Coder au Front (End)

Une formation d'introduction à la programmation de scripts exécutés dans le navigateur web...

... Ce qu'on appelle aussi parfois dans les milieux anglo-saxons
« client-side scripting[1] »...

... donnée les 10 et 11 septembre 2012 à Bordeaux par Jonathan Perret (jonathan@ut7.fr) & Emmanuel Gaillot (emmanuel@ut7.fr), sous la haute bénédiction de /ut7.

1 ... Mais ce qui ne s'abrège pas en CSS parce que si c'était le cas ça ficherait trop la grouille avec l'acronyme CSS qui veut dire « Cascading Style Sheet » et vu que c'est déjà bien le pastis tel quel, inutile d'en rajouter davantage...

Introduction

Il existe un plaisir quasi-rebelle à coder en JavaScript. Celui d'utiliser un langage dont la beauté étonnante s'ouvre à celui qui sait prendre le temps de la regarder. Celui d'utiliser des paradigmes audacieusement exotiques sous des apparences de déjà-vu. Celui enfin d'avoir le sentiment d'être un explorateur en milieu loin d'être conquis, sous ses allures trompeusement banales.

On se demande parfois pourquoi il est fait tout un pataquès de certaines choses qui paraissent à la fois touffues, complexes et rebutantes par ailleurs. Wagner. Le whisky. Le cigare. JavaScript rentre probablement dans cette catégorie. Le goût pour JavaScript est un goût qui s'éduque, un *acquired taste*, comme les Anglais disent à propos de la gastronomie du fromage et des escargots. Ce goût-là pour Javascript a un coût moindre à développer - c'est un langage gratuit qui s'exécute dans n'importe quel navigateur web un tant soit peu récent. Le coût restant, c'est le temps qu'il faut passer à se familiariser avec le langage, en en lisant et en en écrivant. C'est aussi l'énergie qu'il faut pour se lancer dans l'aventure, pour faire le premier pas sur une route inconnue et mal débroussaillée.

Cette formation se veut une invitation à faire le premier pas. Nous avons confiance qu'une fois celui-ci fait, les autres suivront – sinon sans peine, tout au moins avec plaisir.

Éléments de langage

Nous parlons, bien évidemment de JavaScript – d'une manière restrictive, châtiée, pour n'en garder que les « bons morceaux », ceux qui font la force du langage. Nous laissons délibérément de côté les morceaux de second choix, au point que nous n'en parlons même pas. Voici, à l'inverse, les joyaux de JavaScript que nous comptons contempler.

- **La programmation fonctionnelle –** JavaScript permet à ses fonctions de prendre des fonctions comme paramètres. Ou de retourner des fonctions en résultat d'exécution. Et ça a des impacts intéressants.
- **Les objets et l'héritage –** JavaScript, telle la lumière et sa dualité corpusculaire / ondulatoire, permet de décrire l'héritage de deux manières – pseudo-classique et prototypale.
- **Les mécanismes d'encapsulation –** JavaScript est fâcheusement porté sur la variable globale. C'est triste. Mais les mécanismes d'encapsulation permettent de faire des merveilles. Alors on est moins triste.

Pour une pratique heureuse de la programmation

Programmer est une tâche incroyablement complexe – qu'il s'agisse de JavaScript ou non. Modifier un programme, c'est un peu comme promener le chien : c'est un plaisir tant que ce n'est pas le chien qui nous promène. Parce qu'on n'aime pas être mené par le bout du nez par nos programmes, nous faisons en sorte d'en rester (davantage) les maîtres. En JavaScript comme ailleurs. Nous mettons l'accent dans cette formation sur trois techniques nous aidant à aller dans ce sens.

