**05 ES6 Classes**

**1) ES6 Classes**:

JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax does not introduce a new object-oriented inheritance model to JavaScript.

Classes are in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.

**Example**:

*/\**

*function Circle(radius) {*

*this.radius = radius;*

*this.draw = function() {*

*console.log("draw");*

*};*

*}*

*\*/*

*//write code using ES-6 class*

class Circle {

*//instance property*

constructor(radius) {

*this*.radius = radius;

*this*.move = function() {

console.log("move");

};

}

*//prototypical property*

draw() {

console.log("draw");

}

}

const c = new Circle(10);

console.log(c); *//Circle {radius: 10, move: ƒ}*

*/\**

*Circle {radius: 10, move: ƒ}*

*move: ƒ ()*

*radius: 10*

*\_\_proto\_\_:*

*constructor: class Circle*

*draw: ƒ draw()*

*\_\_proto\_\_: Object*

*\*/*

In JavaScript classes are essentially function.

console.log(typeof Circle); *//function*

**2) Hoisting**:

Hoisting is JavaScript's default behavior of moving declarations to the top.

In JavaScript we can define a class in two ways by using declaration or by using expression syntax.

**Example**:

*//class Declaration*

class Circle{

}

*//class Expression*

const Square = Class{

}

Unlike functions class declaration or class expression are not hoisting. If we cannot create the Circle object like this.

**Example**:

const c = new Circle(); *//Uncaught ReferenceError: Circle is not defined*

class Circle{

}

**Note**:

The recommended approach to declare a class in JavaScript is "using declaration". Using Expression

is not used.

**3) Static Methods**:

The static keyword defines a static method for a class. Static methods aren't called on instances of the class. Instead, they're called on the class itself. These are often utility functions, such as functions to create or clone objects.

**Example**:

class Circle {

constructor(radius) {

*this*.radius = radius;

}

draw() {

console.log("draw");

}

static possition() {

console.log("possition");

}

}

const c = new Circle(20);

console.log(c); *//Circle {radius: 20}*

*/\**

*Circle {radius: 20}*

*radius: 20*

*\_\_proto\_\_:*

*constructor: class Circle*

*draw: ƒ draw()*

*\_\_proto\_\_: Object*

*\*/*

The static member is not available in object level it is available only class level. We can access static member as following.

**Example**:

class Circle {

constructor(radius) {

*this*.radius = radius;

}

draw() {

console.log("draw");

}

static possition() {

console.log("possition");

}

}

console.log(Circle.possition()); *//position*

**4) The this keyword**:

The JavaScript this keyword refers to the object it belongs to.

1. In a method, this refers to the owner object.
2. Alone, this refers to the global object.
3. In a function, this refers to the global object.
4. In a function, in strict mode, this is undefined.
5. In an event, this refers to the element that received the event.
6. Methods like call(), and apply() can refer this to any object.

**Inside function this point to the current object**:

const Circle = function() {

*this*.draw = function() {

console.log(*this*);

};

};

const c = new Circle();

c.draw(); *//Circle {draw: ƒ}*

*/\**

*Circle {draw: ƒ}*

*draw: ƒ ()*

*\_\_proto\_\_: Object*

*\*/*

**Here this point to window object**:

const Circle = function() {

*this*.draw = function() {

console.log(*this*); *//this refer window now*

};

};

const c = new Circle();

const draw = c.draw;

*//function call*

draw();

*/\**

*Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …}*

*\*/*

Here we call "draw()" as a standalone function which is not part of the object. When we call draw() as a function "this" point to the window object in browser.

**In a function, in strict mode, this is undefined**:

"use strict";

const Circle = function() {

*this*.draw = function() {

console.log(*this*);

};

};

const c = new Circle();

const draw = c.draw;

*//function call*

draw(); *//undefined*

**By default, the body of our classes is strict mode ("use strict")**

class Circle {

draw() {

console.log(*this*);

}

}

const c = new Circle();

const draw = c.draw;

*//function call*

draw();

*//undefined*

**5)** **Private Members Using Symbols**:

Abstraction is one of the core principals of object-oriented programming. Abstraction means hide the details and complexity and only show the essential part.

To implement abstraction, we use private properties and method. So, we hide certain members of an object so that are not accessible from outside the object. Suppose we have a Circle object and in this object, we have radius property. We want to make this property private.

