Licence 1

Informatique, Développement d'Application

Cours : Introduction au JavaScript

Séquence 2 : Les objets, les tableaux et les fonctions

UVS
UNIVERSITÉ VIRTUELLE DU SÉNÉGAL

2. Les objets, les tableaux et les fonctions

2.1. Les objets

La programmation orientée objet a pour but de permettre une plus grande flexibilité et maintenabilité du code. Elle est populaire pour les projets logiciels de grande ampleur. Étant donné l'accent mis sur la modularité, le code orienté objet est censé être plus simple à développer, plus facile à reprendre, à analyser et permettre de répondre à des situations complexes en comparaison à d'autres méthodes de programmation moins modulaires.

Le type de données fondamental de JavaScript est l'objet. Un objet est une valeur composite: il agrège plusieurs valeurs (valeurs primitives ou autres objets) et vous permet de stocker et de récupérer ces valeurs par nom. Un objet est une collection non ordonnée de propriétés, chacune ayant un nom et une valeur. Les noms de propriété sont des chaînes, donc nous pouvons dire que les objets mappent les chaînes aux valeurs.

Un objet est plus qu'une simple association de chaîne à valeur, cependant. En plus de conserver son propre ensemble de propriétés, un objet JavaScript hérite également des propriétés d'un autre objet, appelé «prototype». Les méthodes d'un objet sont généralement des propriétés héritées, et cet «héritage prototypique» est une caractéristique clé de JavaScript.

Toute valeur dans JavaScript qui n'est pas une chaîne, un nombre, **true**, **false**, **null** ou **undefined** est un objet. Et même si les chaînes, les nombres et les booléens ne sont pas des objets, ils se comportent comme des objets immuables. En effet, chaque fois que vous essayez de faire référence à une propriété d'une chaîne s, JavaScript convertit la valeur de chaîne en un objet. Cet objet hérite des méthodes de **String** et est utilisé pour résoudre la référence de propriété. Une fois la propriété est résolue, l'objet nouvellement créé est supprimé. De la même manière, un objet temporaire est également créé à l'aide du constructeur **Number () ou Boolean ()** pour les nombres et les booléens aussi.

2.1.1. Création d'un objet

Les objets peuvent être créés de façon littérale, avec le mot-clé **new**, et (dans ECMAScript 5) avec la fonction **Object.create ()**. Les sous-sections ci-dessous décrivent chaque technique.

2.1.1.1. Créer un objet litéral

La manière la plus simple de créer un objet consiste à le faire de façon littérale dans votre code JavaScript. Un objet littéral est une liste séparée par des virgules d'associations nom-valeur eux même séparées par des deux-points, entourées d'accolades. Un nom de propriété est un identifiant JavaScript ou une chaîne littérale. Une valeur de propriété est une expression JavaScript; la valeur de l'expression (il peut s'agir d'une valeur primitive ou d'une valeur d'objet) devient la valeur de la propriété. Voici quelques exemples:

```
var vide = {}; // Un objet sans propriétés
var point = {x: 0, y: 0}; // Deux propriétés
```

```
var point2 = {x: point.x, y: point.y + 1}; // Valeurs plus complexes
var livre = {
        "sous-titre": "Le Guide Définitif",
        auteur: {
                firstname: "David",
                surname: "Flanagan"
        }
};
```

Les noms de propriétés incluant des espaces et traits d'union sont entourés de quotes ou double quotes.

