Chapter 22: Classes

Section 22.1: Class Constructor

The fundamental part of most classes is its constructor, which sets up each instance's initial state and handles any parameters that were passed when calling `new`.

It's defined in a `class` block as though you're defining a method named `constructor`, though it's actually handled as a special case.

```
class MyClass {
    constructor(option) {
        console.log(`Creating instance using ${option} option.`);
        this.option = option;
    }
}
```

Example usage:

```
const foo = new MyClass('speedy'); // logs: "Creating instance using speedy option"
```

A small thing to note is that a class constructor cannot be made static via the `static` keyword, as described below for other methods.

Section 22.2: Class Inheritance

Inheritance works just like it does in other object-oriented languages: methods defined on the superclass are accessible in the extending subclass.

If the subclass declares its own constructor then it must invoke the parents constructor via `super()` before it can access `this`.

```
class SuperClass {

    constructor() {
        this.logger = console.log;
    }

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

}

class SubClass extends SuperClass {

    constructor() {
        super();
        this.name = 'subclass';
    }

}

const subClass = new SubClass();

subClass.log(); // logs: "Hello subclass"
```

Section 22.3: Static Methods

Static methods and properties are defined on *the class/constructor itself*, not on instance objects. These are specified in a class definition by using the `static` keyword.

```
class MyClass {
    static myStaticMethod() {
        return 'Hello';
    }

    static get myStaticProperty() {
        return 'Goodbye';
    }
}

console.log(MyClass.myStaticMethod()); // logs: "Hello"
console.log(MyClass.myStaticProperty); // logs: "Goodbye"
```

We can see that static properties are not defined on object instances:

```
const myClassInstance = new MyClass();

console.log(myClassInstance.myStaticProperty); // logs: undefined
```

However, they *are* defined on subclasses:

```
class MySubClass extends MyClass {};

console.log(MySubClass.myStaticMethod()); // logs: "Hello"
console.log(MySubClass.myStaticProperty); // logs: "Goodbye"
```

Section 22.4: Getters and Setters

Getters and setters allow you to define custom behaviour for reading and writing a given property on your class. To the user, they appear the same as any typical property. However, internally a custom function you provide is used to determine the value when the property is accessed (the getter), and to perform any necessary changes when the property is assigned (the setter).

In a `class` definition, a getter is written like a no-argument method prefixed by the `get` keyword. A setter is similar, except that it accepts one argument (the new value being assigned) and the `set` keyword is used instead.

Here's an example class which provides a getter and setter for its `.name` property. Each time it's assigned, we'll record the new name in an internal `.names_` array. Each time it's accessed, we'll return the latest name.

```
class MyClass {
    constructor() {
        this.names_ = [];
    }

    set name(value) {
        this.names_.push(value);
    }

    get name() {
        return this.names_[this.names_.length - 1];
    }
}
```

```
const myClassInstance = new MyClass();
myClassInstance.name = 'Joe';
myClassInstance.name = 'Bob';

console.log(myClassInstance.name); // logs: "Bob"
console.log(myClassInstance.names_); // logs: ["Joe", "Bob"]
```

If you only define a setter, attempting to access the property will always return **`undefined`**.

```
const classInstance = new class {
    set prop(value) {
        console.log('setting', value);
    }
};

classInstance.prop = 10; // logs: "setting", 10

console.log(classInstance.prop); // logs: undefined
```

If you only define a getter, attempting to assign the property will have no effect.

```
const classInstance = new class {
    get prop() {
        return 5;
    }
};

classInstance.prop = 10;

console.log(classInstance.prop); // logs: 5
```

Section 22.5: Private Members

JavaScript does not technically support private members as a language feature. Privacy - described by Douglas Crockford - gets emulated instead via closures (preserved function scope) that will be generated each with every instantiation call of a constructor function.

The `Queue` example demonstrates how, with constructor functions, local state can be preserved and made accessible too via privileged methods.

```
class Queue {

  constructor () {                  // - does generate a closure with each instantiation.

    const list = [];                // - local state ("private member").

    this.enqueue = function (type) {  // - privileged public method
                                    //   accessing the local state
      list.push(type);              //   "writing" alike.
      return type;
    };
    this.dequeue = function () {      // - privileged public method
                                    //   accessing the local state
      return list.shift();          //   "reading / writing" alike.
    };
  }
}
```

```
var q = new Queue;            //
                              //
q.enqueue(9);                 // ... first in ...
q.enqueue(8);                 //
q.enqueue(7);                 //
                              //
console.log(q.dequeue());     // 9 ... first out.
console.log(q.dequeue());     // 8
console.log(q.dequeue());     // 7
console.log(q);               // {}
console.log(Object.keys(q));  // ["enqueue","dequeue"]
```

With every instantiation of a `Queue` type the constructor generates a closure.

Thus both of a `Queue` type's own methods `enqueue` and `dequeue` (see `Object.keys(q)`) still do have access to `list` that continues to *live* in its enclosing scope that, at construction time, has been preserved.

Making use of this pattern - emulating private members via privileged public methods - one should keep in mind that, with every instance, additional memory will be consumed for every *own property* method (for it is code that can't be shared/reused). The same is true for the amount/size of state that is going to be stored within such a closure.

