() {console.log}
6      }
```

Please note that we are only creating the class here, not actually running it yet. First, we create and name the class. Then we have a *constructor function* within, alongside however many methods we want. The constructor function is a type of function that initializes an object before that object is created. Essentially, it holds information (including the initial value of **this**) that is passed to and modified by the object we create.

After you write the class, you can use the **new** operator to execute it, creating a new object with the listed methods. Any constructor methods that you provide will be called automatically by **new**. Here's an example:

```
class User {
    constructor(name) {
        this.name = name;
    }

    sayHi() {
        alert(this.name);
    }
}

    // Usage
    let user = new User("John");
    user.sayHi();
```

When we use **new User(“John”)**, a new object is created and the constructor runs. Take note that classes do not have commas between methods.

Class Expression

Like functions, classes can be defined inside an expression, passed around, returned, and assigned. Like a Named Function Expression, class expressions can be named. This name will be only visible inside the class. Here's an example:

```
let User = class MyClass {
    sayHi() {
        alert(MyClass);
    }
};

new User().sayHi(); // Works
alert(MyClass); // Error
```

We can also dynamically make classes like so:

```
function makeClass() {
    return class {
        sayHi() {
            alert(phrase);
        }
    };
}

let user = makeClass("Hello");
new user().sayHi(); // Hello
```

Getters/Setters

Classes can use getters/setters as well as computed properties, just like literal objects. Declaring them stores them in the class's prototype. Here's an example:

```
class User {
    constructor(name) {
        this.name = name;
    }

    get name() {
        return this._name;
    }

    set name(value) {
        if (value.length > 4) {
            alert("Name is too short.");
            return;
        }
        this.name = value;
    }
}

let user = new User("John");
alert(user.name); // John

user = new User("Po"); // Name is too short
```

In this example, the getter function returns the **name** variable. The setter function then evaluates the length of that variable when a new user is created to ensure that the name isn't too short.

Computed Names

Classes can also use computed method names just like object literals can. Here's an example:

```
class User {
    ["say" + "Hi"]() {
        alert("Hello");
    }
}

new User().sayHi();
```

The computed name is made by evaluating the expression in brackets, which has great potential for making dynamic property names.

Class Fields

A class field is a syntax that allows us to add properties to classes. Keep in mind that old browsers may require a polyfill to perform this function. All we have to do to make a class field is declare it with an equal sign. Here's an example:

```
class User {
    name = "John";

    sayHi() {
        alert(`Hello, ${this.name}`);
    }
}

new User().sayHi(); // Hello, John
```

However, class fields are set in individual objects, not the prototype. We can also put complex expressions and function calls as class fields.

All functions have a dynamic **this**, which depends on the context of the call. If we move a function declared as a class field around, the value of **this** will become undefined. We can deal with it with wrapper functions or by binding the method to the object in the constructor, or… we could use an arrow function. Like this:

```
class Button {
    constructor(value) {
        this.value = value;
    }

    click = () => {
        alert(this.value);
    }
}

let button = new Button("Hello");

setTimeout(button.click, 1000); // After one second, “Hello”
```

Unlike a normal function, the arrow function is recreated each time on a per-object basis. So it’s a separate function for each object, which resolves the problem of losing **this**. It’s also especially useful for event listeners in browser environments.