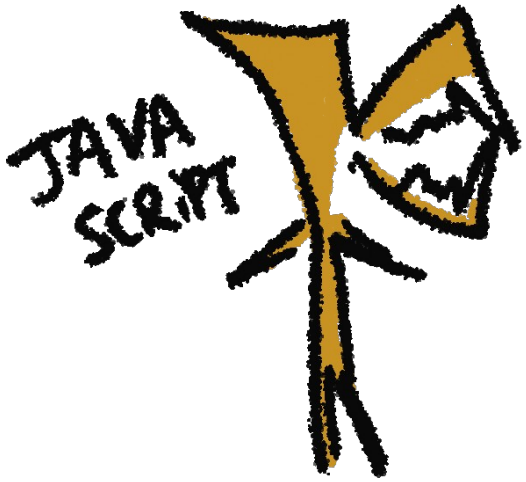
- **Développement piloté par les tests –** on construit au fur et à mesure du développement un harnais de tests automatisés, selon un cycle bien précis. Il existe de nombreux outils pour mettre en œuvre cette technique en JavaScript. Celui que nous utilisons ici est **Jasmine**.
- **Scripts discrets –** dit autrement, « *Unobtrusive JavaScript* ». Dans un souci d'ordre et de séparation slip / chaussettes, on ne mélange pas le HTML et les scripts. Cette technique repose sur une utilisation particulière des balises HTML (via les sélecteurs jQuery), et par ailleurs de la balise `<script>`.
- **Découpage en modules –** Inutile de tout mettre au même endroit ou dans le même fichier. Pensons réutilisation, pensons modularité.

Chaperons

Parce qu'il n'y a pas que des morceaux de premier choix en JavaScript, on pourrait par inattention croquer dans un truc un peu moisi. Ça serait dommage. D'autant plus qu'il y a largement assez de bons morceaux pour coder de manière intéressante en JavaScript. Nous nous aidons par conséquent d'un certain nombre d'outils pour augmenter nos chances de rester sur le chemin, loin des marais. Les outils en questions sont les suivants.

- **JSLint –** il s'agit d'un vérificateur de style intransigeant, à la limite du pénible, qui pointe nombre des fautes qu'on ne voudrait pas laisser dans le code qu'on écrit si notre maman devait le relire. C'est pénible, donc, mais c'est vraiment pour notre bien et on l'en remercie.
- **jQuery –** nous nous plaisons à imaginer jQuery comme la « stdlib » de JavaScript pour le navigateur : une bibliothèque standard de fonctions bas-niveau qui permettent au développeur de se concentrer sur la justesse du code qu'il écrit, plutôt que sur les problèmes de compatibilité entre les plates-formes et leurs failles. jQuery nous amènera à parler des sélecteurs CSS (*Cascading Style Sheets*)
- **CoffeeScript** – il s'agit d'un surlangage, dont la vocation est d'être un gigantesque sucre syntaxique pour implémenter de manière systématique les bons morceaux de JavaScript.

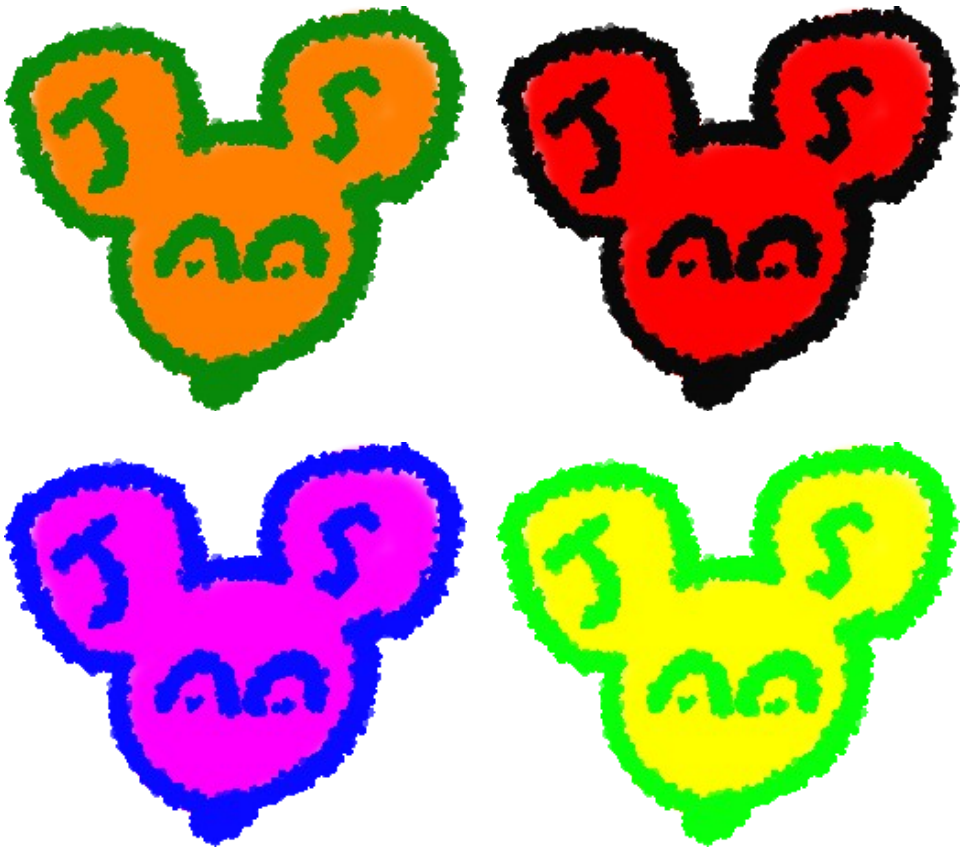
Il est délicat d'aborder chacun de ces sujets en isolation des autres. Pendant la formation, nous prenons la liberté de sauter d'un sujet à l'autre, d'un domaine à son voisin, au gré de ce qu'il nous paraît judicieux de clarifier au moment où le besoin se présente. Dans ce document en revanche, nous nous efforçons d'être plus linéaires, dans l'espoir que vous, joyeux participants, puissiez vous y reporter pendant les exercices, ou ultérieurement.



