JavaScript Notes
(part 2)

Julian Shen
08/09/2021
(compilated elements from
codewithmosh.com JavaScript (part 2)
MDN Web Docs Mozilla)

Adding or Removing Properties of Objects

- The delete operator removes a property from an object, if no more references to the same property are held, it is eventually released automatically. On successful deletion, it will return true, else false will be returned.

Example

```
function Circle(radius) {
	this.radius = radius;
}

const circle = new Circle(10);

// Adding Property
circle.location = { x: 1, y: 1 };
console.log(circle); // Circle { radius: 10, location: { x: 1, y:
1 } }
circle.draw = function() {
	console.log('draw');
}
console.log(circle); // Circle { radius: 10, location: { x: 1, y: 1 },
draw: [Function] }

// Delete Property
delete circle.draw;
console.log(circle); // Circle { radius: 10, location: { x: 1, y:
1 } }
```

Abstraction

- Abstraction is a way to reduce complexity and allow efficient design and implementation in complex software systems. It hides the technical complexity of systems behind simpler API’s. It provides only the necessary details on a ‘need to know’ basis.

Private Properties and Methods of Objects

- Closure is the combination of a function bundled together (enclosed) with references to its surrounding state (lexical environment). It gives you access to an outer function’s scope from an inner function. In JS, closures are created every time a function is created.
- Closure determines what variables will be accessible to an inner function. Scope is temporary, Closure is not temporary.

Example

```
function Circle(radius) {
      // public
      this.radius = radius;
      this.draw = function() {
            computeOptimumLocation();
            console.log('draw');
      };

      // private
      let defaultLocation = { x: 0, y: 0 };
      let computeOptimumLocation = function() {
            console.log('compute');
            console.log('Location', defaultLocation);
      };
}

const circle = new Circle(10);
circle.draw(); // compute, Location { x: 0, y: 0 }, draw

// For draw(), draw can access variables within its scope {...}, as well as variables defined in its parent 'Circle'
// 'defaultLocation' and 'computeOptimumLocation()' are in the closure of the 'draw()' variable.
```

Getters and Setters of Objects

The static method Object.defineProperty() defines a new property directly on an object, or modifies an existing property on an object, and returns the object.

Syntax

```
Object.defineProperty(obj, prop, descriptor)
```

- obj – the object on which to define the property
- prop – the name or symbol of the property to be defined or modified
- descriptor – the descriptor for the property being defined or modified.

Example

```
function Circle(radius) {
      // public
      this.radius = radius;
      this.draw = function() {
            computeOptimumLocation();
            console.log('draw');
      };
      // getters and setters
      Object.defineProperty(this, 'defaultLocation', {
            get: function() {
                  return defaultLocation;
            },
            set: function(value) {
                  if (!value.x || !value.y)
                        throw new Error('Invalid location.');
                  defaultLocation = value;
            }
      });
      // private
      let defaultLocation = { x: 0, y: 0 };
      let computeOptimumLocation = function() {
            console.log('Location', defaultLocation);
      };
}
const circle = new Circle(10);
circle.draw(); // Location { x: 0, y: 0 }, draw
console.log(circle.defaultLocation); // { x: 0, y: 0 }
// circle.defaultLocation = 1; // Error: Invalid location.
// set defaultLocation
circle.defaultLocation = { x: 1, y: 3 };
console.log(circle.defaultLocation); // { x: 1, y: 3 }
```

Object Prototypes

- Prototypes are the mechanism by which JS objects inherit features from one another.
- Objects can have a prototype object, which acts as a template object that it inherits methods and properties from.
- An object’s prototype object may also have a prototype object, which it inherits methods and properties from, and so on. This is referred to as a prototype chain.
- The Object.getPrototypeOf() method returns the prototype of the specified object. (__proto__ is deprecated)

Syntax

```
Object.getPrototypeOf(obj)
```

Example

```
// Single-level Inheritance

// Both x and y inherit from Object prototype
let x = {};
let y = {};
console.log(Object.getPrototypeOf(x) === Object.getPrototypeOf(y)); // true


// Multi-level Inheritance

// myArray inherits from Array prototype, which inherits from Object prototype
let myArray = [];

function Circle(radius) {
      this.radius = radius;

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

// circle1 inherits from Circle prototype, which inherits from Object prototype
const circle1 = new Circle(10);
```

Property Descriptors

- The Object.getOwnPropertyDescriptors() method returns an object containing all own property descriptors of an object.

- The static method Object.defineProperty() defines a new property directly on an object, or modifies an existing property on an object, and returns the object.