2.1.1.2. Créer un objet avec new

L'opérateur **new** crée et initialise un nouvel objet. Le mot-clé **new** doit être suivi d'une invocation de fonction. Une fonction utilisée de cette manière est appelée un constructeur et sert à initialiser un objet nouvellement créé.
Le cœur de JavaScript intègre des constructeurs pour les types natifs.
Par exemple:

```
var o = new Objet (); // crée un objet vide: pareil que {}
var a = new Array (); //crée un tableau vide: pareil que []
var d = new Date (); //crée un objet Date correspondant à l'  heure actuelle.
var r = new RegExp ("js");
```

En plus de ces constructeurs intégrés, il est courant de définir vos propres fonctions de constructeur pour initialiser les objets nouvellement créés. Exemple :

```
function person(first, last, age, eye) {
   this.firstName = first;
   this.lastName = last;
   this.age = age;
   this.eyeColor = eye;
}
var myFather = new person("John", "Doe", 50, "blue");
var myMother = new person("Sally", "Rally", 48, "green");
```

2.1.1.3. Les prototypes

Avant de pouvoir aborder la troisième technique de création d'objet, il faut s'arrêter un instant pour expliquer les prototypes. Chaque objet JavaScript a un second objet JavaScript associé à celui-ci. Ce second objet est connu comme un prototype, et le premier objet hérite des propriétés du prototype.

Tous les objets créés par de façon littérale ont le même objet prototype, et nous pouvons nous référer à cet objet prototype dans le code JavaScript faisant **Object.prototype**. Les objets créés à l'aide de **new** et d'une invocation de constructeur utilisent la valeur de la propriété prototype de la fonction constructeur comme prototype. Ainsi, l'objet créé par **new Object ()** hérite de **Object.prototype** tout comme l'objet créé par {}. De même, l'objet créé **par new Array ()** utilise **Array.prototype** comme prototype, et l'objet créé par **new Date ()** utilise **Date.prototype** comme prototype.

Object.prototype est l'un des rares objets qui n'a pas de prototype: il n'hérite d'aucune propriété. D'autres objets prototypes sont des objets normaux qui ont un prototype. Tous les constructeurs intégrés (et la plupart des constructeurs définis par l'utilisateur) ont un prototype qui hérite de Object.prototype.

Par exemple, **Date.prototype** hérite des propriétés de **Object.prototype**, c'est ainsi qu'un objet Date créé avec **new Date ()** hérite des propriétés de **Date.prototype** et **Object.prototype**.

2.1.1.4. Object.create()

ECMAScript 5 définit une méthode, Object.create (), qui crée un nouvel objet, en utilisant son premier argument comme prototype de cet objet. Object.create () prend également un deuxième argument facultatif qui décrit les propriétés du nouvel objet. **Object.create ()** est une fonction statique, pas une méthode invoquée sur des objets individuels. Pour l'utiliser, il suffit de passer l'objet prototype souhaité:

```
var o1 = Object.create({x:1, y:2}); // o1 hérite des propriétés x et y.
```

2.1.2. Affectation et utilisation des propriétés

Pour obtenir la valeur d'une propriété, on utilise les opérateurs point (.) ou crochets ([]). L'élément à gauche des opérateurs doit être une expression dont la valeur est un objet. Si vous utilisez l'opérateur point, l'élément à droite doit être un identifiant simple qui nomme la propriété. Si vous utilisez des crochets, la valeur entre parenthèses doit être une expression qui évaluée contient le nom de propriété désirée :

```
var author = book.author; //récupérer la propriété « author » de l'objet book
var name = author["surname"]; //récupérer la propriété « surname» de l'objet author.
```

Pour créer ou définir une propriété, utilisez un point ou des crochets comme vous le feriez pour interroger la propriété, mais placez-les du côté gauche d'une expression d'affectation:

```
book.edition = 6; // Crée une propriété "édition" pour book.
book["surname"] = "ECMAScript"; // Définit la propriété "surname".
```

UVS
UNIVERSITE VIRTUELLE DU SENEGAL

2.1.3. Les Getters et Setters des propriétés

Nous avons dit qu'une propriété d'objet est un nom, une valeur et un ensemble d'attributs. Dans ECMAScript 5, la valeur peut être remplacée par une ou deux méthodes, appelées getter et setter. Les propriétés définies par les getters et les setters sont parfois appelées propriétés d'accesseur pour les distinguer des propriétés de données ayant une valeur simple.

