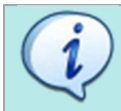
Le testing

Une manière d'éviter de devoir passer par un debugger serait de pouvoir s'assurer du bon fonctionnement du code rédigé avant toute chose. C'est la philosophie des tests unitaires : s'assurer que les méthodes écrites renvoient le résultat attendu, de manière unitaire (granulaire).

Et quand les tests sont rédigés avant même de coder, on appelle ça le TDD : *test-driven development*.



Plus sur les outils de tests unitaires pour du TDD en JavaScript :
http://stackoverflow.com/questions/300855/javascript-unit-test-tools-for-tdd
Introduction aux tests unitaires en JavaScript :
http://coding.smashingmagazine.com/2012/06/27/introduction-to-javascript-unit-testing/

Orienté objet

Nous avons abordé les objets plusieurs fois dans ce cours via les types objets, les objets du DOM, les objets littéraux utilisés comme paramètre dans des méthodes jQuery. Mais nous n'avons pas abordé comment crée nos objets nous-même ou ce à quoi ils pourraient servir.

Les objets sont un moyen d'organiser l'information de manière logique : les propriétés et méthodes d'un sujet sont regroupées, « encapsulées » dans un objet. Il y a 2 manière de définir des objets : soit de manière littérale - dans ce cas, l'objet n'est utilisable qu'une seule fois, soit via un constructeur – dans le cas plusieurs instances de cet objet peuvent être créées.

Objet littéral

Déclarer et utiliser un objet littéral

Les objets littéraux permettent de créer une liste de propriétés et valeurs en JavaScript. Ils sont beaucoup utilisés par exemple pour :

- paramétrer les plug-ins jQuery ou configurer les animations jQuery
- envoyer des données en AJAX
- structure du code

La syntaxe pour créer un objet littéral est celle-ci :

```
{
    'propriété1' : 'valeur1',
    'propriété2' : 'valeur2'
}
```

La propriété et la valeur sont séparés par : et chaque paire propriété/valeur doit être séparée par une virgule mais il ne fait pas en mettre après la dernière paire propriété/valeur ! Les apostrophes autour des propriétés sont facultatives sauf si les noms des propriétés contiennent des caractères spéciaux.

On peut créer un objet littéral et l'associer à une variable

```
var data = {
    firstName : 'Michel',
    lastName : 'Dupont',
    city : 'Bruxelles'
};
```

Et après on peut bien sûr récupérer chaque propriété individuellement :

```
console.log(data.firstName);
```

Voici un exemple d'utilisation d'objet littéral passé en paramètre de la méthode *animate()*

```
$('#menu').animate(
  {
     left : '0',
     fontSize : '15'
  }
);
```

Les objets littéraux peuvent être plus complexes : on peut des propriétés plus complexes comme des tableau ou d'autres objets littéraux mais également ajouter des méthodes.

```
var person = {
    firstName: "Georges",
    lastName: "Abitbol",
    age: 57,
    sentences: ["J'ai la classe", "Adieu monde de merde", "Allons
dans une bonne auberge"],
    saySomething: function(){
        alert(this.sentences[1]);
    }
};

person.saySomething();
```

Ici l'objet *person* possède plusieurs propriétés mais aussi une méthode *saySomething().* Remarquez comment *this* est utilisé pour accéder aux propriétés de l'objet, en effet *this* représente le contexte d'invocation de la méthode *saySomething(),* qui est l'objet dans lequel elle est définie.

Autre utilisation d'un objet littéral : le tableau associatif

Sous le capot, un objet littéral est en fait un Array (ou tableau) qu'on appelle tableau **associatif**.

Pour rappel, un Array est une variable qui contient plusieurs valeurs, appelées *items*. Chaque item est accessible au moyen d'un indice (*index*) numérique**.** Dans un tableau associatif, on accède à un

item non pas par un indice numérique mais pas une valeur textuelle appelée clé (ou *key*).

Les **propriétés** d'un objet littéral peuvent être donc vues comme des **clés** qui permettent d'accéder aux **valeurs** du tableau associatif. On pourrait donc écrire :

```js
var data = {
    firstName : 'Michel',
    lastName : 'Dupont',
    city : 'Bruxelles'
};

data['firstName'] = 'Jean';
```

Pour rajouter des éléments à ce tableau, il suffit simplement de spécifier une nouvelle propriété/clé en lui donnait une valeur, par exemple :

```js
data.country = 'Belgique';
```

Ou sous cette forme

```js
data['country'] = 'Belgique';
```

La boucle for in

Il existe une variante de la boucle for qui s'appelle *for in*. Cette boucle permet de boucler sur un tableau associatif (et donc par définition, sur un objet littéral !). Le fonctionnement est quasiment le même que pour un *Array*, excepté qu'ici il suffit de fournir une « variable clé » qui reçoit un identifiant (au lieu d'un index) et de spécifier l'objet à parcourir :

```js
for(var key in data) {
    console.log(data[key]);
}
```

Quel est l'intérêt ? Les objets littéraux se voient souvent attribuer des propriétés ou méthodes supplémentaires par certains navigateurs ou certains scripts tiers utilisés dans la page, ce qui fait que la boucle *for in* va permettre de tous les énumérer.

Constructeur

Si on veut représenter plusieurs fois le même objet, qui a des méthodes similaires mais avec des valeurs différentes pour ses propriétés, l'objet littéral n'est pas l'idéal car il faudra sans cesse copier-coller la structure pour chaque instance… dans ce cas, on peut passer par un constructeur pour créer un « canevas » général et instancier plusieurs entités à partir de ce canevas.
Ce système nous permet aussi de prévoir des valeurs par défaut pour les propriétés.

```
function Person(fname, lname, age){
    this.firstName = fname;
    this.lastName = lname;
    this.age = age;
    this.sentences = ["Hello world"];
    this.saySomething = function(sentence){
        if(!sentence){
            sentence = this.sentences[0];
        }
        alert(sentence);
    }
}

var georges = new Person("Georges", "Abitbol", 57);
georges.saySomething();
var beavis = new Person("Beavis", "", 15);
beavis.sentences.push("That sucks");
beavis.saySomething(beavis.sentences[1]);
```

Structurer son code grâce aux objets

Un des atouts majeurs de l'orienté objet, c'est de pouvoir nous aider à mieux structurer notre code, pour le rendre plus flexible, réutilisable, pouvoir isoler des variables et des méthodes et les rendre « privées », c'est-à-dire les rendre invisible par le code externe à l'objet.

Pour illustrer cette structuration, voici 1 seul et même exercice, codé de manière « classique » : tout dans l'espace global, aucune structuration ; et l'autre codé de manière « objet »

Contrôle « captcha » numérique, manière « classique » : http://jsfiddle.net/ZP3fR/
Contrôle « captcha » numérique, manière « objet » : http://jsfiddle.net/NH7uA/

Plus sur la programmation orienté-objet en JavaScript :
http://javascriptissexy.com/javascript-objects-in-detail/
http://javascriptissexy.com/oop-in-javascript-what-you-need-to-know/
http://eloquentjavascript.net/chapter8.html

Bonnes pratiques de développement

Isoler son code grâce au *module pattern*

On y réfléchit pas trop, mais dès qu'on créer une variable globale ou une fonction globale, on fait quelque chose de très dangereux et d'assez « sale » car on induit des risques de conflits avec d'autres variables ou fonctions. Qui plus est, la majorité de vos codes sont autonomes ou *self-contained*, bien souvent propres à une seule page.