Property Descriptors (data descriptors and accessor descriptors)

- value – the value associated with the property (data descriptors only) (Default: undefined)

- writable – true if and only if the value associated with the property may be changed (data descriptors only) (Default: false)

- configurable – true if and only if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object (Default: false)

- enumerable – true if and only if this property shows up during enumeration of the properties on the corresponding object (Default: false)

- get – a getter for the property, or undefined if non-existent (accessor descriptors only) (Default: undefined)

- set – a setter for the property, or undefined if non-existent (accessor descriptors only) (Default: undefined)

Syntax

Object.getOwnPropertyDescriptors(obj)

Object.defineProperty(obj, prop, descriptor)

- obj – the object on which to define the property
- prop – the name or symbol of the property to be defined or modified
- descriptor – the descriptor for the property being defined or modified

Example

```
let person = { name: 'Julian' };
console.log(person); // { name: 'Julian' }

for (let key in person) {
      console.log(key); // name
}

let objectBase = Object.getPrototypeOf(person);
let descripter = Object.getOwnPropertyDescriptor(objectBase,
'toString');
console.log(descripter); //{ value: [Function: toString], writable:
true, enumerable: false, configurable: true }

Object.defineProperty(person, 'name', {
      writable: false,
      enumerable: false
});

person.name = 'John';
console.log(person.name); // Julian // Didn't change because writable:
false
console.log(Object.keys(person)); // [] // Empty array because
enumerable: false
```

Prototypes and Instance Members

Example

```
function Circle(radius) {
      // Instance Members
      this.radius = radius;
      this.move = function() {
            console.log('move');
      }
}

// Prototype Members
// add draw method to Circle's prototype (Object)
Circle.prototype.draw = function() {
      this.move();
      console.log('draw');
}
// change toString method from Object prototype
Circle.prototype.toString = function() {
      return 'Circle with radius: ' + this.radius;
}

const c1 = new Circle(1);
c1.draw(); // move, draw
c1.move(); // move
console.log(c1.toString()); // Circle with radius: 10
```

Iterating Instance and Prototype Members

- The Object.keys() method returns an array of a given object’s own enumerable property names, iterated in the same order a normal loop would.
- The Object.hasOwn() static method returns true if the specified object has the indicated property as its own property. If the property is inherited, or does not exist, the method returns false.

(.hasOwnProperty() is problematic…?)

Syntax

```
Object.keys(obj)

Object.hasOwn(instance, prop)
```

- instance – the JS object instance to test
- prop – the string name or symbol of the property to test

Example

```
function Circle(radius) {
      // Instance Members
      this.radius = radius;
      this.move = function() {
            console.log('move');
      }
}
// Prototype Members
Circle.prototype.draw = function() {
      console.log('draw');
}
const c1 = new Circle(1);

// Object.keys() only returns instance members
console.log(Object.keys(c1)); // [ 'radius', 'move' ]

// For...in returns all members (instance + prototype)
for (let key in c1) {
      console.log(key); // radius, move, draw
}

// hasOwnProperty(), 'own' refers to 'instance'
console.log(c1.hasOwnProperty('radius')); // true
console.log(c1.hasOwnProperty('draw')); // false
// console.log(Object.hasOwn(c1, 'radius')); // true
// console.log(Object.hasOwn(c1, 'draw')); // false
```

Prototypical Inheritance

- The Object.create() method creates a new object, using an existing object as the prototype of the newly created object.
- The constructor property returns a reference to the Object constructor function that created the instance object. Note: the value of this property is a reference to the function itself, not a string containing the function’s name.

Example

```
function Shape() {
    // ...
}
Shape.prototype.duplicate = function() {
    console.log('duplicate');
}

function Circle(radius) {
    this.radius = radius;
}
// ***** set Circle to inherit from Shape
Circle.prototype = Object.create(Shape.prototype);
// ***** When setting one object to inherit from another, its constructor needs to be reset as well
Circle.prototype.constructor = Circle;

// add draw() to Circle’s prototype
Circle.prototype.draw = function() {
    console.log('draw');
}

const c = new Circle(1);
c.draw(); // draw
// Now, c (Circle Object) can call on duplicate() which it inherits from Shape Object
c.duplicate(); // duplicate
```

Calling the Super Constructor

- The call() method calls a function with a given this value and arguments provided individually.