Grammaire JavaScript

Comme promis ailleurs, nous nous focalisons uniquement sur les « morceaux de premier choix » de JavaScript, nous faisant ainsi l'écho de Douglas Crockford dans son livre ***JavaScript: The Good Parts***[2].

Et comme nous ne voyons pas l'intérêt de dupliquer ce qui y est décrit avec une concision admirable, nous vous y renvoyons directement – en particulier au chapitre 2, entièrement dédié à la grammaire de JavaScript.



2 Crockford, Douglas, *JavaScript: The Good Parts*, ed. Yahoo ! Inc., 2008.

Certains langages autorisent que les fonctions soient considérées comme des valeurs. On peut alors les assigner à une variable, les passer en argument à des fonctions, ou les retourner en résultat d'exécution.

JavaScript est un de ces langages.

Ainsi :

```
var odette = function (x) {
        return 12 - x;
    },

    paulette = function (f) {
        return f(12);
    },

    georgette = function (x) {
        return function (y) { return y - x; };
    };
```

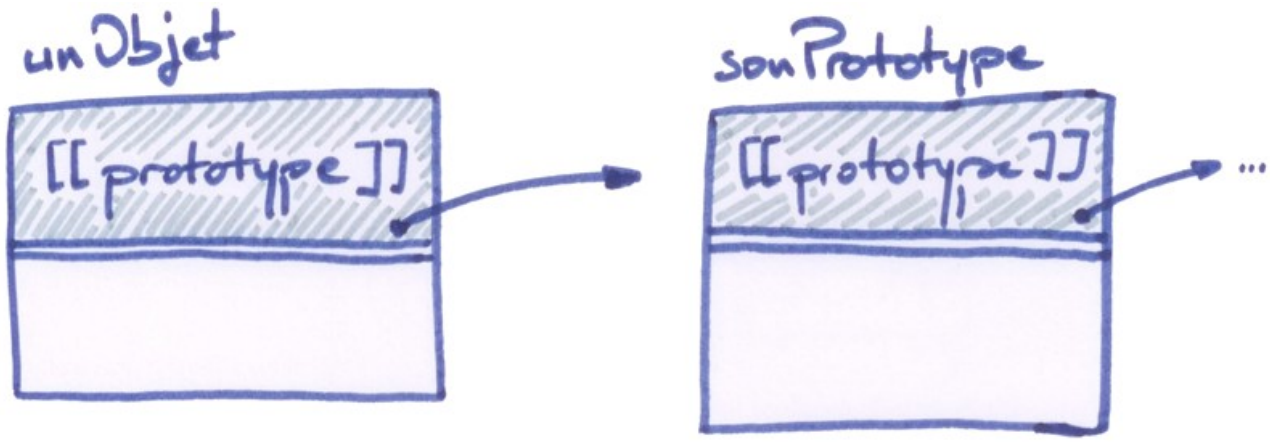
Pour aller plus loin...

