Chapitre 3 : JavaScript et la programmation orientée objet

Objectif : Comprendre les concepts de base derrière la programmation orientée objet et comment ils s'appliquent à JavaScript (« tout est objet ») et comment créer des constructeurs et instancier des objets.

Prérequis : Connaissances de base en informatique et compréhension des notions HTML et CSS, notions de JavaScript (voir Premiers pas et Blocs de construction)

1. Introduction

JavaScript avait la réputation d'être un langage peu élaboré et peu structuré, dont l'utilisation se limitait aux contrôles des saisies. De ce fait, une utilisation standard de Java- Script dans des pages HTML entraînait rapidement des interférences entre différentes fonctions ou variables, lesquelles sont globales à une page, et, par voie de conséquence, des difficultés à maintenir le code, d'autant que la lisibilité de celui-ci tend à diminuer fortement en fonction de sa taille.

Avec l'arrivée des applications Internet riches, le code JavaScript embarqué se doit d'être beaucoup plus robuste et structuré, puisque les fonctionnalités et les traitements deviennent beaucoup plus complexes. Il doit également prendre en compte toutes les possibilités du langage afin d'être modulaire et évolutif.

Le langage JavaScript offre la possibilité de mettre en œuvre la plupart des concepts objet, tels que l'héritage et le polymorphisme. Les développeurs peuvent tirer parti de ces mécanismes pour augmenter la qualité, la lisibilité et la modularité de leurs applications JavaScript.

La façon de mettre en œuvre les concepts objet est spécifique à ce langage et est essentiellement

fondée sur les fonctions et closures de JavaScript.

2. Rappel des principes de la programmation objet

Avant d'aborder la programmation orientée objet avec JavaScript, nous allons revenir sur les concepts de ce paradigme ainsi que sur ses avantages. Pour illustrer notre propos, nous utiliserons le langage de modélisation UML. L'objectif de cette section n'est pas de décrire les spécificités de tel ou tel langage objet mais d'illustrer leurs principaux concepts.

La programmation orientée objet est un paradigme visant à améliorer la modularité des applications en utilisant des abstractions plus évoluées que les fonctions ou les procédures. Son objectif est de rassembler en une seule entité logique (la classe) un ensemble de données (les attributs) et un ensemble de traitements (les méthodes). Les méthodes peuvent s'appuyer sur les attributs pour réaliser leurs traitements. L'objectif est de modulariser les traitements afin de faciliter leur réutilisation.

Ce type de langage permet de mettre en œuvre le mécanisme d'héritage afin d'étendre les mécanismes d'une classe. Un des avantages de ce mécanisme réside dans la possibilité d'utiliser des classes existantes sans connaître leurs implémentations. Le **polymorphisme** va de pair avec **l'héritage** puisqu'il spécifie qu'un objet peut être considéré du point de vue du type de sa classe d'instanciation mais également du type d'une de ses classes mères.

Les sections qui suivent entrent plus en détail dans les concepts de la programmation orientée objet.

2.1. Classes et objets

Le concept central de la programmation objet est la classe. Composée d'attributs et de méthodes, c'est elle qui permet de définir la structure des entités manipulées. Attributs et méthodes permettent de définir respectivement les états et les comportements de la classe.

Dans la plupart des langages objet, le mot-clé dédié class est introduit afin de définir les classes. Dans toute cette section, nous illustrons notre propos à l'aide de la classe **Vehicule,** dont la figure suivante illustre la structure par l'intermédiaire du diagramme de classes du langage UML.

Vehicule

-marque : String **-modele : String** **-couleur : String** **-vitesse : int**
+Vehicule() **+repeindre(entrée couleur : String)** **+accelerer()** **+freiner()**

Une classe possède toujours une ou plusieurs méthodes particulières, appelées ***constructeur,*** permettant de construire l'objet. Certains langages en fournissent un par défaut s'il n'est pas explicitement défini dans la classe.

Une classe possède également des attributs, permettant de stocker des états, et des méthodes, permettant d'exécuter des traitements en utilisant éventuellement ces attributs.

Les langages objet introduisent le mot-clé **this** afin de permettre à un objet de s'**autoréférencer**. Il permet notamment de différencier les variables locales par rapport aux attributs de classe ou encore de permettre à un objet de se passer en paramètre à un autre objet.

La notion de package est également introduite afin d'organiser les classes. Un package correspond à un ensemble de classes organisées selon une structure arborescente.

2.2. Instances et objets

Une classe n'est pas utilisable directement dans une application, car elle correspond à un concept abstrait et structurel. Les applications travaillent sur des objets ou instances correspondant à des occurrences de classes. Ces instances sont créées à la demande à partir du modèle de structure qu'est la classe par l'intermédiaire du mot-clé new.