**Example**:

class Circle {

constructor(radius) {

*this*.radius = radius;

}

}

**Use underscore approach**:

Using an underscore (\_) as a naming convention. Some developer name their private properties or methods using an underscore. But this approach is not correct because still we can access the member from outside the object.

**Example**:

class Circle {

constructor(radius) {

*this*.\_radius = radius;

}

}

const circle = new Circle(10);

*//radius is not private because it is accessible from outside object*

console.log(circle.\_radius); *//10*

**Use ES6 symbol approach**:

We can use ES6 symbol to implement private properties and method. In ES6 we have a new primitive type call symbol. Symbol() is a function to generate symbol. The symbol is generally a unique identifier.

Every time we call the function, we generate a unique identifier. Remember this is not a constructor function so we cannot use the new keyword.

Every time we call the Symbol() function we will get a new unique value. We can use this unique value as the property name of an object.

**Example**:

*//Symbol() is a function to generate symbol.*

const \_radius = Symbol();

class Circle {

constructor(radius) {

*this*[\_radius] = radius;

}

}

const circle = new Circle(10);

console.log(circle); *//Circle {Symbol(): 10}*

*/\**

*Circle {Symbol(): 10}*

*Symbol(): 10*

*\_\_proto\_\_: Object*

*\*/*

Now the property radius is represent by Symbol (Symbol(): 10). Here the property radius is not accessible from outside Circle object.

const circle = new Circle(10);

*//dont show any property*

console.log(Object.getOwnPropertyNames(circle)); *//[]*

But we can access the property value.

const circle = new Circle(10);

const key = Object.getOwnPropertySymbols(circle)[0];

console.log(circle[key]); *//10*

**Implement private method**:

const \_radius = Symbol();

const \_draw = Symbol();

class Circle {

constructor(radius) {

*this*[\_radius] = radius;

}

[\_draw]() {}

}

const circle = new Circle(10);

console.log(circle); *//Circle {Symbol(): 10}*

*/\**

*Circle {Symbol(): 10}*

*Symbol(): 10*

*\_\_proto\_\_:*

*constructor: class Circle*

*Symbol(): ƒ [\_draw]()*

*\_\_proto\_\_: Object*

*\*/*

**6) Private Members Using WeakMaps**:

WeakMap is a new type in ES6 to implement private properties and methods in an object. The WeakMap object is a collection of key/value pairs in which the keys are weakly referenced. The keys must be objects and the values can be arbitrary values.

**WeakMap Object**:

The WeakMap object is a collection of key/value pairs in which the keys are objects only and the values can be arbitrary values. The object references in the keys are held weakly, meaning that they are a target of garbage collection (GC) if there is no other reference to the object anymore. The WeakMap API is the same as the Map API.

One difference to Map objects is that WeakMap keys are not enumerable (i.e., there is no method giving you a list of the keys). If they were, the list would depend on the state of garbage collection, introducing non-determinism.

Consider the Circle object

class Circle {

constructor(radius) {

*this*.radiu = radius;

}

}

const c = new Circle(10);

Now we make the radius property into a private property.

const \_radius = new WeakMap();

class Circle {

constructor(radius) {

*//this refer to Circle object*

\_radius.set(*this*, radius);

}

}

const c = new Circle(10);

console.log(c); *//Circle {}*

*/\**

*Circle {}*

*\_\_proto\_\_:*

*constructor: class Circle*

*\_\_proto\_\_: Object*

*\*/*

Here radius is not accessible from outside the Circle object. Now if we want to access the radius property inside Circle class, we have to do the following thing.

**Example**:

const \_radius = new WeakMap();

class Circle {

constructor(radius) {

*//this refer to Circle object*

\_radius.set(*this*, radius);

}

*//access radius from outside class*

draw() {

*//this refer to Circle object*

console.log(\_radius.get(*this*));

}

}

const c = new Circle(10);

c.draw(); *//10*

**Define a private method**:

Here to define a private method we use another key map.

**Example**:

const \_radius = new WeakMap();

const \_move = new WeakMap();

class Circle {

constructor(radius) {

*//this refer to Circle object*

\_radius.set(*this*, radius);

*//this -> key (Circle object) function -> value*

\_move.set(*this*, function() {

*//Circle class is executed in "strict mode" and this will be set*

*to undefined*

console.log("Move", *this*);

});

}

*//access radius from outside class*

draw() {

\_move.get(*this*)();

console.log("Draw");

}

}

const c = new Circle(10);

console.log(c.draw());

*/\**

*Move undefined*

*Draw*

*undefined*

*\*/*

("");

Here after the move we will see undefine because the body of Circle class is executed in "strict mode".