- *JavaScript: The Good Parts*, chapitre 4.

Objets et héritage

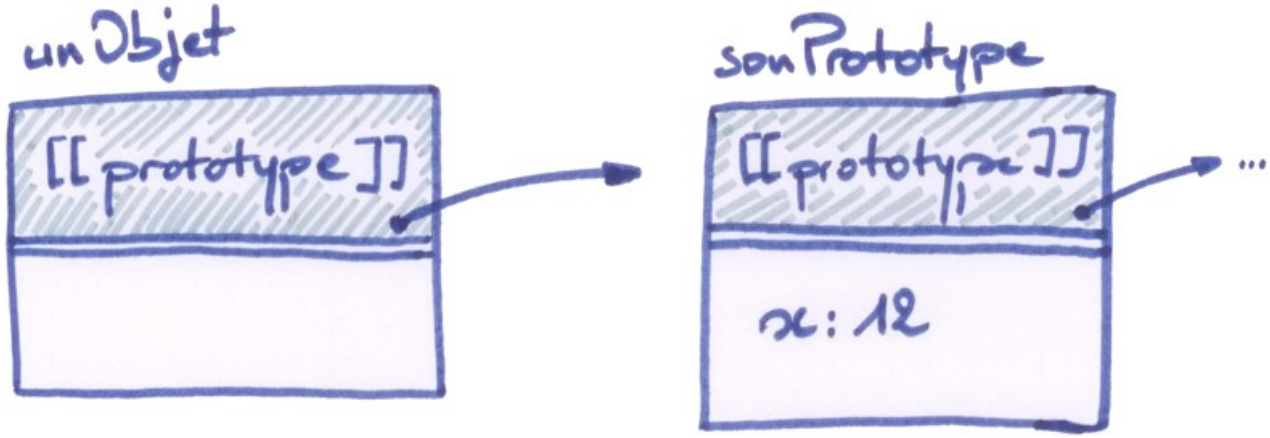
En JavaScript, les objets ont chacun un **prototype** auxquels ils peuvent accéder, mais qui n'est pas directement accessible depuis l'extérieur de l'objet, et qu'on note conventionnellement [[prototype]].

Le prototype d'un objet est lui-même un objet, qui a donc son propre prototype.



Le mécanisme d'héritage est le suivant – si on souhaite accéder à la propriété `x` d'un objet, et que cet objet n'a pas ladite propriété `x`, JavaScript cherche à l'obtenir de son prototype. Si celui-ci ne l'a pas, JavaScript cherche à l'obtenir du prototype de ce prototype, etc. jusqu'à arriver en haut de la chaîne prototypale. Si tout en haut la propriété n'est toujours pas connue, JavaScript renvoie `undefined`.