Syntax

```
call()
call(thisArg)
call(thisArg, arg1,…,argN)
```

- thisArg (optional) – the value to use as this when calling function
- arg1,…argN – arguments for the function

Example

```
// Intermediate Function Inheritance
// ***** function to create inheritance chain
function extend(Child, Parent) {
      Child.prototype = Object.create(Parent.prototype);
      Child.prototype.constructor = Child;
}
// Shape
function Shape(color) {
      this.color = color;
}
Shape.prototype.duplicate = function() {
      console.log('duplicate');
}
// Circle
function Circle(radius, color) {
      // ***** calling the super constructor
      Shape.call(this, color);
      this.radius = radius;
}
extend(Circle, Shape);
// Square
function Square(size) {
      this.size = size;
}
extend(Square, Shape);

const s = new Square(25);
const c = new Circle(1, 'red');
console.log(c.color); // red
c.duplicate(); // duplicate
s.duplicate(); // duplicate
```

Method Overriding in Objects

Example

```
function extend(Child, Parent) {
	Child.prototype = Object.create(Parent.prototype);
	Child.prototype.constructor = Child;
}
// Shape
function Shape() {
}
Shape.prototype.duplicate = function() {
	console.log('duplicate');
}
// Circle
function Circle() {
}
extend(Circle, Shape);
// ***** Method Overriding (place after extend function)
Circle.prototype.duplicate = function() {
	console.log('duplicate circle');
}
// ***** How to call Parent's overridden duplicate function
Circle.prototype.parentDuplicate = function() {
	Shape.prototype.duplicate.call(this);
}

const c = new Circle();
c.duplicate(); // duplicate circle
c.parentDuplicate(); // duplicate
```

Polymorphism

- Polymorphism is the presentation of one interface for multiple data types.

Example

```
function extend(Child, Parent) {
      Child.prototype = Object.create(Parent.prototype);
      Child.prototype.constructor = Child;
}
// Shape
function Shape() {
}

Shape.prototype.duplicate = function() {
      console.log('duplicate');
}
// Circle
function Circle() {
}
extend(Circle, Shape);
Circle.prototype.duplicate = function() {
      console.log('duplicate circle');
}
// Square
function Square() {
}
extend(Square, Shape);
Square.prototype.duplicate = function() {
      console.log('duplicate square');
}

const shapes = [
      new Circle(),
      new Square(),
      new Circle(),
      new Square()
];

for(let shape of shapes) {
      shape.duplicate(); // duplicate circle, duplicate square, duplicate circle, duplicate square
}
```

<u>Mixins</u>

- A mixin is a class in which some or all of its methods and/or properties are unimplemented, requiring another class or interface to provide the missing implementations. Used to simplify the design of APIs where multiple interfaces need to include the same methods and properties.

<u>Example</u>

```
// ***** Function to assign multiple feature objects
function mixin(target, ...sources) {
     Object.assign(target, ...sources);
}
// ***** defining an 'eat' feature as an object
const canEat = {
     eat: function() {
          console.log('eating');
     }
};
// ***** defining a 'walk' feature as an object
const canWalk = {
     walk: function() {
          console.log('walking');
     }
};
// ***** defining a 'swim' feature as an object
const canSwim = {
     swim: function() {
          console.log('swim');
     }
}
// Person
function Person() {
}
// ***** assigning 'eat' and 'walk' features to Person prototype
mixin(Person.prototype, canEat, canWalk, canSwim);
const person = new Person();
console.log(person); // Person {}
// Fish
function Fish() {
}
// ***** assigning ‘eat’ and ‘swim’ features to Fish prototype
mixin(Fish.prototype, canEat, canSwim);
const fish = new Fish();
console.log(fish); // Fish {}
```

ES6 Classes

- Classes are a template for creating objects. They encapsulate data with code to work on that data. Classes in ES6 are syntactic sugar over prototypical inheritance.
- Methods defined in the constructor exist within the instance of the class
- Methods defined in the body of the class exist within the prototype of the class
- Note: Because classes are constructor functions, using typeof will return ‘function’.

Example

```
class Circle {
	constructor(radius) {
		// exists within the instance of the class
		this.radius = radius;
		this.filled = false;
		this.move = function() {}
	}
	// exists within the prototype of the class
	draw() {
		console.log('draw');
	}
	fill() {
		this.filled = true;
	}
	unFill() {
		this.filled = false;
	}
}

const c = new Circle(25);
console.log(c.radius); // 25
c.draw(); // draw
console.log(c.filled); // false
c.fill();
console.log(c.filled); // true
c.unFill();
console.log(c.filled); // false
for(let key in c) {
	console.log(key); // radius, filled, move
}
```

Class Hoisting

- Function declarations are hoisted, function expressions are not.
- Classes can be defined as declarations or expressions

Example

```
// Class Declaration (hoisted) (used more frequently)
class Circle {
}
// Class Expression (not hoisted)
const Square = class {
};
```

Static Methods

- The `static` keywork defines a static method or property for a class. Neither static methods nor static properties can be called on instances of a class, instead they are called on the class itself.

