ES6 | Part 2

Agenda

- Classes
- Inheritance
- Module
- Import, export
- Map
- Set
- Babel
- Exercise

Classes

In object-oriented programming, a class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).

Classes

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

Class declarations

```
class Record {
        constructor(name, age) {
        this.age = age;
            this.name = name;
    }
}
```

Class Expressions

```
let test = class Record {
        constructor(name, age) {
        this.name = name;
        this.age = age;
        }
};
```

Class

An important difference between function declarations and class declarations is that function declarations are hoisted and class declarations are not.

You first need to declare your class and then access it, otherwise code like the following will throw a ReferenceError:

```
const p = new Record(); // ReferenceError

class Record {}
```

It throws the error as the class declarations are not hoisted

```
const val = returnFun(); // No error

function returnFun(){ return 'No error occured'}
```

It does not throws the error as the function declarations are hoisted

Class body and method definitions

Constructor

The constructor method is a special method for creating and initializing an object created with a class. There can only be one special method with the name "constructor" in a class. A SyntaxError will be thrown if the class contains more than one occurrence of a constructor method.

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

Prototype Methods

```
class Records {
 constructor(name,age) {
    this.name = name; this.age = age;}
  // Getter
  get info() {
    return this.createInfo(); }
  // Method
  createInfo() {
    return `${this.name} is ${this.age} years old`;}
}
const information = new Record('TTN', 10);
console.log(information.info); // TTN is 10 years old
```

Static Methods

The static keyword defines a static method for a class. Static methods are called without instantiating their class and cannot be called through a class instance. Static methods are often used to create utility functions for an application.

```
class Foo(){
   static methodName(){
      console.log("bar")
   }
}
```

Static Methods

Since these methods operate on the class instead of instances of the class, they are called on the class.

There are two ways to call static methods:

```
Foo.methodName()
// calling it explicitly on the Class name
// this would give you the actual static value.
```

```
this.constructor.methodName()
// calling it on the constructor property of the class
// this might change since it refers to the class of the current
instance, where the static property could be overridden
```

Public field declaration

```
class Rectangle {
 height = 0;
 width;
 constructor(height, width) {
  this.height = height;
  this.width = width;
 }
}
```

By declaring fields up-front, class definitions become more self-documenting, and the fields are always present.

Private field declaration

```
class Rectangle {
 #height = 0;
 #width;
 constructor(height, width) {
  this.#height = height;
  this.#width = width;
 }
}
```

Private fields cannot be created later through assigning to them, the way that normal properties can.