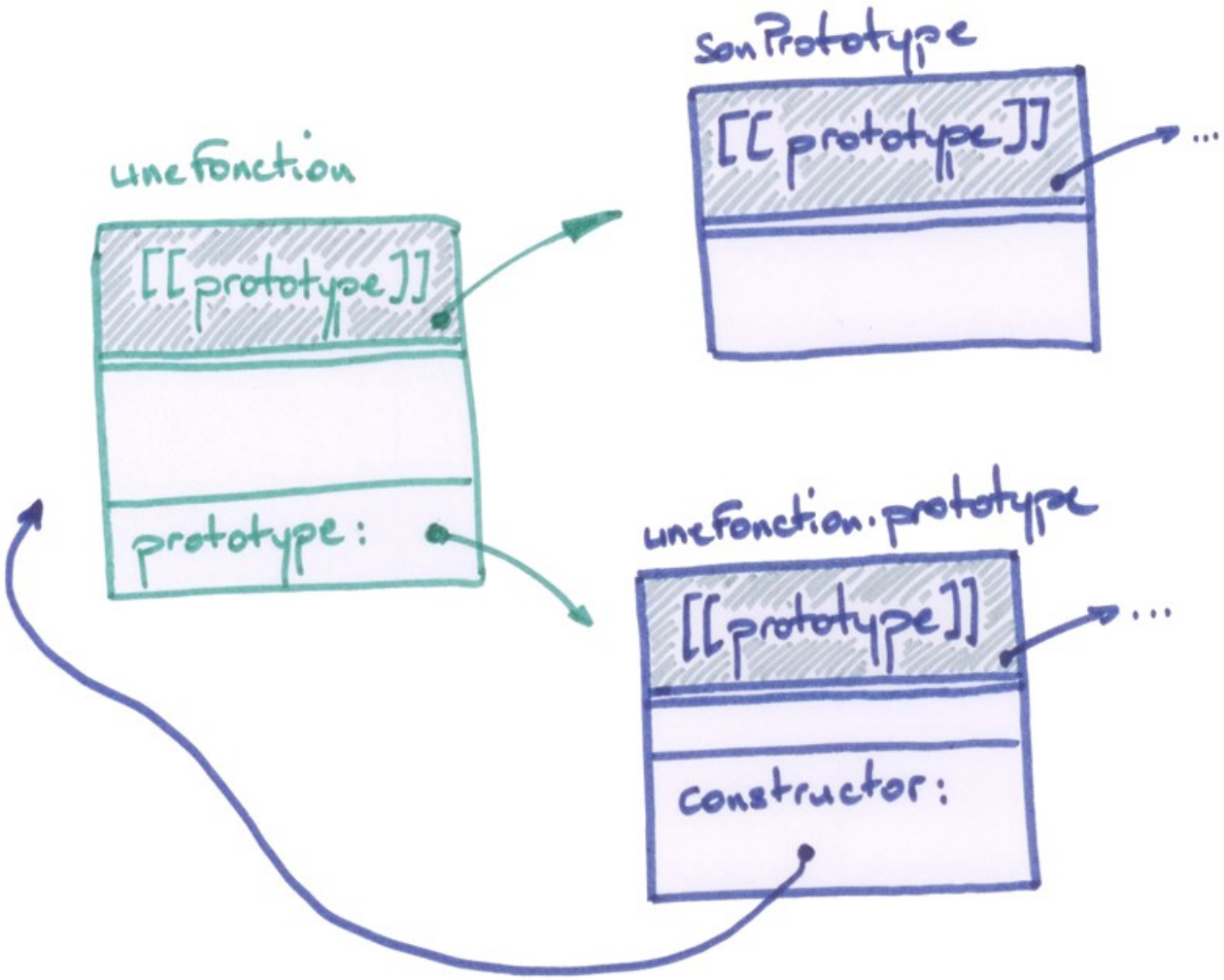
Ainsi, dans l'exemple suivant, `unObjet` n'a pas la propriété `x`, mais son prototype nommé `sonPrototype` l'a. Par conséquent, `unObjet.x` renvoie `12`.



Par ailleurs : les fonctions en JavaScript sont des objets. Elles ont par conséquent chacune également un prototype. *Au moment où la fonction est déclarée*, l'objet fonction se voit par ailleurs automatiquement muni d'une propriété accessible publiquement depuis l'extérieur de l'objet. Cette propriété a pour nom `prototype` (ce qui est probablement le nom le plus mal choisi de la Terre) et pointe vers un objet créé spécialement pour l'occasion. Ce dernier, bien sûr, comme tous les objets, a lui-même un prototype.

Afin de rendre les choses le moins obscur possible, on différenciera le *prototype* d'une fonction, de la *propriété prototype* de cette même fonction. **Les deux objets vers lequels pointent ces deux éléments sont différents.**

Toujours au moment où la fonction est déclarée, l'objet qui est assigné à la propriété `prototype` de la fonction se voit doté d'une propriété qui a pour nom `constructor`. **Cette propriété pointe vers la fonction nouvellement déclarée.**



So far, so good.

Nous en arrivons à l'instanciation d'un nouvel objet.

La manière la plus simple en JavaScript consiste à créer l'objet à partir d'un autre, qui devient son prototype. Cela se fait avec la méthode `Object.create`.

Ainsi :

```
var sonPrototype = {x: 12},
    unObjet = Object.create(sonPrototype);
var resultat = unObjet.x; //12
```

Ce paradigme d'héritage est appelé *héritage prototypal*. Bien qu'un peu exotique, il n'en reste pas moins très intuitif.

