JavaScript Classes

Classes are one of the features introduced in the **ES6** version of JavaScript.
A class is a blueprint for the object. You can create an object from the class.

You can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions, you build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

Creating JavaScript Class

JavaScript class is similar to the Javascript constructor function, and it is merely a syntactic sugar.
The constructor function is defined as:

```
// constructor function
function Person () {
    this.name = 'John',
    this.age = 23
}

// create an object
const person1 = new Person();
```

Instead of using the `function` keyword, you use the `class` keyword for creating JS classes. For example,

```
// creating a class
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

The `class` keyword is used to create a class. The properties are assigned in a constructor function.

Now you can create an object. For example,

```
// creating a class
class Person {
  constructor(name) {
    this.name = name;
  }
}

// creating an object
const person1 = new Person('John');
const person2 = new Person('Jack');

console.log(person1.name); // John
console.log(person2.name); // Jack
```

Here, `person1` and person2 are objects of `Person` class.

Note: The `constructor()` method inside a class gets called automatically each time an object is created.

Javascript Class Methods

While using constructor function, you define methods as:

```
// constructor function
function Person (name) {

   // assigning  parameter values to the calling object
    this.name = name;

    // defining method
    this.greet = function () {
        return ('Hello' + ' ' + this.name);
    }
}
```

It is easy to define methods in the JavaScript class. You simply give the name of the method followed by `()`. For example,

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

    // defining method
    greet() {
        console.log(`Hello ${this.name}`);
    }
}

let person1 = new Person('John');

// accessing property
console.log(person1.name); // John

// accessing method
person1.greet(); // Hello John
```

Note: To access the method of an object, you need to call the method using its name followed by `()`.

Getters and Setters

In JavaScript, getter methods get the value of an object and setter methods set the value of an object.

JavaScript classes may include getters and setters. You use the `get` keyword for getter methods and `set` for setter methods. For example,

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

    // getter
    get personName() {
        return this.name;
    }

    // setter
    set personName(x) {
        this.name = x;
    }
}

let person1 = new Person('Jack');
console.log(person1.name); // Jack

// changing the value of name property
person1.personName = 'Sarah';
console.log(person1.name); // Sarah
```

Hoisting

A class should be defined before using it. Unlike functions and other JavaScript declarations, the class is not hoisted. For example,

```
// accessing class
const p = new Person(); // ReferenceError

// defining class
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

As you can see, accessing a class before defining it throws an error.

'use strict'

Classes always follow 'use-strict'. All the code inside the class is automatically in strict mode. For example,

```
class Person {
  constructor() {
    a = 0;
    this.name = a;
  }
}

let p = new Person(); // ReferenceError: Can't find variable: a
```

Note: JavaScript class is a special type of function. And the `typeof` operator returns `function` for a class.

For example,

```
class Person {}
console.log(typeof Person); // function
```

JavaScript Constructor Function

In JavaScript, a constructor function is used to create objects. For example,

```
// constructor function
function Person () {
    this.name = 'John',
    this.age = 23
}

// create an object
const person = new Person();
```

Run Code

In the above example, `function Person()` is an object constructor function.

To create an object from a constructor function, we use the `new` keyword.

Note: It is considered a good practice to capitalize the first letter of your constructor function.

Create Multiple Objects with Constructor Function

In JavaScript, you can create multiple objects from a constructor function. For example,

```
// constructor function
function Person () {
    this.name = 'John',
    this.age = 23,

     this.greet = function () {
        console.log('hello');
    }
}

// create objects
const person1 = new Person();
const person2 = new Person();

// access properties
console.log(person1.name);  // John
console.log(person2.name);  // John
```
Run Code

In the above program, two objects are created using the same constructor function.

JavaScript this Keyword

In JavaScript, when `this` keyword is used in a constructor function, `this` refers to the object when the object is created. For example,

```
// constructor function
function Person () {
    this.name = 'John',
}

// create object
const person1 = new Person();

// access properties
console.log(person1.name);  // John
```
Run Code

Hence, when an object accesses the properties, it can directly access the property as `person1.name`.

JavaScript Constructor Function Parameters

You can also create a constructor function with parameters. For example,

```
// constructor function
function Person (person_name, person_age, person_gender) {

   // assigning  parameter values to the calling object
    this.name = person_name,
    this.age = person_age,
    this.gender = person_gender,

    this.greet = function () {
        return ('Hi' + ' ' + this.name);
    }
}


// creating objects
const person1 = new Person('John', 23, 'male');
const person2 = new Person('Sam', 25, 'female');

// accessing properties
console.log(person1.name); // "John"
console.log(person2.name); // "Sam"
```
Run Code

In the above example, we have passed arguments to the constructor function during the creation of the object.

```
const person1 = new Person('John', 23, 'male');
const person2 = new Person('Sam', 25, 'male');
```

This allows each object to have different properties. As shown above,

`console.log(person1.name);` gives `John`

JavaScript Methods and this Keyword

In JavaScript, objects can also contain functions. For example,

```
// object containing method
const person = {
    name: 'John',
    greet: function() { console.log('hello'); }
};
```

Run Code

In the above example, a `person` object has two keys (`name` and `greet`), which have a string value and a function value, respectively.
Hence basically, the JavaScript **method** is an object property that has a function value.