Les accesseurs sont définies comme une ou deux fonctions dont le nom est le même que le nom de la propriété avec le mot clé **function** remplacé par **get** et / ou **set**.

Exemple :

```
var person = {
   firstName: 'Jimmy',
   lastName: 'Smith',
   get fullName() {
      return this.firstName + ' ' + this.lastName;
   },
   set fullName (name) {
      var words = name.toString().split(' ');
      this.firstName = words[0] || '';
      this.lastName = words[1] || '';
   }
}

person.fullName = 'Jack Franklin';
console.log(person.firstName); // Jack
console.log(person.lastName) // Franklin
```

2.1.4. Sérialisation d'objets

La sérialisation d'objets est le processus de conversion d'un objet en une chaîne de charactère à partir de laquelle il peut être restauré ultérieurement. ECMAScript 5 fournit des fonctions natives JSON.stringify () et JSON.parse () pour sérialiser et restaurer les objets JavaScript. Ces fonctions utilisent le format d'échange de données JSON. JSON signifie "JavaScript Object Notation" et sa syntaxe est très similaire à celle de l'objet JavaScript:

```
o = {x:1, y:{z:[false,null,""]}}; //définition d'un objet de test
s = JSON.stringify(o); // s est la chaine : '{"x":1,"y":{"z":[false,null,""]}}'
p = JSON.parse(s); // p est une copie de o
```

2.1.5. Les méthodes de Object

Comme indiqué précédemment, tous les objets JavaScript (à l'exception de ceux explicitement créés sans prototype) héritent des propriétés de Object.prototype. Ces propriétés héritées sont principalement des méthodes, et parce qu'elles sont universellement disponibles, elles intéressent particulièrement les programmeurs JavaScript. Cette section explique une poignée de méthodes d'objets universelles définies sur Object.prototype, mais qui sont censées être remplacées par d'autres classes plus spécialisées.

2.1.5.1. La méthode toString()

La méthode **toString ()** ne prend aucun argument; il renvoie une chaîne qui représente en quelque sorte la valeur de l'objet sur lequel il est appelé. JavaScript appelle cette méthode d'un objet chaque fois qu'il doit convertir l'objet en chaîne. Cela se produit, par exemple, lorsque vous utilisez l'opérateur + pour concaténer une chaîne avec un objet ou lorsque vous passez un objet à une méthode qui attend une chaîne.
La méthode par défaut toString () n'est pas très informative. Par exemple, la ligne de code suivante sera évaluée simplement en à la chaîne **"[objet objet]"** :

```
var s = { x:1, y:1 }.toString();
```

Comme cette méthode par défaut n'affiche pas beaucoup d'informations utiles, de nombreuses classes définissent leur propre version de toString ().

2.1.5.2. La méthode toJSON()

Object.prototype ne définit pas réellement une méthode toJSON (), mais la méthode JSON.stringify () recherche une méthode toJSON () sur n'importe quel objet à sérialiser. Si cette méthode existe sur l'objet à sérialiser, elle est invoquée et la valeur de retour est sérialisée au lieu de l'objet d'origine.

2.1.5.3. La méthode valueOf()

La méthode valueOf () ressemble beaucoup à la méthode toString (), mais elle est appelée lorsque JavaScript doit convertir un objet en un type primitif autre qu'une chaîne, généralement un nombre. JavaScript appelle automatiquement cette méthode si un objet est utilisé dans un contexte où une valeur primitive est requise. La méthode valueOf () par défaut n'a rien d'intéressant, mais certaines classes intégrées définissent leur propre méthode valueOf ().

2.2. Tableaux

Un tableau est une collection ordonnée de valeurs. Chaque valeur est appelée un élément et chaque élément a une position numérique dans le tableau, connu sous le nom d'index. Les

tableaux JavaScript ne sont pas typés: un élément de tableau peut être de n'importe quel type, et différents éléments du même tableau peuvent être de types différents. Les éléments de tableau peuvent même être des objets ou d'autres tableaux, ce qui vous permet de créer des structures de données complexes, telles que des tableaux d'objets et des tableaux de tableaux.