Une autre manière d'instancier un objet consiste à appeler une fonction, précédé du mot réservé `new`. Lorsqu'une fonction `F` est appelée précédée de `new`, il se passe trois choses :

- d'une part, JavaScript crée un objet vide dont le prototype est la propriété `prototype` de `F`
- d'autre part, l'objet créé est assigné à la variable `this` le temps de l'exécution de la fonction `F`
- enfin, la fonction renvoie `this` en fin d'exécution.

Ainsi :

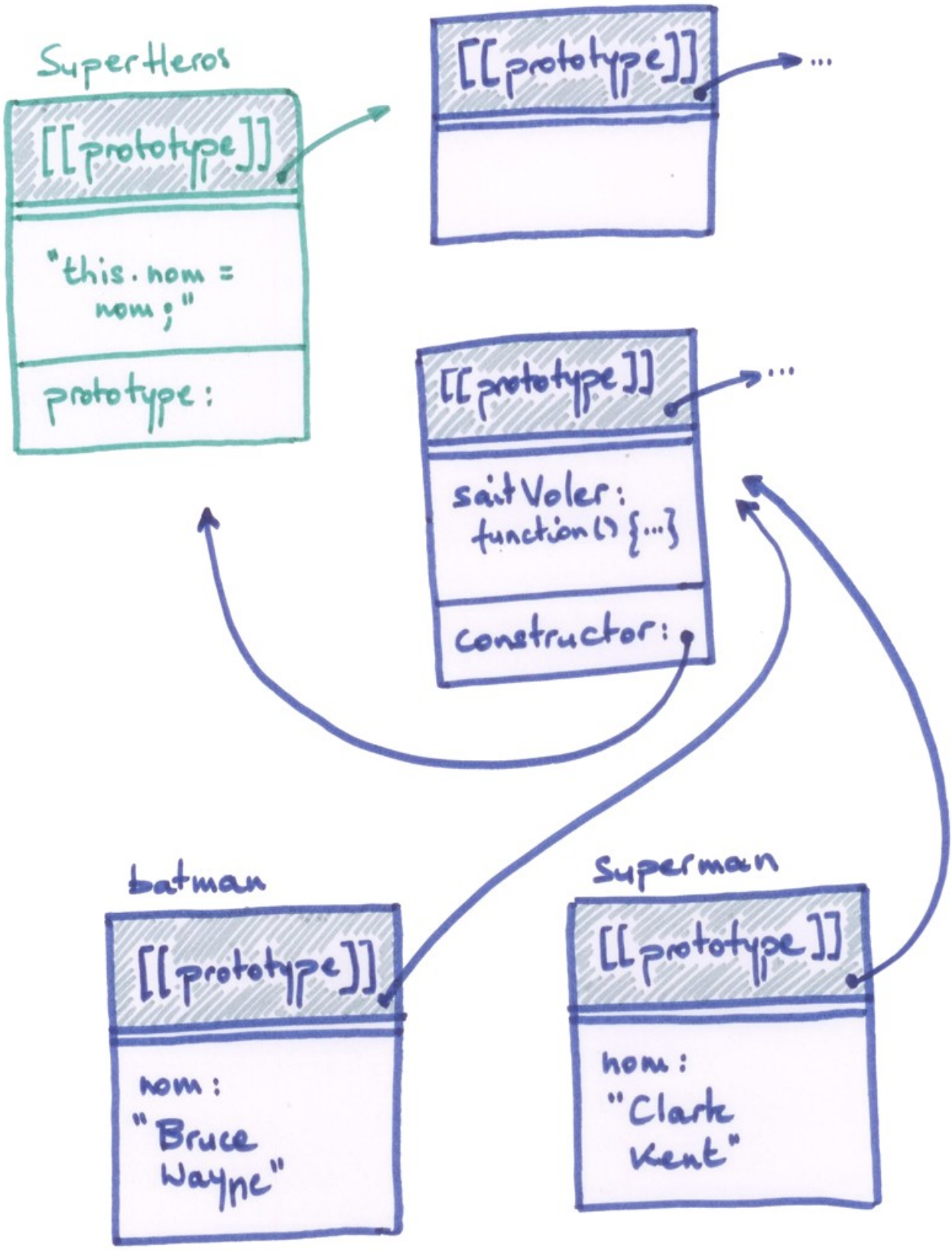
```
var SuperHeros = function (nom) {
        this.nom = nom;
    },
    batman = new SuperHeros("Bruce Wayne");
    superman = new SuperHeros("Clark Kent");

SuperHeros.prototype.saitVoler = function () {
    return true;
};

var r1 = batman.saitVoler(); //true
var r2 = superman.saitVoler(); //true
```

