JavaScript Class

A JavaScript class is a blueprint for creating `objects`. A class encapsulates data and functions that manipulate data.

Unlike other programming languages such as `Java` and `C#`, `JavaScript classes` are syntactic sugar over the `prototypal inheritance`. In other words, ES6 classes are just special functions.

Classes before ES6 revisited

Before ES6, JavaScript had no concept of classes. To mimic a class, you often use the `constructor/prototype pattern` as shown in the following example:

```
function Person(name) {
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
};

var john = new Person("John Doe");
console.log(john.getName());
```

Output:

```
John Doe
```

How it works.

First, create the `Person` as a constructor function that has a property name called `name`. The `getName()` function is assigned to the `prototype` so that it can be shared by all instances of the `Person` type.

Then, create a new instance of the `Person` type using the `new` operator. The `john` object, hence, is an instance of the `Person` and `Object` through `prototypal inheritance`.

The following statements use the `instanceof` operator to check if `john` is an instance of the `Person` and `Object` type:

```
console.log(john instanceof Person); // true
console.log(john instanceof Object); // true
```

ES6 class declaration

ES6 introduced a new syntax for declaring a class as shown in this example:

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

This `Person` class behaves like the `Person` type in the previous example. However, instead of using a constructor/prototype pattern, it uses the `class` keyword.

In the `Person` class, `the constructor()` is where you can initialize the properties of an instance. JavaScript automatically calls the `constructor()` method when you instantiate an object of the class.

The following creates a new `Person` object, which will automatically call the `constructor()` of the `Person` class:

```
let john = new Person("John Doe");
```

The `getName()` is called a method of the `Person` class. Like a constructor function, you can call the methods of a class using the following syntax:

```
objectName.methodName(args)
Code language: CSS (css)
For example:

let name = john.getName();
console.log(name); // "John Doe"
```

To verify the fact that classes are special functions, you can use the `typeof` operator of to check the type of the `Person` class.

```
console.log(typeof Person); // function
```

It returns `function` as expected.

The `john` object is also an instance of the `Person` and `Object` types:

```
console.log(john instanceof Person); // true
console.log(john instanceof Object); // true
```

Class vs. Custom type

Despite the similarities between a class and a custom type defined via a constructor function, there are some important differences.

First, class declarations are not hoisted like function declarations.

For example, if you place the following code above the `Person` class declaration section, you will get a `ReferenceError`.

```
let john = new Person("John Doe");
```

Error:

```
Uncaught ReferenceError: Person is not defined
```

Second, all the code inside a class automatically executes in the strict mode. And you cannot change this behavior.

Third, class methods are `non-enumerable`. If you use a constructor/prototype pattern, you have to use the `Object.defineProperty()` method to make a property non-enumerable.

Finally, calling the class constructor without the `new` operator will result in an error as shown in the following example.

```
let john = Person("John Doe");
```

Error:

```
Uncaught TypeError: Class constructor Person cannot be invoked without 'new'
```

Note that it's possible to call the constructor function without the new operator. In this case, the constructor function behaves like a regular function.

Summary Use the JavaScript `class` keyword to declare a new class. A `class` declaration is syntactic sugar over `prototypal inheritance` with additional enhancements.

JavaScript Getters and Setters

Introduction to the JavaScript `getters` and `setters` The following example defines a class called `Person`:

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

let person = new Person("John");
console.log(person.name); // John
```

The `Person` class has a property `name` and a constructor. The constructor initializes the `name` property to a string.

Sometimes, you don't want the `name` property to be accessed directly like this:

```
person.name
```

To do that, you may come up with a pair of methods that manipulate the `name` property. For example:

```
class Person {
    constructor(name) {
        this.setName(name);
    }
    getName() {
        return this.name;
    }
    setName(newName) {
        newName = newName.trim();
        if (newName === '') {
            throw 'The name cannot be empty';
        }
        this.name = newName;
    }
}

let person = new Person('Jane Doe');
console.log(person); // Jane Doe

person.setName('Jane Smith');
console.log(person.getName()); // Jane Smith
```

In this example, the `Person` class has the `name` property. Also, it has two additional methods `getName()` and `setName()`.

The `getName()` method returns the value of the `name` property.

The `setName()` method assigns an argument to the `name` property. The `setName()` removes the whitespaces from both ends of the `newName` argument and throws an exception if the `newName` is empty.

The constructor() calls the `setName()` method to initialize the name property:

```
constructor(name) {
    this.setName(name);
}
```

The `getName()` and `setName()` methods are known as getter and setter in other programming languages such as Java and C++.

ES6 provides a specific syntax for defining the getter and setter using the get and set keywords. For example:

```
class Person {
    constructor(name) {
        this._name = name;
    }
    get name() {
        return this._name;
    }
    set name(newName) {
        newName = newName.trim();
        if (newName === '') {
            throw 'The name cannot be empty';
        }
        this._name = newName;
    }
}
```

How it works.

First, the `name` property is changed to `_name` to avoid the name collision with the getter and setter.

Second, the getter uses the `get` keyword followed by the method name:

```
get name() {
    return this._name;
}
```

To call the getter, you use the following syntax:

```
let name = person.name;
```

When JavaScript sees the access to name property of the `Person` class, it checks if the `Person` class has any `name` property.

If not, JavaScript checks if the Person class has any method that binds to the `name` property. In this example, the `name()` method binds to the `name` property via the `get` keyword. Once JavaScript finds the getter method, it executes the getter method and returns a value.

Third, the setter uses the `set` keyword followed by the method name:

```
set name(newName) {
    newName = newName.trim();
    if (newName === '') {
        throw 'The name cannot be empty';
    }
    this._name = newName;
}
```

JavaScript will call the `name()` setter when you assign a value to the `name` property like this:

```
person.name = 'Jane Smith';
```

If a class has only a getter but not a setter and you attempt to use the setter, the change won't take any effect. See the following example:

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

let person = new Person("Jane Doe");
console.log(person.name);

// attempt to change the name, but cannot
person.name = 'Jane Smith';
console.log(person.name); // Jane Doe
```

In this example, the `Person` class has the `name` getter but not the `name` setter. It attempts to call the setter. However, the change doesn't take effect since the Person class doesn't have the `name` setter.

Using getter in an object literal

The following example defines a getter called `latest` to return the latest attendee of the `meeting` object:

```
let meeting = {
    attendees: [],
    add(attendee) {
        console.log(`${attendee} joined the meeting.`);
        this.attendees.push(attendee);
        return this;
    },
    get latest() {
        let count = this.attendees.length;
        return count == 0 ? undefined : this.attendees[count - 1];
    }
};

meeting.add('John').add('Jane').add('Peter');
console.log(`The latest attendee is ${meeting.latest}.`);
```

Output:

```
John joined a meeting.
Jane joined a meeting.
Peter joined a meeting.
```

The latest attendee is Peter.

Summary

- Use the `get` and `set` keywords to define the JavaScript getters and setters for a class or an object.
- The `get` keyword binds an object property to a method that will be invoked when that property is looked up.
- The `set` keyword binds an object property to a method that will be invoked when that property is assigned.

JavaScript Class Expressions

Introduction to JavaScript class expressions

Similar to `functions`, `classes` have expression forms. A class expression provides you with an alternative way to define a new class.

A class expression doesn't require an identifier after the `class` keyword. You can use a class expression in a `variable declaration` and pass it into a function as an argument.

For example, the following defines a class expression:

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

How it works.

On the left side of the expression is the `Person` variable. It’s assigned to a class expression.

The class expression starts with the keyword `class` followed by the class definition.

A class expression may have a name or not. In this example, we have an unnamed class expression.

If a class expression has a name, its name can be local to the class body.