Les tableaux JavaScript sont une forme spécialisée d'objet JavaScript et les index de tableau ne sont en réalité rien de plus que des noms de propriété qui sont des entiers. Les tableaux héritent des propriétés de **Array.prototype**, qui définit un ensemble riche de méthodes de manipulation de tableaux. La plupart de ces méthodes sont génériques, ce qui signifie qu'elles fonctionnent correctement non seulement pour les tableaux réels, mais pour tout "objet de type tableau". Il est à noter que dans ECMAScript 5, les chaînes se comportent comme des tableaux de caractères.

2.2.1. Création d'un tableau

Le moyen le plus simple de créer un tableau est d'utiliser la syntaxe littérale de tableau, qui est simplement une liste d'éléments de tableau séparés par des virgules entre crochets. Par exemple:

```
var empty = []; // Un tableau sans éléments
var primes = [2, 3, 5, 7, 11]; // Un tableau avec 5 éléments numériques
var misc = [1.1, true, "a",]; // 3 éléments de types différents + une virgule de fin
```

Les valeurs d'un tableau littéral n'ont pas besoin d'être des constantes; ils peuvent être des expressions arbitraires:

```
var base = 1024;
var table = [base, base+1, base+2, base+3];
```

Les tableaux littéraux peuvent contenir des objets littéraux ou d'autres tableaux:

```
var b = [[1, {x: 1, y: 2}], [2, {x: 3, y: 4}]];
```

Si vous omettez une valeur d'un de tableau littéral, l'élément omis reçoit la valeur **undefined**:

```
var count = [1,, 3]; // Un tableau avec 3 éléments, le milieu indéfini.
var undefs = [,,]; // Un tableau avec 2 éléments, tous deux indéfinis.
```

La syntaxe littérale de tableau autorise une virgule de fin optionnelle, donc [,,] n'a que deux éléments, pas trois.

Une autre façon de créer un tableau est avec le constructeur **Array ()**. Vous pouvez appeler ce constructeur de trois manières distinctes:

L'appeler sans arguments:

```
var a = new Array ();
```

Cette méthode crée un tableau vide sans éléments et est équivalent au tableau literal [].

L'appeler avec un seul argument numérique, qui spécifie une longueur: var a = new Array(10);

Cette technique crée un tableau avec la longueur spécifiée. Cette forme du constructeur Array () peut être utilisée pour préallouer un tableau lorsque vous savez à l'avance combien d'éléments seront nécessaires. Notez qu'aucune valeur n'est stockée dans le tableau et que les propriétés d'index de tableau "0", "1", etc ne sont même pas définies pour le tableau

Spécifiez explicitement deux ou plusieurs éléments de tableau ou un seul élément non numérique pour le tableau:

var a = new Array(5, 4, 3, 2, 1, "testing, testing");

Dans cette forme, les arguments du constructeur deviennent les éléments du nouveau tableau. L'utilisation de la syntaxe littérale de tableau est presque toujours plus simple que cette utilisation du constructeur Array ().

2.2.2. Lecture et écriture d'éléments dans un tableau

Vous accédez à un élément d'un tableau en utilisant l'opérateur **[]**. Une référence au tableau doit apparaître à gauche des crochets. Une expression arbitraire ayant une valeur entière non négative doit se trouver à l'intérieur des crochets. Vous pouvez utiliser cette syntaxe pour lire et écrire la valeur d'un élément d'un tableau. Ainsi, les éléments suivants sont tous des instructions JavaScript légales :

```
var a = ["world"]; // Initialiser avec un tableau à un seul élément
var value = a[0]; // Lire l'élément 0
a[1] = 3.14; // Ecriture d'un élément à l'index 1

i = 2;
a[i] = 3; // Écriture de l'élément 2
a[i + 1] = "hello"; // Ecriture d'un élément 3
a[a[i]] = a[0]; // Lire les éléments 0 et 2, écrire l'élément 3
```

