Le Javascript / Les Objets

Parlons Objets

Rappelez-vous, tout ce qui n'est pas du type primitif est un **objet**. Et nous pouvons également créer les nôtres:

```
const voiture = {

}
```

ou

```
const voiture = new Object()
```

ou encore

```
const voiture = Object.create()
```

Vous pouvez également initialiser un objet en utilisant le mot-clé `new` avant une fonction avec une lettre majuscule. Cette fonction sert de constructeur pour cet objet. Là, nous pouvons initialiser les arguments que nous recevons en tant que paramètres, pour configurer l'état initial de l'objet:

```
function Voiture(marque, model){
    this.marque = marque
    this.model = model
}
```

On peut alors créer un objet Voiture:

```
const maVoiture = new Voiture('honda','civic')

maVoiture.marque // honda
maVoiture.model  // civic
```

Les objets sont **toujours passés par référence**.

Si vous affectez à une variable la même valeur qu'une autre, si c'est un type primitif comme un nombre ou une chaîne, elles sont passées par valeur :

Prenons cet exemple:

```
let age = 36
let myAge = age
myAge = 37
age //36
const car = {
  color: 'blue'
}
const anotherCar = car
anotherCar.color = 'yellow'
car.color //'yellow
```

Même les tableaux ou les fonctions sont, sous la coupe, des objets, il est donc très important de comprendre comment ils fonctionnent.

Les propriétés

Les objets ont des propriétés, qui sont composées d'une étiquette associée à une valeur.

La valeur d'une propriété peut être de n'importe quel type, ce qui signifie qu'il peut s'agir d'un tableau, d'une fonction et même d'un objet, car les objets peuvent imbriquer d'autres objets.

Voici la syntaxe littérale d'objet que nous avons vue dans le chapitre précédent:

```
const voiture = {

}
```

Nous pouvons définir la couleur de la voiture comme ceci:

```
const voiture = {
    couleur: 'rouge'
}
```

Mais nous pouvons aussi utiliser une chaine comme nom de propriété à condition d'utiliser des guillemets ''.

```
const voiture = {
    couleur: 'rouge',
    'la couleur': 'vert'
}
```

Les caractères de nom de variable non valides incluent les espaces, les tirets et d'autres caractères spéciaux.

Comme vous le voyez, lorsque nous avons plusieurs propriétés, nous séparons chaque propriété par une virgule.

On peut récupérer la valeur d'une propriété en utilisant 2 syntaxes différentes.

La première est la notation par points:

```
voiture.couleur // 'rouge'
```

La seconde (qui est la seule que nous pouvons utiliser pour les propriétés avec des noms invalides), consiste à utiliser des crochets:

```
voiture['la couleur']  // 'vert'
```

Et comme nous l'avons dit nous pouvons mettre des objets dans des objets.

```
const voiture = {
    marque: {
```

```
        nom: 'honda`
    },

    couleur: 'noir'
}
```

Ici pour accéder à la marque, nous faisons:

```
voiture.marque.nom  // 'honda'
```

Nous pouvons modifier ensuite l'objet. Ajouter une propriété ou même en supprimer:

```
const voiture = {
    couleur: 'rouge'
}

voiture.couleur = 'blanc'

voiture.model = 'R6'

voiture.model  // 'R6'

delete voiture.model
```

Les méthodes

J'ai parlé des fonctions dans un chapitre précédent.

Les fonctions peuvent être assignées à une propriété de l'objet, et dans ce cas elles sont appelées méthodes.

Dans cet exemple, la propriété start a une fonction assignée, et nous pouvons l'invoquer en utilisant la syntaxe à points que nous avons utilisée pour les propriétés, avec les parenthèses à la fin:

```
const voiture = {
    marque: 'Honda',
    model: 'Civic',
    start: function() {
        console.log('Je démarre !!!')
    }
}

voiture.start()
```

A l'intérieur de la méthode ainsi définie nous avons accès à l'instance de l'objet avec le mot clé `this`.
Ici par exemple pour avoir accès à `model` nous utiliserons `this.model`.

```
const voiture = {
    marque: 'Honda',
    model: 'Civic',
    start: function() {
        console.log(`La ${this.marque} ${this.model} démarre !!!`)
```

```
    }
}

voiture.start()
```

Mais attention, ça ne fonctionne pas avec les fonctions fléchées. En effet, les fonctions fléchées ne sont pas liées à l'objet.

C'est la raison pour laquelle les fonctions régulières sont souvent utilisées comme méthodes objet.

Les Classes

Nous avons parlé des **objets**, qui sont l'une des parties les plus intéressantes de JavaScript.

Dans ce chapitre, nous allons remonter d'un niveau, en introduisant les **classes**.

Que sont les classes? Elles permettent de définir un modèle commun pour plusieurs objets.

Prenons un objet personne :

```
const personne {
    nom: 'Germain'
}
```

Nous pouvons créer une classe `Personne` (notez que l'on utilise une majuscule), qui a la propriété `nom`:

```
class Personne {
    nom
}
```

Maintenant, nous pouvons créer un objet `germain` comme ceci:

```
const germain = new Personne()
```

`germain` est appelé **instance** de Personne. On peut affecter une valeur à `nom`

```
germain.nom = 'Germain'
```

Tout se passe comme pour les objets. Cependant, il serait intéressant de pouvoir alimenter directement les propriétés à l'instanciation. C'est ce que l'on fait avec la méthode `constructor`:

```
class Personne {
    constructor(nom){
        this.nom = nom
    }

    salut() {
        return 'Salut, je suis' + this.nom
    }
}
```

Ainsi, nous pouvons instancier comme ceci:

```
const germain = new Personne('Germain')

germain.salut() // 'Salut, je suis germain'
```

LA méthode `salut()` est à l'instance de l'objet mais nous pouvons définir des méthodes attachées à la classe avec `static`:

```
class Personne {
    constructor(nom){
        this.nom = nom
    }

    salut() {
        return 'Salut, je suis' + this.nom
    }

    static salutGeneral(){
        return 'Salut !!!'
    }
}

Person.salutGeneral() // 'Salut !!!'
```

Héritages

Une **classe** peut **hériter** d'autres classes, et les objets initialisés à l'aide de cette classe héritent de toutes les méthodes des deux classes.

Supposons que nous ayons une classe `Personne`:

```
class Personne {
    constructor(nom){
        this.nom = nom
    }

    salut() {
        return 'Salut, je suis' + this.nom
    }
}
class Concepteur extends Personne {
    salut(){
        return super.salut()+' et aussi concepteur'
    }
}
```