The following creates an instance of the `Person` class expression. Its syntax is the same as if it were a class declaration.

```
let person = new Person('John Doe');
```

Like a `class declaration`, the type of a class expression is also a `function`:

```
console.log(typeof Person); // function
```

Similar to function expressions, class expressions are not hoisted. It means that you cannot create an instance of the class before defining the class expression.

First-class citizen

`JavaScript classes are first-class citizens`. It means that you can pass a class into a function, return it from a function, and assign it to a variable.

See the following example:

```
function factory(aClass) {
    return new aClass();
}

let greeting = factory(class {
    sayHi() { console.log('Hi'); }
});

greeting.sayHi(); // 'Hi'
```

How it works.

First, define a `factory()` function that takes a class expression as an argument and returns the instance of the class:

```
function factory(aClass) {
    return new aClass();
}
```

Second, pass an unnamed class expression to the `factory()` function and assign its result to the greeting variable:

```
let greeting = factory(class {
    sayHi() { console.log('Hi'); }
});
```

The class expression has a method called `sayHi()`. And the greeting variable is an instance of the class expression.

Third, call the `sayHi()` method on the greeting object:

```
greeting.sayHi(); // 'Hi'
```

Singleton

Singleton is a design pattern that limits the instantiation of a class to a single instance. It ensures that only one instance of a class can be created throughout the system.

Class expressions can be used to create a singleton by calling the class constructor immediately.

To do that, you use the `new` operator with a class expression and include the parentheses at the end of the class declaration as shown in the following example:

```
let app = new class {
    constructor(name) {
        this.name = name;
    }
    start() {
        console.log(`Starting the ${this.name}...`);
    }
}('Awesome App');

app.start(); // Starting the Awesome App...
```

How it works.

The following is an unnamed class expression:

```
new class {
    constructor(name) {
        this.name = name;
    }
    start() {
        console.log(`Starting the ${this.name}...`);
    }
}
```

The class has a `constructor()` that accepts an argument. It also has a method called `start()`.

The class expression evaluates to a class. Therefore, you can call its constructor immediately by placing parentheses after the expression:

```
new class {
    constructor(name) {
        this.name = name;
    }
    start() {
        console.log(`Starting the ${this.name}...`);
    }
}('Awesome App')
```

This expression returns an instance of the class expression which is assigned to the app variable.

The following calls the `start()` method on the app object:

```
app.start(); // Starting the Awesome App...
```

Summary

- ES6 provides you with an alternative way to define a new class using a class expression.
- Class expressions can be named or unnamed.
- The class expression can be used to create a singleton object.

JavaScript Inheritance Using extends & super

Implementing JavaScript inheritance using `extends` and `super`

Prior to ES6, implementing a proper inheritance required multiple steps. One of the most commonly used strategies is prototypal inheritance.

The following illustrates how the `Bird` inherits properties from the `Animal` using the prototypal inheritance technique:

```
function Animal(legs) {
    this.legs = legs;
}

Animal.prototype.walk = function() {
    console.log('walking on ' + this.legs + ' legs');
}

function Bird(legs) {
    Animal.call(this, legs);
}

Bird.prototype = Object.create(Animal.prototype);
Bird.prototype.constructor = Animal;


Bird.prototype.fly = function() {
    console.log('flying');
}

var pigeon = new Bird(2);
pigeon.walk(); // walking on 2 legs
pigeon.fly();  // flying
```

ES6 simplified these steps by using the `extends` and `super` keywords.

The following example defines the `Animal` and `Bird` classes and establishes the inheritance through the `extends` and `super` keywords.

```
class Animal {
    constructor(legs) {
        this.legs = legs;
    }
    walk() {
        console.log('walking on ' + this.legs + ' legs');
    }
}

class Bird extends Animal {
    constructor(legs) {
        super(legs);
    }
    fly() {
        console.log('flying');
    }
}
```

```
let bird = new Bird(2);

bird.walk();
bird.fly();
```

How it works.

First, use the `extends` keyword to make the `Bird` class inheriting from the `Animal` class:

```
class Bird extends Animal {
   // ...
}
```

The `Animal` class is called a **base class** or **parent class** while the `Bird` class is known as a derived class or child class. By doing this, the Bird class inherits all methods and properties of the `Animal` class.