Si `F` est appelée sans être précédée de `new`, seules des catastrophes peuvent arriver. Aucun objet vide n'est créé, et this a pour valeur l'objet global. Là où on croit pointer vers un objet spécifique, on se retrouve en fait à pointer vers l'objet global, avec tous les risques d'anomalies difficiles à déceler que cela peut entraîner. Pour éviter ces confusions et par convention, les fonctions destinées à être appelées précédées de new ont un nom qui commence par une majuscule. Ces fonctions particulières sont appelées des *constructeurs*.

Et d'ailleurs, que vaut (dans l'exemple précédent) `batman.constructor` ? Cette propriété n'est pas dans l'objet `batman`. JavaScript cherche du coup à l'obtenir du prototype de `batman`. Par définition, il s'agit de `SuperHeros.prototype`. Est-ce que cet objet contient la propriété `constructor` ? Oui, par définition encore une fois. Et cette propriété pointe vers `SuperHeros`. Ainsi, la valeur de `batman.constructor` est `SuperHeros`.



Ce deuxième paradigme d'héritage est appelé *pseudo-classique*, en ce sens qu'il imite les mécanismes d'héritage par classes des langages orientés-objet... mais cette imitation peut être trompeuse. En effet, elle ne propose qu'un ensemble limité de fonctionnalités habituellement associées à l'héritage par classes. Malgré cela, parce qu'il s'agit d'un paradigme largement utilisé en JavaScript, il est important de le connaître et de le comprendre.

Pour aller plus loin :

- Il existe un troisième paradigme pour implémenter l'héritage en JavaScript, basé sur la programmation fonctionnelle. Ce paradigme est décrit dans *JavaScript: The Good Parts*, au chapitre 5.
- Toujours au chapitre 5 de ce même livre : la manière dont Douglas Crockford décrit le mécanisme d'héritage entre une classe parente et une classe fille n'est pas complet. Pour une version plus robuste d'implémentation de ce mécanisme, regarder comment CoffeeScript traduit `class A extends (class B).`

Mécanismes d'encapsulation

Où JavaScript stocke-t-il les variables déclarées ? Dans l'espace global. Ce qui en fait des variables globales... ce qui ne se fait pas chez les gens de bonne compagnie.

Comment faire, alors, pour limiter la portée des variables ?

Commençons par noter que la portée des variables déclarées à l'intérieur d'une fonction est limitée à cette fonction.

```
var a = 12,
    f = function () {
        var a = 49;
        return a;
    }
var r = a;   // 12
var s = f(); // 49
var t = a;   // 12
```

On peut ainsi définir une fonction englobant notre module, qu'on exécuterait tout de suite.

```
var fonctionEnglobante = function () {
    var a = 12;
    // imaginons ici du code, des choses,
    // des fonctionnalités incroyables utilisant a
}
fonctionEnglobante() ;
var r = a ; //undefined
```

... ce qu'on peut abréger de la manière suivante.

```
(function () {
    var a = 12;
    // imaginons ici du code, des choses,
    // des fonctionnalités incroyables utilisant a
}());
var r = a; //erreur : a n'est pas défini
```

Il n'y a plus besoin de nommer la fonction englobante, en revanche il faut l'entourer de parenthèses pour que JavaScript interprète correctement qu'il ne s'agit pas d'une simple déclaration de fonction. Ce mécanisme d'encapsulation, s'il est utilisé systématiquement pour englober le code d'un fichier, permet de modulariser l'ensemble du code, un module par fichier, et d'en limiter la portée.