2.2.3. Longueur d'un tableau

Chaque tableau a une propriété **length**, et c'est cette propriété qui rend les tableaux différents des objets JavaScript habituels. Sa valeur est la valeur de l'indice le plus élevé du tableau plus un (le nombre d'élément dans le tableau) :

```
[].length // => 0: le tableau n'a pas d'éléments
['a','b','c'].length // => 3: l'indice le plus élevé est 2, la longueur est donc 3
```

Les tableaux ont deux comportements spéciaux.

Pour la première, si vous affectez une valeur à un élément de tableau dont l'indice **i** est supérieur ou égal à la longueur courante du tableau, la valeur de la propriété **length** est définie sur **i + 1**.

Le second comportement spécial que les tableaux implémentent afin de maintenir la longueur invariable est que si vous définissez la propriété length sur un entier non négatif n plus petit que sa valeur actuelle, tous les éléments du tableau dont l'indice est supérieur ou égal à n sont supprimés du tableau:

```
a = [1,2,3,4,5]; // Initialisation d'un tableau à 5 éléments.
a.length = 3; // a est maintenant [1,2,3].
a.length = 0; // Supprimer tous les éléments. a est [].
a.length = 5; // La longueur est 5, mais pas d'éléments, comme new Array (5)
```

Dans ECMAScript 5, vous pouvez rendre la propriété length d'un tableau en lecture seule avec Object.defineProperty () :

```
a = [1,2,3]; // Initialisation d'un tableau à 3 éléments.
Object.defineProperty (a, "length", {writable: false}); // Crée la //propriété
length en lecture seulement.
a.length = 0; // a est inchangé.
```

2.2.4. Ajout et suppression d'éléments de tableau

Nous avons déjà vu le moyen le plus simple d'ajouter des éléments à un tableau: il suffit d'attribuer des valeurs aux nouveaux index:

```
a = []    // Un tableau vide.
a[0] = "zero"; // ajouter un élément dans le tableau.
a[1] = "one";
```

Vous pouvez également utiliser la méthode **push ()** pour ajouter une ou plusieurs valeurs à la fin d'un tableau:

```
a = []; // Commencer avec un tableau vide
a.push ("zero") // Ajoute une valeur à la fin comme. a = ["zero"]
a.push ("un", "deux") // Ajoute deux autres valeurs. a = ["zéro", "un", "deux"]
```

Pousser une valeur sur un tableau **a** revient à attribuer une valeur à **[a.length]**. Vous pouvez utiliser la méthode **unshift ()** pour insérer une valeur au début d'un tableau, en déplaçant les éléments du tableau existant vers des index plus élevés.

Vous pouvez supprimer des éléments de tableau avec l'opérateur de suppression, tout comme vous pouvez supprimer les propriétés de l'objet:

```
a = [1,2,3];
```

```
delete a[1]; // maintenant , a n'a aucun élément à l'index 1
a.length ; // => 3: delete n'affecte pas la longueur du tableau
```

La suppression d'un élément de tableau est similaire à l'attribution **d'undefined** à cet élément. Notez que l'utilisation de delete sur un élément de tableau ne modifie pas la propriété length et ne décale pas les éléments avec des index plus élevés pour remplir l'espace laissé par la propriété deleted.