Second, in the Bird's constructor, call `super()` to invoke the Animal's constructor with the legs argument.

JavaScript requires the child class to call `super()` if it has a constructor. As you can see in the `Bird` class, the `super(legs)` is equivalent to the following statement in ES5:

```
Animal.call(this, legs);
```

If the `Bird` class doesn't have a constructor, you don't need to do anything else:

```
class Bird extends Animal {
    fly() {
        console.log('flying');
    }
}
```

It is equivalent to the following class:

```
class Bird extends Animal {
    constructor(...args) {
        super(...args);
    }
    fly() {
        console.log('flying');
    }
}
```

However, the child class has a constructor, it needs to call `super()`. For example, the following code results in an error:

```
class Bird extends Animal {
    constructor(legs) {
    }
    fly() {
        console.log('flying');
    }
}
```

Error:

```
ReferenceError: Must call super constructor in derived class before accessing
'this' or returning from derived constructor
```

Because the `super()` initializes the `this` object, you need to call the `super()` before accessing the this object. Trying to access `this` before calling `super()` also results in an error.

For example, if you want to initialize the `color` property of the `Bird` class, you can do it as follows:

```
class Bird extends Animal {
    constructor(legs, color) {
        super(legs);
        this.color = color;
    }
    fly() {
        console.log("flying");
    }
    getColor() {
        return this.color;
    }
}

let pegion = new Bird(2, "white");
console.log(pegion.getColor());
```

Shadowing methods

ES6 allows the child class and parent class to have methods with the same name. In this case, when you call the method of an object of the child class, the method in the child class will shadow the method in the parent class.

The following `Dog` class extends the `Animal` class and redefines the `walk()` method:

```
class Dog extends Animal {
    constructor() {
        super(4);
    }
    walk() {
        console.log(`go walking`);
    }
}

let bingo = new Dog();
bingo.walk(); // go walking
```

To call the method of the parent class in the child class, you use `super.method(arguments)` like this:

```
class Dog extends Animal {
    constructor() {
        super(4);
    }
    walk() {
        super.walk();
        console.log(`go walking`);
    }
}

let bingo = new Dog();
bingo.walk();
// walking on 4 legs
// go walking
```

Inheriting static members

Besides the properties and methods, the child class also inherits all static properties and methods of the parent class. For example:

```
class Animal {
    constructor(legs) {
        this.legs = legs;
    }
    walk() {
        console.log('walking on ' + this.legs + ' legs');
    }
    static helloWorld() {
        console.log('Hello World');
    }
}

class Bird extends Animal {
    fly() {
        console.log('flying');
    }
}
```

In this example, the Animal class has the helloWorld() static method and this method is available as Bird.helloWorld() and behaves the same as the Animal.helloWorld() method:

```
Bird.helloWorld(); // Hello World
```

Inheriting from built-in types

JavaScript allows you to extend a built-in type such as Array, String, `Map`, and `Set` through inheritance.

The following `Queue` class extends the `Array` reference type. The syntax is much cleaner than the `Queue` implemented using the constructor/prototype pattern.

```
class Queue extends Array {
    enqueue(e) {
        super.push(e);
    }
    dequeue() {
        return super.shift();
    }
    peek() {
        return !this.empty() ? this[0] : undefined;
    }
    empty() {
        return this.length === 0;
    }
}

var customers = new Queue();
customers.enqueue('A');
customers.enqueue('B');
customers.enqueue('C');

while (!customers.empty()) {
    console.log(customers.dequeue());
}
```

Summary

- Use the `extends` keyword to implement the inheritance in ES6. The class to be extended is called a base class or parent class. The class that extends the base class or parent class is called the derived class or child class.
- Call the `super(arguments)` in the child class's constructor to invoke the parent class's constructor.
- Use `super` keyword to call methods of the parent class in the methods of the child class.