Prenons l'exemple de la classe **Vehicule**. Cette dernière définit ses **attributs (marque, modèle, couleur)** et **méthodes (repeindre, accelerer, freiner)**. Une **instance** de véhicule correspond à un véhicule d'une marque et d'un modèle particuliers. Elle possède une couleur, et il est possible d'interagir avec elle afin d'accélérer ou de freiner, par exemple.

Le code suivant illustre l'utilisation d'une instance de la classe **Vehicule** :

```
Vehicule monVehicule = new Vehicule("maMarque","monModele","bleue");
monVehicule.accelerer ();
(...)
monVehicule.freiner();
```

Nous constatons qu'une instance ou objet correspond à une entité existante dans l'application. Les attributs de la classe prennent des valeurs spécifiques, qui déterminent leurs comportements. La classe elle-même permet uniquement de définir la structure de ces instances.

2.3. Encapsulation et visibilité

Comme nous l'avons vu, une classe correspond à une entité possédant des attributs et des méthodes. Il n'est pas forcément utile de rendre ces derniers accessibles depuis l'extérieur de la classe. L'encapsulation revient à interdire l'accès à certains éléments d'une classe afin de protéger ses états et fonctionnements internes.

La plupart du temps, les attributs de classe ne doivent pas être exposés directement à l'extérieur de la classe. C'est la raison pour laquelle la mise en œuvre d'accesseurs et de mutateurs constitue une bonne pratique de conception.

Afin de mettre en œuvre l'encapsulation, la plupart des langages objet offrent un support permettant de contrôler la visibilité des attributs et méthodes par des éléments de langage.

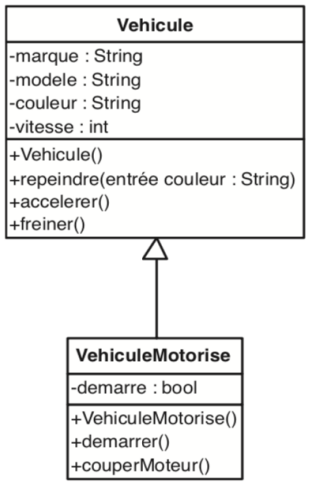
Il existe plusieurs niveaux de visibilité des attributs et méthodes, dont la signification est quasiment commune à la plupart des langages objet.

Niveau de visibilité	Description
Par défaut	Seules les entités du même package et les sous-classes ont accès aux éléments de la classe.
Privé	Les entités externes n'ont pas accès aux éléments de la classe.
Protégé	Seules les sous-classes ont accès aux éléments de la classe.
Public	Les entités externes ont accès aux éléments de la classe.

2.4. L'héritage

Par souci de modularité, les langages orientés objet mettent en œuvre des mécanismes d'héritage. Ces derniers permettent de définir des sous-classes afin d'enrichir et de bénéficier d'attributs et de méthodes de classes existantes. Le mot-clé **extends** est générale- ment mis à disposition du développeur afin de créer les relations d'héritage entre classes.

La figure 3.2 illustre l'utilisation de la méthode UML et de son diagramme de classes afin de décrire un lien d'héritage entre la classe **Vehicule** précédemment décrite et la classe **VehiculeMotorise**.



La classe **VehiculeMotorise** spécialise la classe **Vehicule** et peut utiliser le constructeur de la classe **Vehicule** par l'intermédiaire du mot-clé **super**. Il est également possible de spécifier de nouveaux attributs ainsi que de nouvelles méthodes. Les méthodes peuvent appeler des méthodes de la classe mère, si nécessaire.

D'une manière générale, les langages orientés objet offrent un mécanisme permettant de référencer la classe mère. Dans le cas de Java, le mot-clé super est mis à la disposition des sous-classes afin de référencer la classe mère. Il permet ainsi d'utiliser les constructeurs, méthodes et attributs accessibles de cette dernière.

2.5. Encapsulation et visibilité

La plupart du temps, les langages orientés objet donnent la possibilité de contrôler l'accès aux éléments auxquels ont accès les classes.

En reprenant les niveaux de visibilité décrits précédemment, les différentes possibilités sont les suivantes :

- Une méthode ou un attribut public peut être utilisé par n'importe quelle classe
- Une méthode ou un attribut protégé peut être utilisé par les sous-classes mais n'est pas visible par les classes sans lien d'héritage.
- Une méthode ou un attribut privé ne peut être utilisé que par la classe elle-même. Les sous-classes n'y ont pas accès.

Reprenons l'exemple des classes **Vehicule** et **VehiculeMotorise** de la section précédente. Si nous modifions la visibilité de la méthode **accelerer()** de la classe **Vehicule** avec la valeur protégée, cette dernière peut être utilisée dans la classe **VehiculeMotorise** mais n'est pas accessible en dehors des sous-classes de la classe **Vehicule**. Si sa visibilité devient privée, elle ne peut être utilisée que dans la classe **Vehicule**.