2.2.5. Itérer sur les tableaux

La façon la plus courante de faire une boucle dans les éléments d'un tableau est avec une boucle for.

```
var keys = Object.keys(o); // Obtenir un tableau de noms des propriétés
//de l'objet o

var values = [] ; // Stocker les valeurs associées aux propriétés dans ce
tableau for(var i = 0; i < keys.length; i++) { // pour chaque index du tableau
   var key = keys[i]; // Récupère la clé à cet index
   values[i] = o[key]; // Stocke la valeur dans le tableau de valeurs
}
```

Dans les boucles imbriquées ou dans d'autres contextes où les performances sont critiques, vous pouvez parfois voir cette boucle d'itération optimisée de sorte que la longueur du tableau ne soit recherchée qu'une seule fois plutôt qu'à chaque itération:

```
for(var i = 0, len = keys.length; i < len; i++) {
        // Le corps de la boucle reste le même
}
```

Vous pouvez également utiliser une boucle **for/in** avec des tableaux. Cette boucle assigne les noms de propriété énumérables (y compris les index de tableau) à la variable de boucle, l'un après l'autre. Les index qui n'existent pas ne seront pas itérés pour les tableaux clairsemés (voir la section précédente avec l'opérateur delete) :

```
for(var index in sparseArray) {
        var value = sparseArray[index];
        // On peut donc utiliser index and value
}
```

ECMAScript 5 définit un certain nombre de nouvelles méthodes d'itération des éléments de tableau en les passant chacun, dans l'ordre de l'index, à une fonction que vous définissez. La méthode **forEach ()** est la plus générale de ces méthodes:

```
var data = [1,2,3,4,5]; // Ceci est le tableau sur lequel nous voulons itérer
var sumOfSquares = 0; // Nous voulons calculer la somme des carrés des //données
```

```
data.forEach(function(x) {
// Passer chaque élément de données à cette fonction
sumOfSquares sumOfSquares += x*x; // additionne les carrés
});
sumOfSquares // contient 55 : 1+4+9+16+25
```

2.2.6. Tableaux multidimensionnels

JavaScript ne prend pas en charge les tableaux multidimensionnels réels, mais vous pouvez les approximer avec des tableaux de tableaux. Pour accéder à une valeur dans un tableau de tableaux, utilisez simplement l'opérateur [] deux fois. Par exemple, supposons que la matrice de variables est un tableau de tableaux de nombres. Chaque élément de la matrice [x] est un tableau de nombres. Pour accéder à un nombre particulier dans ce tableau, vous devez écrire matrix [x] [y]. Voici un exemple concret qui utilise un tableau à deux dimensions comme table de multiplication:

```
// Créer un tableau multidimensionnel
var table = new Array(10); // 10 lignes de la table
for(var i = 0; i < table.length; i++){
        table[i] = new Array(10); // Chaque ligne a 10 colonnes
}
```

2.3. Fonctions

Une fonction est un bloc de code JavaScript qui est défini une fois mais peut être exécuté, ou invoqué, autant de fois que nécessaire. Les fonctions JavaScript sont paramétrées: une définition de fonction peut inclure une liste d'identifiants, appelés paramètres, qui fonctionnent comme des variables locales pour le corps de la fonction. Les invocations de fonction fournissent des valeurs ou des arguments pour les paramètres de la fonction. Les fonctions utilisent souvent leurs valeurs d'argument pour calculer une valeur de retour qui devient la valeur de l'expression d'invocation de fonction. En plus des arguments, chaque invocation a une autre valeur, le contexte d'invocation, qui est la valeur du mot-clé **this**.

Si une fonction est affectée à la propriété d'un objet, elle est appelée méthode de cet objet. Lorsqu'une fonction est invoquée sur ou via un objet, cet objet est le contexte d'appel ou la valeur pour la fonction. Les fonctions conçues pour initialiser un objet nouvellement créé sont appelées constructeurs.

En JavaScript, les fonctions sont des objets, et ils peuvent être manipulés par des programmes. JavaScript peut assigner des fonctions à des variables et les passer à d'autres fonctions, par exemple. Les fonctions étant des objets, vous pouvez définir des propriétés sur ces objets et même y appeler des méthodes.

Les définitions de fonction JavaScript peuvent être imbriquées dans d'autres fonctions, et elles ont accès à toutes les variables qui sont dans la portée où elles sont définies.