Example

```
class Circle {
      constructor(radius) {
            this.radius = radius;
      }

      // Instance Method
      draw() {
            console.log('draw');
      }

      // Static Method
      static parse(str) {
            const radius = JSON.parse(str).radius;
            return new Circle(radius);
      }
}

const circle = Circle.parse('{ "radius": 1 }');
console.log(circle); // Circle { radius: 1 }
```

This

- A function’s this keyword behaves a little differently in JS compared to other languages, including differences between strict mode and non-strict mode.
- In most cases, the value of this is determined by how a function is called (runtime binding). It can’t be set by assignment during execution, and it may be different each time the function is called.
- Note: Arrow functions don’t provide their own this binding (it retains the this value of the enclosing lexical context).

Example

```
// Enable 'strict' mode (prevents us from accidentally modifying global object)
'use strict';

const Circle = function() {
      this.draw = function() { console.log(this); }
}

const c = new Circle();

// Method Call
c.draw(); // Circle { draw: [Function] }

const draw = c.draw;
console.log(draw); // [Function]

// Function Call (by default, 'this' points to window in browser or global in node)
draw(); // undefined

----------------------------------------------------------------------
// Body of classes are automatically in 'strict' mode
class Circle {
      draw() {
            console.log(this);
      }
}

const c = new Circle();
const draw = c.draw;
draw(); // undefined
```

Private Members Using Symbols

- Symbol is a built-in object whose constructor returns a symbol primitive that is guaranteed to be unique.

Example

```
// Create Symbols
const _radius = Symbol();
const _draw = Symbol();
const _roll = Symbol();
// Circle
class Circle {
      constructor(radius) {
            // private properties
            this[_radius] = radius;
            this[_roll] = function() {
                  console.log('roll');
            }
      }
      // private method
      [_draw]() {
            console.log('draw');
      }
}

const c = new Circle(25);
console.log(Object.getOwnPropertySymbols(c)); // [ Symbol(), Symbol() ]
console.log(c); // Circle { [Symbol()]: 25, [Symbol()]: [Function] }
```

Private Members Using WeakMaps

- 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.

Example

```
const _radius = new WeakMap();
const _move = new WeakMap();
class Circle {
      constructor(radius) {
            _radius.set(this, radius);
            _move.set(this, () => {
                  console.log('move', this);
            });
      }
      draw() {
            console.log(_radius.get(this));
            _move.get(this)();
            console.log('draw');
      }
}
const c = new Circle(25);
c.draw(); // 25, move Circle {}, draw
```

Getters And Setters for Classes

Example

```
const _radius = new WeakMap();
class Circle {
      constructor(radius) {
            _radius.set(this, radius);
      }
      get radius() {
            return _radius.get(this);
      }
      set radius(value) {
            if (value <= 0) throw new Error('invalid radius');
            _radius.set(this, value);
      }
}

const c = new Circle(1);
console.log(c.radius); // 1
c.radius = 20;
console.log(c.radius); // 20
```

Inheritance

Example

```
class Shape {
	constructor(color) {
		this.color = color;
	}
	move() {
		console.log('move');
	}
}
// Circle class inherits from Shape class
class Circle extends Shape {
	constructor(color, radius) {
		super(color);
		this.radius = radius;
	}
	draw() {
		console.log('draw'); // draw
		console.log('Color is ' + this.color); // Color is red
		console.log('Radius is ' + this.radius); // Radius is 1
	}
}
const c = new Circle('red', 1);
c.move(); // move
c.draw(); // draw, Color is red, Radius is 1
```

Method Overriding in Classes

Example

```
class Shape {
	move() {
		console.log('move');
	}
}
// Circle
class Circle extends Shape {
	// method overriding
	move() {
		console.log('circle move'); // circle move
		// access Shape's move() method (super)
		super.move(); // move
	}
}
const c = new Circle();
c.move(); // circle move, move
```

Table of Contents

Title Page --- 1
Adding or Removing Properties of Objects: delete ------------------ 2
Abstraction -- 2
Private Properties and Methods of Objects: closure ---------------- 3
Getters and Setters: Object.defineProperty() --------------------- 4
Object Prototypes: Object.getPrototypeOf() ----------------------- 5
Property Descriptors: getOwnPropertyOf(), defineProperty() ------- 6,7
Prototypes and Instance Members: .prototype. --------------------- 8
Iterating Prototypes and Instance Members: keys(), hasOwn() ------- 9
Prototypical Inheritance: Object.create(), constructor ------------ 10
Calling the Super Constructor: call() ---------------------------- 11
Method Overriding in Objects ------------------------------------- 12
Polymorphism --- 13
Mixins --- 14
ES6 Classes -- 15
Class Hoisting --- 16
Static Methods --- 16
This --- 17
Private Members Using Symbols: Symbol() -------------------------- 18
Private Members Using WeakMaps: WeakMap() ------------------------ 19
Getters and Setters for Classes ---------------------------------- 19
Inheritance: extends --- 20
Method Overriding in Classes: super ------------------------------ 20
Table of Contents -- 21