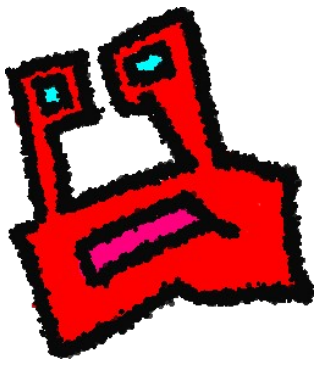
Problème toutefois : comment *exposer* l'interface de ce module pour qu'il puisse être accessible depuis un autre module ?

On utilise une variable globale – d'un nom ayant peu de probabilité d'exister, par convention tout en majuscules, puis on y déclare les éléments de l'interface du module, accessible depuis l'extérieur. Attention, si le module s'étend sur plusieurs fichier, il s'agit d'étendre le contenu de la variable globale, et non de la redéfinir.

```
var MON_MODULE = MON_MODULE || {};
(function () {
    var uneFonctionPrivée = function () {
            // ...
        },
        uneFonctionPublique = function () {
            return "Coucou!";
        }

    MON_MODULE.uneFonctionPublique = uneFonctionPublique;
}());

MON_MODULE.uneFonctionPublique(); // "Coucou !"
MON_MODULE.uneFonctionPrivée();   // erreur
```



Le cycle de développement piloté par les tests (*Test-Driven Development*, ou TDD) est le suivant.

1. Écrire un test qui échoue *pour les bonnes raisons*
2. Écrire le code qui fait passer ce test *et tous les autres pré-existants*
3. Nettoyer tout ce qui doit l'être, de manière à ce que
    - tous les tests continuent de passer
    - il n'y a pas de duplication
    - le code révèle son intention sans ambiguïté
    - la structure du code est la plus simple possible
4. Si l'on pense que le programme ne fait pas encore tout ce qu'il devrait faire, recommencer en 1.

L'outil que nous utilisons pendant la formation est Jasmine. Il s'agit d'un framework qui peut exécute les tests dans le navigateur web et produire un rapport indiquant si certains tests échouent, et si oui lesquels.

Une fois Jasmine installé dans un répertoire ad hoc du projet (par ex., `lib`) il faut configurer la page template `SpecRunner.html` pour y inclure d'une part les modules de tests spécifiques à l'application à tester, et les modules de l'application elle-même.

Un module de test ressemble typiquement à l'exemple qui suit.

```
(function () {
    describe("Une fixture", function() {
        var maFixture;
        beforeEach(function() {
            // initialise maFixture
        });
        it("présente un comportement donné", function() {
            expect(maFixture.comportement()).toEqual(unResultat);
        });
        it("présente un autre comportement donné", function() {
            expect(maFixture.autreComportement()).toEqual(autreResultat);
        });
    });
}));
```

Il est possible de créer de bouchonner un objet.

```
spyOn(objet, "nomMéthode");
```

Il est ensuite possible de vérifier que cette méthode nomMéthode a effectivement été appelée, avec les paramètres corrects.

```
expect(objet.nomMéthode).toHaveBeenCalledWith("les", "bons", "args");
```

Pour aller plus loin :

- Cf. documentation de Jasmine : https://github.com/pivotal/jasmine/wiki
- Dans un projet conséquent, Jasmine n'est pas utilisé directement, mais *via* un wrapper qui varie selon la nature du projet (Rails, node, etc.). Rechercher sur le web quel wrapper utiliser pour votre projet spécifique.

D'autres ressources

- Documentation CoffeeScript : http://coffeescript.org/
- Utiliser JSLint en local : https://github.com/reid/node-jslint
- Documentation jQuery : http://docs.jquery.com/

Pour le reste, un des cauchemars concernant JavaScript est que les résultats de recherche arrivant en tête de moteur de recherche ne sont pas forcément les plus sérieux. Une des documentations les plus précises et complètes sur le *Client-Side Scripting* est accessible sur MDN (*Mozilla Developer Network*). Pour obtenir des résultats de recherche pertinents, penser à mentionner le mot MDN dans la requête de recherche.

À titre d'exemple, on pourra comparer les résultats de recherche Google pour « JavaScript setTimeOut » et « MDN JavaScript setTimeOut »