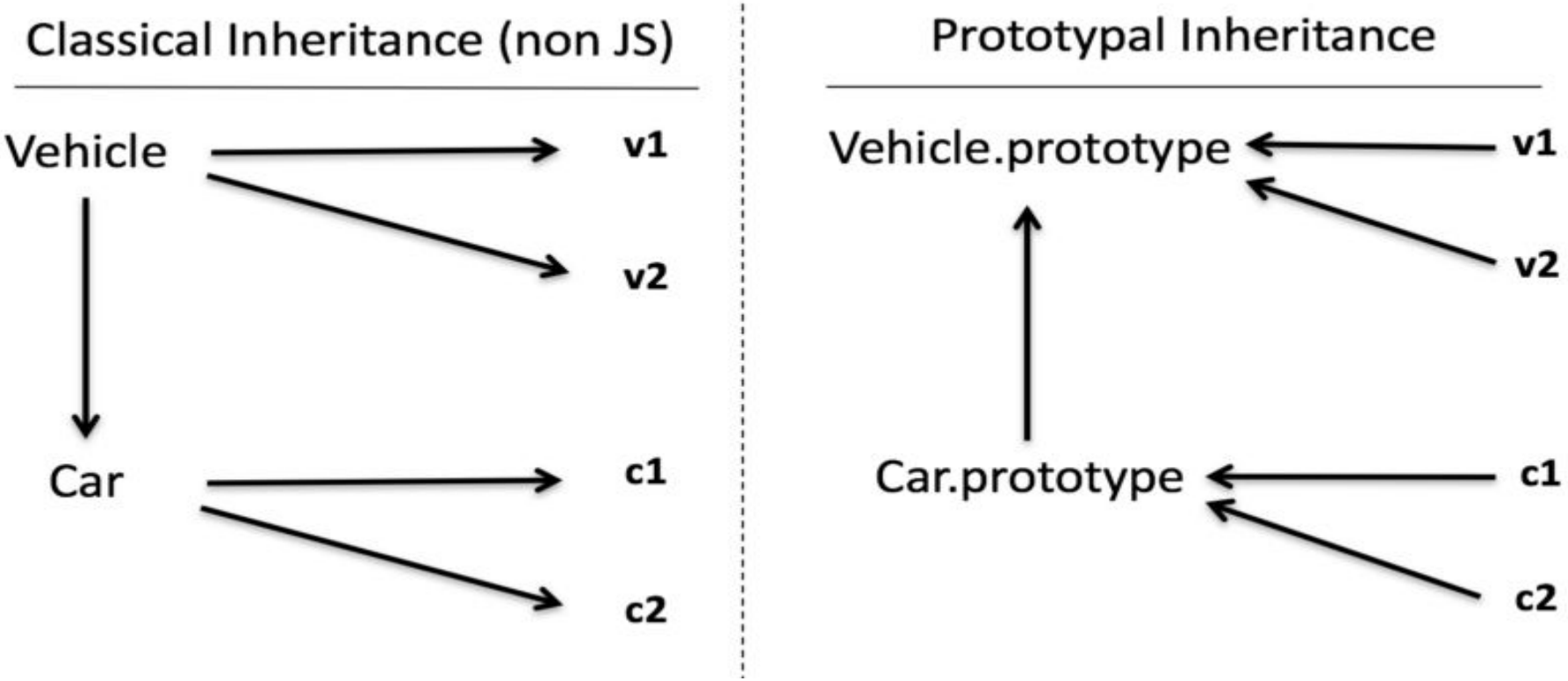
Private field declaration

It's an error to reference private fields from outside of the class; they can only be read or written within the class body. By defining things which are not visible outside of the class, you ensure that your classes' users can't depend on internals, which may change version to version.

Inheritance

Inheritance in most class-based object-oriented languages is a mechanism in which one object acquires all the properties and behaviors of another object. JavaScript is not a class-based language although *class* keyword is introduced in ES2015, it is just syntactical layer. JavaScript still works on *prototype* chain.

Classical vs Prototypal Inheritence



Classical Inheritance

- `Vehicle` is parent class and `v1`,`v2` are the instances of `Vehicle`
- `Car` is child class of `Vehicle` and `c1`, `c2` are instances of `Car`
- In classical inheritance it creates a copy of the behavior from parent class into the child when we extend the class and after that parent and child *class* are separate entity.
- Similarly, another copy of the behavior happens when we create an instance or object out of the class and both are separate entity again.
- It's like car is manufactured from the vehicle and car blueprints but after that both are separate entity because they are not linked because It's a copy. That's the reason all arrows going downwards (property and behavior flowing down)

Prototypal Inheritance

- `v1` and `v2` are linked to `Vehicle.prototype` because it's been created using *new* keyword.
- Similarly, `c1` and `c2` is linked to `Car.prototype` and `Car.prototype` is linked to `Vehicle.prototype`.
- In JavaScript when we create the object it does not copy the properties or behavior, it creates a link. Similar kind of linkage gets created in case of extend of class as well.
- All arrows go in the opposite direction compare to classical non-js inheritance because it's a behavior delegation link. These links are known as prototype chain.
- This pattern is called *Behavior Delegation Pattern* which commonly known as ***prototypal inheritance*** in JavaScript.

Modules

Almost every language has a concept of *modules* — a way to include functionality declared in one file within another. Typically, a developer creates an encapsulated library of code responsible for handling related tasks. That library can be referenced by applications or other modules.

Advantages

1. The same modules can be shared across any number of applications.
2. Ideally, modules need never be examined by another developer, because they've has been proven to work.
3. Code referencing a module understands it's a dependency. If the module file is changed or moved, the problem is immediately obvious.
4. Module code (usually) helps eradicate naming conflicts. Function `x()` in module1 cannot clash with function `x()` in module2. Options such as namespacing are employed so calls become `module1.x()` and `module2.x()`.

What are modules in JS?

Anyone starting web development a few years ago would have been shocked to discover there was no concept of modules in JavaScript. It was impossible to directly reference or include one JavaScript file in another.

Multiple <script> tags

HTML can load any number JavaScript files using multiple `<script>` tags

```
<script src="lib1.js"></script>
<script src="lib2.js"></script>
<script src="core.js"></script>
```

The average web page in 2018 uses 25 separate scripts, yet it’s not a practical solution:

Multiple <script> tags issues

Each script initiates a new HTTP request, which affects page performance. HTTP/2alleviates the issue to some extent, but it doesn't help scripts referenced on other domains such as a CDN.

- Every script halts further processing while it's run.
- Dependency management is a manual process. In the code above, if `lib1.js` referenced code in `lib2.js`, the code would fail because it had not been loaded. That could break further JavaScript processing.
- Functions can override others unless appropriate module patterns are used. Early JavaScript libraries were notorious for using global function names or overriding native methods.

Script Concatenation

One solution to problems of multiple `<script>` tags is to concatenate all JavaScript files into a single, large file. This solves some performance and dependency management issues, but it could incur a manual build and testing step.