Accessing Object Methods

You can access an object method using a dot notation. The syntax is:

```
objectName.methodKey()
```

You can access property by calling an **objectName** and a **key**. You can access a method by calling an **objectName** and a **key** for that method along with `()`. For example,

```
// accessing method and property
const person = {
    name: 'John',
    greet: function() { console.log('hello'); }
};

// accessing property
person.name; // John

// accessing method
person.greet(); // hello
```
Run Code

Here, the `greet` method is accessed as `person.greet()` instead of `person.greet`. If you try to access the method with only `person.greet`, it will give you a function definition.

```
person.greet; // ƒ () { console.log('hello'); }
```

JavaScript Built-In Methods

In JavaScript, there are many built-in methods. For example,

```
let number = '23.32';
let result = parseInt(number);

console.log(result); // 23
```
Run Code

Here, the `parseInt()` method of Number object is used to convert numeric string value to an integer value.
To learn more about built-in methods, visit JavaScript Built-In Methods.

Adding a Method to a JavaScript Object

You can also add a method in an object. For example,

```
// creating an object
let student = { };

// adding a property
student.name = 'John';

// adding a method
student.greet = function() {
    console.log('hello');
}

// accessing a method
student.greet(); // hello
```
Run Code

In the above example, an empty `student` object is created. Then, the `name` property is added. Similarly, the `greet` method is also added. In this way, you can add a method as well as property to an object.

JavaScript this Keyword

To access a property of an object from within a method of the same object, you need to use the `this` keyword. Let's consider an example.

```
const person = {
    name: 'John',
    age: 30,

    // accessing name property by using this.name
    greet: function() { console.log('The name is' + ' ' + this.name); }
};

person.greet();
```
Run Code

Output

```
The name is John
```

In the above example, a `person` object is created. It contains properties (`name` and `age`) and a method `greet`.

In the method `greet`, while accessing a property of an object, `this` keyword is used.

In order to access the **properties** of an object, `this` keyword is used following by `.` and **key**.

Note: In JavaScript, `this` keyword when used with the object's method refers to the object. `this` is bound to an object.

However, the function inside of an object can access it's variable in a similar way as a normal function would. For example,

```
const person = {
    name: 'John',
    age: 30,
    greet: function() {
        let surname = 'Doe';
        console.log('The name is' + ' ' + this.name + ' ' + surname); }
};

person.greet();
```

Run Code

Output

```
The name is John Doe
```

JavaScript Class Inheritance

Class Inheritance

Inheritance enables you to define a class that takes all the functionality from a parent class and allows you to add more.

Using class inheritance, a class can inherit all the methods and properties of another class.

Inheritance is a useful feature that allows code reusability.

To use class inheritance, you use the `extends` keyword. For example,

```
// parent class
class Person {
    constructor(name) {
        this.name = name;
    }

    greet() {
        console.log(`Hello ${this.name}`);
    }
}

// inheriting parent class
class Student extends Person {

}

let student1 = new Student('Jack');
student1.greet();
```

Run Code

Output

```
Hello Jack
```

In the above example, the `Student` class inherits all the methods and properties of the `Person` class. Hence, the `Student` class will now have the `name` property and the `greet()` method.

Then, we accessed the `greet()` method of `Student` class by creating a `student1` object.

JavaScript super() keyword

The `super` keyword used inside a child class denotes its parent class. For example,

```
// parent class
class Person {
    constructor(name) {
        this.name = name;
    }

    greet() {
        console.log(`Hello ${this.name}`);
    }
}

// inheriting parent class
class Student extends Person {

    constructor(name) {

        console.log("Creating student class");

        // call the super class constructor and pass in the name parameter
        super(name);
    }

}

let student1 = new Student('Jack');
student1.greet();
```

Here, `super` inside `Student` class refers to the `Person` class. Hence, when the constructor of `Student` class is called, it also calls the constructor of the `Person` class which assigns a name property to it.

Overriding Method or Property

If a child class has the same method or property name as that of the parent class, it will use the method and property of the child class. This concept is called method overriding. For example,

```
// parent class
class Person {
    constructor(name) {
        this.name = name;
        this.occupation = "unemployed";
    }

    greet() {
        console.log(`Hello ${this.name}.`);
    }

}

// inheriting parent class
class Student extends Person {

    constructor(name) {

        // call the super class constructor and pass in the name parameter
        super(name);

        // Overriding an occupation property
        this.occupation = 'Student';
    }

    // overriding Person's method
    greet() {
        console.log(`Hello student ${this.name}.`);
        console.log('occupation: ' + this.occupation);
    }
}

let p = new Student('Jack');
p.greet();
```
Run Code

Output

```
Hello student Jack.
occupation: Student
```

Here, the `occupation` property and the `greet()` method are present in parent `Person` class and the child `Student` class. Hence, the `Student` class overrides the `occupation` property and the `greet()` method.

Uses of Inheritance

- Since a child class can inherit all the functionalities of the parent's class, this allows code reusability.
- Once a functionality is developed, you can simply inherit it. No need to reinvent the wheel. This allows for cleaner code and easier to maintain.
- Since you can also add your own functionalities in the child class, you can inherit only the useful functionalities and define other required features.