UNIVERSITE VIRTUELLE DU SENEGAL

2.3.1. Définir des fonctions

Les fonctions sont définies avec le mot-clé **function**, qui peut être utilisé dans une expression de définition ou dans une déclaration de fonction. Dans les deux cas, les définitions de fonction commencent par le mot clé **function** suivi des composants:

Un identifiant qui nomme la fonction. Le nom est une partie obligatoire des déclarations de fonction: il est utilisé comme nom d'une variable et l'objet de fonction nouvellement défini est affecté à la variable. Pour les expressions de définition de fonction, le nom est facultatif: s'il est présent, le nom fait référence à l'objet fonction uniquement dans le corps de la fonction elle-même.
Une paire de parenthèses autour d'une liste séparée par des virgules de zéro ou plusieurs identifiants. Ces identifiants sont les noms des paramètres de la fonction et se comportent comme des variables locales dans le corps de la fonction.
Une paire d'accolades avec zéro ou plusieurs instructions JavaScript à l'intérieur. Ces instructions sont le corps de la fonction: elles sont exécutées chaque fois que la fonction est invoquée.

Les exemples suivants montrent certaines définitions de fonctions. Notez qu'une fonction définie en tant qu'expression n'est utile que si elle fait partie d'une expression plus grande, telle qu'une affectation ou un appel, qui fait quelque chose avec la fonction nouvellement définie.

```
// Affiche le nom et la valeur de chaque propriété de o. Retour undefined.
    function printprops(o) {
        for(var p in o)
            console.log(p + ": " + o[p] + "\n");
    }


// Calcule la distance entre les points cartésiens (x1, y1) et (x2,
    y2). function distance(x1, y1, x2, y2) {
        var dx = x2 - x1;
        var dy = y2 - y1;
        return Math.sqrt(dx*dx + dy*dy);
    }


// Une fonction récursive (qui s'appelle elle-même) qui calcule les factoriels
//  Rappelez-vous que x! est le produit de x et de tous les entiers positifs inférieurs //à
celui-ci.

    function factorial(x) {
        if (x <= 1) return 1;
        return x * factorial(x-1);
```

UVS
UNIVERSITE VIRTUELLE DU SENEGAL

```
}

// Cette expression de fonction définit une fonction qui équivaut au carré de son //
argument.
// Notez que nous l'assignons à une variable var
    square = function (x) {return x * x; }


//  Les expressions de fonction peuvent inclure des noms, ce qui est utile pour la
//récursivité.
    var f = function fact(x) { if (x <= 1) return 1; else return x*fact(x-1); };

// Les expressions de fonction peuvent également être utilisées comme //arguments pour
d'autres fonctions:
    data.sort(function(a,b) { return a-b; });


// Les expressions de fonction sont parfois définies et appelées immédiatement:
    var tensquared = (function(x) {return x*x;}(10));
```

2.3.2. Invoquer des fonctions

Le code JavaScript qui constitue le corps d'une fonction n'est pas exécuté lorsque la fonction est définie mais lorsqu'elle est invoquée. Les fonctions JavaScript peuvent être invoquées de quatre manières:

en tant que fonctions,
en tant que méthodes,
en tant que constructeurs,
indirectement via leurs méthodes call () et apply ().

2.3.2.1. Invocation en tant que fonction

Les fonctions sont appelées en tant que fonctions ou en tant que méthodes avec une expression d'invocation. Une expression d'invocation consiste en une expression de fonction qui évalue un objet de fonction suivi d'une parenthèse ouvrante, d'une liste séparée par des virgules de zéro ou plusieurs expressions d'argument et d'une parenthèse fermante. Si l'expression de la fonction est une expression d'accès à la propriété, si la fonction est la propriété d'un objet ou d'un élément d'un tableau, il s'agit alors d'une expression d'invocation de méthode. Le code suivant inclut un certain nombre d'expressions d'invocation de fonction régulière:

```
printprops({x:1});
var total = distance(0,0,2,1) + distance(2,1,3,5);
var probability = factorial(5)/factorial(13);
```