Now in the particular implementation, maybe in the move method, we want to access the instance of the Circle object. For this we have to use an array function in "\_move.set()". Because array function uses this value of there containing function. In this case this inherited from constructor function.

**Example**:

const \_radius = new WeakMap();

const \_move = new WeakMap();

class Circle {

constructor(radius) {

*//this refer to Circle object*

\_radius.set(*this*, radius);

*//this -> key (Circle object) function -> value*

\_move.set(*this*, () => {

*//now this inherit from constructor*

console.log("Move", *this*);

});

}

*//access radius from outside class*

draw() {

\_move.get(*this*)();

console.log("Draw");

}

}

const c = new Circle(10);

console.log(c.draw());

*/\**

*Move Circle {}*

*Draw*

*undefined*

*\*/*

We are using a separate WeakMap for each property and method. Why we don’t create just one WeakMap for the all the private members. Like this

**Example**:

const \_radius = new WeakMap();

const \_move = new WeakMap();

const privateProps = new WeakMap();

class Circle {

constructor(radius) {

privateProps.set(*this*, {

radius: radius,

move: () => {}

});

\_radius.set(*this*, radius);

\_move.set(*this*, () => {

console.log("Move", *this*);

});

}

draw() {

\_move.get(*this*)();

console.log("Draw");

}

}

const c = new Circle(10);

console.log(c.draw());

But this approach is not recommended.

**7) Getters and Setters**:

The get syntax binds an object property to a function that will be called when that property is looked up.

The set syntax binds an object property to a function to be called when there is an attempt to set that property.

In ES6 we can create a getter() and setter() method much easier way.

**Example**:

const \_radius = new WeakMap();

class Circle {

constructor(radius) {

\_radius.set(*this*, radius);

}

set radius(value) {

if (value <= 0) {

throw new Error("Invalid radius");

}

\_radius.set(*this*, value);

}

get radius() {

return \_radius.get(*this*);

}

}

const c = new Circle(10);

*/\**

*c.radius*

*10*

*c.radius = 20*

*20*

*c.radius = -40*

*Uncaught Error: Invalid radius*

*\*/*

**8) Inheritance**:

ES6 has made JavaScript look a lot simpler with the class syntax and its additional features. Today we are going to combine the class syntax feature with the concept of inheritance to get some code going. Yes, we are going to take a look at the super and extends keywords in JavaScript’s ES6.

**Example**:

class Shape {

move() {

console.log("Move");

}

}

class Circle extends Shape {

draw() {

console.log("Draw");

}

}

const c = new Circle();

console.log(c); *//Circle {}*

*/\**

*Circle {}*

*\_\_proto\_\_: Shape*

*constructor: class Circle*

*draw: ƒ draw()*

*\_\_proto\_\_:*

*constructor: class Shape*

*move: ƒ move()*

*\_\_proto\_\_: Object*

*\*/*

If we have a constructor in the parent class and a if we have a constructor in the child class, inside the child class we must have to call the parent class constructor by using the super() method. Otherwise we will get Error.

**Example**:

class Shape {

constructor(color) {

*this*.color = color;

}

move() {

console.log("Move");

}

}

class Circle extends Shape {

constructor() {}

draw() {

console.log("Draw");

}

}

const c = new Circle();

console.log(c); *//Circle {}*

*/\**

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

*\*/*

**Example**:

class Shape {

constructor(color) {

*this*.color = color;

}

move() {

console.log("Move");

}

}

class Circle extends Shape {

constructor(color, radius) {

*super*(color);

*this*.radius = radius;

}

draw() {

console.log("Draw");

}

}

const c = new Circle("red", 10);

console.log(c); *//Circle {color: "red", radius: 10}*

*/\**

*Circle {color: "red", radius: 10}*

*color: "red"*

*radius: 10*

*\_\_proto\_\_: Shape*

*constructor: class Circle*

*draw: ƒ draw()*

*\_\_proto\_\_:*

*constructor: class Shape*

*move: ƒ move()*

*\_\_proto\_\_: Object*

*\*/*

**9) Method Riding**:

In JavaScript we can reimplement the parent class method in child class. This concept is called method riding.

**Example**:

**10) Exercise**:

**11) Solution**:

05 ES6 Classes