Section 22.6: Methods

Methods can be defined in classes to perform a function and optionally return a result.
They can receive arguments from the caller.

```
class Something {
    constructor(data) {
        this.data = data
    }

    doSomething(text) {
        return {
            data: this.data,
            text
        }
    }
}

var s = new Something({})
s.doSomething("hi") // returns: { data: {}, text: "hi" }
```

Section 22.7: Dynamic Method Names

There is also the ability to evaluate expressions when naming methods similar to how you can access an objects' properties with [ ]. This can be useful for having dynamic property names, however is often used in conjunction with Symbols.

```
let METADATA = Symbol('metadata');

class Car {
  constructor(make, model) {
    this.make = make;
    this.model = model;
  }

  // example using symbols
```

```
  [METADATA]() {
    return {
      make: this.make,
      model: this.model
    };
  }

  // you can also use any javascript expression

  // this one is just a string, and could also be defined with simply add()
  ["add"](a, b) {
    return a + b;
  }

  // this one is dynamically evaluated
  [1 + 2]() {
    return "three";
  }
}

let MazdaMPV = new Car("Mazda", "MPV");
MazdaMPV.add(4, 5); // 9
MazdaMPV[3](); // "three"
MazdaMPV[METADATA](); // { make: "Mazda", model: "MPV" }
```

Section 22.8: Managing Private Data with Classes

One of the most common obstacles using classes is finding the proper approach to handle private states. There are 4 common solutions for handling private states:

Using Symbols

Symbols are new primitive type introduced on in ES2015, as defined at MDN

> A symbol is a unique and immutable data type that may be used as an identifier for object properties.

When using symbol as a property key, it is not enumerable.

As such, they won't be revealed using `for var in` or `Object.keys`.

Thus we can use symbols to store private data.

```
const topSecret = Symbol('topSecret'); // our private key; will only be accessible on the scope of
the module file
export class SecretAgent{
    constructor(secret){
        this[topSecret] = secret; // we have access to the symbol key (closure)
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(topSecret[topSecret]); // we have access to topSecret
        };
    }
}
```

Because `symbols` are unique, we must have reference to the original symbol to access the private property.

```
import {SecretAgent} from 'SecretAgent.js'
```

```
const agent = new SecretAgent('steal all the ice cream');
// ok let's try to get the secret out of him!
Object.keys(agent); // ['coverStory'] only cover story is public, our secret is kept.
agent[Symbol('topSecret')]; // undefined, as we said, symbols are always unique, so only the
original symbol will help us to get the data.
```

But it's not 100% private; let's break that agent down! We can use the `Object.getOwnPropertySymbols` method to get the object symbols.

```
const secretKeys = Object.getOwnPropertySymbols(agent);
agent[secretKeys[0]] // 'steal all the ice cream' , we got the secret.
```

Using WeakMaps

`WeakMap` is a new type of object that have been added for es6.

As defined on MDN

> 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.

Another important feature of `WeakMap` is, as defined on MDN.

> The key in a WeakMap is held weakly. What this means is that, if there are no other strong references to the key, the entire entry will be removed from the WeakMap by the garbage collector.

The idea is to use the WeakMap, as a static map for the whole class, to hold each instance as key and keep the private data as a value for that instance key.

Thus only inside the class will we have access to the `WeakMap` collection.

Let's give our agent a try, with `WeakMap`:

```
const topSecret = new WeakMap(); // will hold all private data of all instances.
export class SecretAgent{
    constructor(secret){
        topSecret.set(this,secret); // we use this, as the key, to set it on our instance private
data
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(topSecret.get(this)); // we have access to topSecret
        };
    }
}
```

Because the const `topSecret` is defined inside our module closure, and since we didn't bind it to our instance properties, this approach is totally private, and we can't reach the agent `topSecret`.

Define all methods inside the constructor

The idea here is simply to define all our methods and members inside the constructor and use the closure to access private members without assigning them to **`this`**.

```
   export class SecretAgent{
```

```
        constructor(secret){
            const topSecret = secret;
            this.coverStory = 'just a simple gardner';
            this.doMission = () => {
                figureWhatToDo(topSecret); // we have access to topSecret
            };
        }
    }
```

In this example as well the data is 100% private and can't be reached outside the class, so our agent is safe.

Using naming conventions

We will decide that any property who is private will be prefixed with `_`.

Note that for this approach the data isn't really private.

```
export class SecretAgent{
    constructor(secret){
        this._topSecret = secret; // it private by convention
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(this_topSecret);
        };
    }
}
```

Section 22.9: Class Name binding

ClassDeclaration's Name is bound in different ways in different scopes -

1. The scope in which the class is defined - **`let`** binding
2. The scope of the class itself - within { and } in `class {}` - **`const`** binding

```
class Foo {
  // Foo inside this block is a const binding
}
// Foo here is a let binding
```

For example,

```
class A {
  foo() {
    A = null; // will throw at runtime as A inside the class is a `const` binding
  }
}
A = null; // will NOT throw as A here is a `let` binding
```

This is not the same for a Function -

```
function A() {
  A = null; // works
}
A.prototype.foo = function foo() {
  A = null; // works
}
A = null; // works
```