ECMAScript 2015

Les objets

Quelques nouveaux raccourcis

Ajouter une méthode

Avant

```
var obj = {
  myFunction: function() {}
};
```

Maintenant

```
var obj = {
  myFunction() {}
};
```

Propriétés homonymes

Avant

```
var obj = {
  attribute: attribute
};
```

Maintenant

```
var obj = {
  attribute
};
```

➔Ne fonctionne que si la propriété et la variable ont le même nom

Propriétés calculées

Avant

```
var prop = "attribute";
var obj = {};

obj[prop] = "test";
```

Maintenant

```
var prop = "attribute";

var obj = {
   [prop]: "test",
   [prop + '1']: "test2"
};

obj.attribute; // "test"
obj.attribute1; // "test2"
```

Marche pour les méthode

```
const obj = {
   ['h'+'ello']() {
     return 'hi';
   }
};
console.log(obj.hello()); // hi
```

Nouvelles APIs

Fusionner des objets

```
Object.assign(target,...objets)
```

Permet de fusionner les objets passés en paramètres avec le premier

Retourne l'objet target

Fusionne dans l'ordre des arguments

Si une propriété est définie dans plusieurs objets, le dernier gagne

Ne fusionne que les propriétés non héritées et énumérables

```
Object.assign({},myObject); // simple clone
Object.assign(target,{newProp:true}); // extends
Object.assign({}, DEFAULTS, options);
```

Comparaison d'objets

```
Object.is(value1, value2)
```

Equivalent au === mais fonctionne avec NaN et +0/-0

```
NaN === NaN // false
+0 === -0 // false

Object.is(NaN,NaN); // true
Object.is(+0,-0); // true
```

Introspection

```
Object.getOwnPropertyNames(obj)
```

Retourne toutes les nom de propriétés avec comme ordre : les clefs représentées par un entier, les clefs représentées par une chaine, les symboles

```
Object.getOwnPropertySymbols(obj)
```

Permet d'obtenir toutes les propriétés propres qui sont définies par un symbole

Exemple

```
const obj = {
   [Symbol('first')]: true,
   '02': true,
   '10': true,
   '01': true,
   '2': true,
   [Symbol('second')]: true,
};

Reflect.getOwnPropertyNames(obj);
// [ '2', '10', '02', '01',
//   Symbol('first'), Symbol('second') ]
```

Rappels de POO en JavaScript

Héritage en ES5

- JavaScript est un langage objet sans classes
- Il est basé sur la notion de **prototype**
- Il est possible de chainer les prototypes
- Un objet est une fonction constructeur utilisant **this** pour définir ses méthodes et attributs
- Un objet est instancié en utilisant **new** avant d'appeler la fonction constructeur
- Il est également possible de définir des objets littéraux pour simplifier la syntaxe

Exemple 1

```
var Parent = function() {
  this.attribute = "I'm the parent";

  this.hello = function() {
    return "Hello " + this.attribute;
  }
};

var mother = new Parent();
mother.hello(); // "Hello I'm the parent"
```

Exemple 2

```
var mother = {
  attribute: "I'm the parent",

  hello: function() {
    return "Hello " + this.attribute;
  }
};

mother.hello(); // "Hello I'm the parent"
```

Héritage

```
var Parent = {
  attribute: "I'm the parent",

  hello: function() {
    return "Hello " + this.attribute;
  }
};

var Child = function() {
  this.attribute = "I'm the child";
};

Child.prototype = Object.create(Parent);
Child.constructor = Child;

var boy = new Child();
boy.hello(); // "Hello I'm the child"
```

Utilisation des classes

Nouvelle syntaxe plus classique

```
class Parent {
  constructor() {
    this.attribute = "I'm a parent";
  }

  hello() {
    return `Hello ${this.attribute}`;
  }
}

class Child extends Parent {
  constructor() {
    super();
    this.attribute = "I'm a child";
  }
}

let boy = new Child();
boy.hello(); // "Hello I'm a child"
```

La théorie

- Nouveau mot clef **class** permettant de déclarer une nouvelle classe
- On hérite d'une autre classe avec **extends**
- Il n'y a pas d'héritage multiple
- Le constructeur est la méthode appelée **constructor**
- Les attributs sont définis dans le constructeur, seules les méthodes sont autorisées dans le corps de la classe
- Les méthode sont définies directement dans le corps de la classe sans séparateur ni mot clef
- Il n'y a pas de notion de visibilité (private, public, protected)
- Tout est public
- L'instanciation se fait toujours avec **new**

Définir des méthodes statiques

Avant :

```
var MyClass = function() {};

MyClass.myStaticMethod = function() {};

MyClass.myStaticMethod(); // Ok

var obj = new MyClass();
obj.myStaticMethod(); // KO
```

Maintenant :

```
class MyClass {
  static myStaticMethod() {}
}
```

La théorie

- Une méthode statique ne peut être appelée que sur la classe elle-même, pas sur ses instances
- Elle n'a pas accès au **this**
- En ES6 on la déclare en ajoutant le mot clef **static** avant le nom de la méthode

Privé via constructeur

```
class Countdown {
  constructor(counter, action) {
    Object.assign(this, {
      dec() {
        if (counter < 1) return;
        counter--;
        if (counter === 0) {
          action();
        }
      }
    });
  }
}
```

Privé via convention

```
class Countdown {
  constructor(counter, action) {
    this._counter = counter;
    this._action = action;
  }
  dec() {
    if (this._counter < 1) return;
    this._counter--;
    if (this._counter === 0) {
      this._action();
    }
  }
}
```

Privé via WeakMap

```
const _counter = new WeakMap();
const _action = new WeakMap();
class Countdown {
  constructor(counter, action) {
    _counter.set(this, counter);
    _action.set(this, action);
  }
  dec() {
    let counter = _counter.get(this);
    if (counter < 1) return;
    counter--;
    _counter.set(this, counter);
    if (counter === 0) {
      _action.get(this)();
    }
  }
}
```

Privé via Symboles

```
const _counter = Symbol('counter');
const _action = Symbol('action');

class Countdown {
  constructor(counter, action) {
    this[_counter] = counter;
    this[_action] = action;
  }
  dec() {
    if (this[_counter] < 1) return;
    this[_counter]--;
    if (this[_counter] === 0) {
      this[_action]();
    }
  }
}
```