UVS
UNIVERSITE VIRTUELLE DU SENEGAL

2.3.2.2. Invocation en tant que méthode

Une méthode n'est rien de plus qu'une fonction JavaScript stockée dans une propriété d'un objet. Si vous avez une fonction f et un objet o, vous pouvez définir une méthode nommée **m** de **o** avec la ligne suivante:

```
o.m = f;
```

Après avoir défini la méthode m () de l'objet o, on peut l'invoquer comme ceci:

```
o.m();
```

Ou, si m () attend deux arguments, vous pouvez l'invoquer comme ceci:

```
o.m(x, y);
```

Le code ci-dessus est une expression d'invocation: il inclut une expression de fonction o.m et deux expressions d'argument, x et y. L'expression de la fonction est elle-même une expression d'accès à la propriété, ce qui signifie que la fonction est invoquée comme une méthode plutôt que comme une fonction régulière.

Les arguments et la valeur de retour d'une invocation de méthode sont traités exactement comme décrit ci-dessus pour l'invocation de fonction régulière. Cependant, les invocations de méthodes diffèrent des invocations de fonctions d'une manière importante: le contexte d'invocation. Les expressions d'accès à la propriété se composent de deux parties: un objet (dans ce cas o) et un nom de propriété (m). Dans une expression d'invocation de méthode comme celle-ci, l'objet o devient le contexte d'invocation, et le corps de la fonction peut se référer à cet objet en utilisant le mot-clé **this**. Voici un exemple concret:

```
var calculator = { // Un object literal
        operand1: 1,
        operand2: 1,
        add: function() {
                // Notez l'utilisation du mot-clé this pour faire référence à cet
//objet.
                this.result = this.operand1 + this.operand2;
        }
};
calculator.add(); // Une invocation de méthode pour calculer 1 + 1.
calculator.result // résultat => 2
```

La plupart des invocations de méthode utilisent la notation par points pour l'accès aux propriétés, mais les expressions d'accès aux propriétés qui utilisent des crochets provoquent également l'invocation de la méthode. Voici les deux invocations de méthode, par exemple:

o["m"](x,y); // Un autre moyen d'écrire o.m(x,y).

a[0](z) // une autre invocation de méthode (en considérant que a[0] est //une fonction).

2.3.2.3. Invocation en tant que constructeur

Si une invocation de fonction ou de méthode est précédée du mot-clé **new**, il s'agit d'un appel de constructeur. Les invocations de constructeurs diffèrent des invocations régulières de fonctions et de méthodes dans la gestion des arguments, le contexte d'invocation et la valeur de retour.

Si une invocation de constructeur inclut une liste d'arguments entre parenthèses, ces expressions d'argument sont évaluées et transmises à la fonction de la même manière qu'elles le seraient pour les invocations de fonction et de méthode. Mais si un constructeur n'a pas de paramètres, la syntaxe d'invocation du constructeur JavaScript permet d'omettre complètement la liste des arguments et les parenthèses. Vous pouvez toujours omettre une paire de parenthèses vides dans un appel de constructeur et les deux lignes suivantes, par exemple, sont équivalentes:

```
var o = new Object();
var o = new Object;
```

Les fonctions de constructeur n'utilisent normalement pas le mot-clé **return**. Ils initialisent typiquement le nouvel objet et le retournent implicitement quand ils atteignent la fin de leur corps. Dans ce cas, le nouvel objet est la valeur de l'expression d'invocation du constructeur. Si, toutefois, un constructeur utilise explicitement l'instruction return pour retourner un objet, alors cet objet devient la valeur de l'expression d'invocation. Si le constructeur utilise return sans valeur, ou s'il renvoie une valeur primitive, cette valeur de retour est ignorée et le nouvel objet est utilisé comme valeur de l'appel.