Module Loader

Systems such as RequireJS and SystemJS provide a library for loading and namespacing other JavaScript libraries at runtime. Modules are loaded using Ajax methods when required. The systems help, but could become complicated for larger code bases or sites adding standard `<script>` tags into the mix.

Module Bundler, Preprocessors and Transpilers

- Bundlers introduce a compile step so JavaScript code is generated at build time.
- Processing is automated so there's less chance of human error.
- Further processing can lint code, remove debugging commands, minify the resulting file, etc.
- Transpiling allows you to use alternative syntaxes such as TypeScript or CoffeeScript.

ES6 Modules

- CommonJS — the `module.exports` and `require` syntax used in Node.js
- Everything inside an ES6 module is private by default, and runs in strict mode (there's no need for `'use strict'`). Public variables, functions and classes are exposed using `export`

Export

```
export const PI = 3.1415926;
export function sum(...args) {
 log('sum', args);
 return args.reduce((num, tot) => tot + num);
}
export function mult(...args) {
 log('mult', args);
 return args.reduce((num, tot) => tot * num);
}
// private function
function log(...msg) {
 console.log(...msg);
}
```

Import

```
// main.js

import { sum } from './lib.js';


console.log( sum(1,2,3,4) ); // 10
```

imports can be aliased to resolve naming collisions:

```
import { sum as addAll, mult as multiplyAll } from './lib.js';

console.log( addAll(1,2,3,4) );      // 10
console.log( multiplyAll(1,2,3,4) ); // 24
```

Map

The `Map` object holds key-value pairs and remembers the original insertion order of the keys.Any value (both objects and primitive values) may be used as either a key or a value

Syntax: `new Map([iterable])`

A `Map` object iterates its elements in insertion order — a `for...of` loop returns an array of `[key, value]` for each iteration.

Map vs Object

The keys of an `Object` are `Strings` and `Symbols`, whereas they can be any value for a `Map`, including functions, objects, and any primitive.

- The keys in Map are ordered while keys added to object are not. Thus, when iterating over it, a Map object returns keys in order of insertion.
- You can get the size of a `Map` easily with the `size` property,while the number of properties in an `Object` must be determined manually.
- A `Map` is an iterable and can thus be directly iterated, whereas iterating over an `Object` requires obtaining its keys in some fashion and iterating over them.
- An `Object` has a prototype, so there are default keys in the map that could collide with your keys if you're not careful. As of ES5 this can be bypassed by using `map = Object.create(null)`, but this is seldom done.
- A `Map` may perform better in scenarios involving frequent addition and removal of key pairs.

Map Functions

- `new Map()` – creates the map.
- `map.set(key, value)` – stores the value by the key.
- `map.get(key)` – returns the value by the key, `undefined` if `key` doesn't exist in map.
- `map.has(key)` – returns `true` if the `key` exists, `false` otherwise.
- `map.delete(key)` – removes the value by the key.
- `map.clear()` – clears the map
- `map.size` – returns the current element count.
- `map.keys()` – returns an iterable for keys,
- `map.values()` – returns an iterable for values,
- `map.entries()` – returns an iterable for entries `[key, value]`, it's used by default in `for..of`.

How map compares keys?

`Map` uses the algorithm SameValueZero. It is roughly the same as strict equality `===`, but the difference is that `NaN` is considered equal to `NaN`. So `NaN` can be used as the key as well. This algorithm can't be changed or customized.

Set

A `Set` is a collection of values, where each value may occur only once.

Syntax: `new Set(iterable)`

Set Functions

`new Set(iterable)` – creates the set, optionally from an array of values (any iterable will do).

- `set.add(value)` – adds a value, returns the set itself.
- `set.delete(value)` – removes the value, returns `true` if `value` existed at the moment of the call, otherwise `false`.
- `set.has(value)` – returns `true` if the value exists in the set, otherwise `false`.
- `set.clear()` – removes everything from the set.
- `set.size` – is the elements count.

Babel

Babel is a Javascript compiler or transpiler.

Babel Set Up

1. npm install -g babel-cli
2. npm install -g babel-preset-es2015
3. echo ‘{ “presets”:[“es2015”]}’> .babelrc
4. babel <filename>
5. babel <filename> --out-file <output file>
6. babel <filename> --watch --out-file <output file>

Exercise

1. Filter unique array members using Set.
2. Filter anagrams using Map.
3. Write a program to implement inheritance upto 3 classes.The Class must contain private and public variables and static functions.
4. Write a program to implement a class having static functions
5. Import a module containing the constants and method for calculating area of circle, rectangle, cylinder.
6. Import a module for filtering unique elements in an array.
7. Write a program to flatten a nested array to single level using arrow functions.
8.

Exercise