2. 6. Classes et méthodes abstraites

Une classe abstraite est une classe qui ne peut être instanciée dans une application. Il s'agit d'une abstraction de haut niveau, destinée à être spécialisée afin de répondre à différents contextes d'utilisation. Elle ne peut donc être utilisée que par l'intermédiaire de classes qui l'étendent. Cette classe définit la plupart du temps une ou plusieurs méthodes abstraites, dont le code n'est pas défini dans la classe. Ces dernières peuvent cependant être utilisées dans les méthodes concrètes de la classe. Son rôle est de permettre aux sous-classes de paramétrer l'algorithme des méthodes de la classe mère. Les classes filles concrètes ont quant à elles la responsabilité de les implémenter. Les langages orientés objet utilisent le mot-clé abstract afin de définir ces types de classes et de méthodes. Si nous reprenons notre classe **Vehicule**, nous pouvons la rendre abstraite ainsi que ses méthodes **accelerer()** et **freiner().** Cette classe ne peut toutefois être instanciée, si bien que les deux méthodes précédentes n'ont pas d'implémentation. Une sous-classe telle que **VehiculeMotorise** aurait la responsabilité, par exemple, de les implémenter.

2. 7. Classes et méthodes finales

Une classe finale est une classe qui ne peut être étendue dans une application et correspond

donc au dernier niveau d'héritage.

N'importe quelle classe peut définir une ou plusieurs méthodes finales. Dans ce cas, ces dernières ne peuvent être surchargées dans une sous-classe. Notons qu'une classe ne doit pas être nécessairement finale pour définir des méthodes finales.

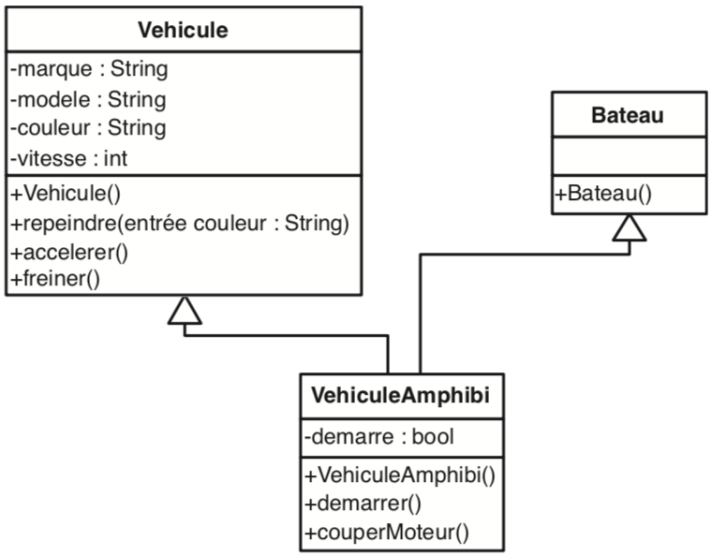
Les langages orientés objet utilisent le mot-clé final afin de définir ces types de classes et de méthodes. Si nous reprenons notre classe Vehicule, nous pouvons rendre ses méthodes **accelerer()** et **freiner()** finales. Ces deux méthodes ne peuvent toutefois être surchargées dans les sous-classes telles que VehiculeMotorise.

Notons qu'il est possible de définir des attributs finaux qui correspondent à des constantes.

2.8. L'héritage multiple

Certains langages orientés objet permettent de faire hériter une classe de plusieurs autres. Ce mécanisme est supporté par C++ mais pas par Java.

La figure suivante illustre un exemple d'héritage multiple fondé sur les classes Vehicule et **VehiculeAmphibi**. Cette dernière hérite désormais de la classe **Amphibi** afin de spécifier que le véhicule motorisé peut se déplacer dans l'eau.



Les mécanismes décrits aux sections précédentes s'appliquent également à l'héritage multiple.

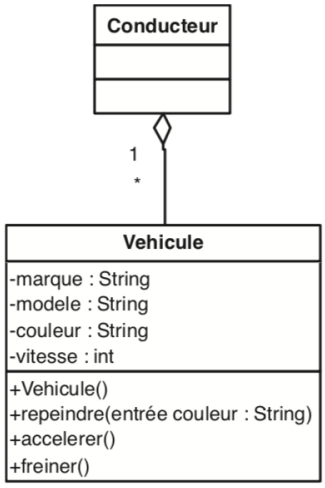
Il faut avoir à l'esprit que la mise en œuvre de l'héritage multiple engendre des problèmes et complexifie les relations entre les classes. Le principal problème peut provenir de l'existence

de méthodes ou d'attributs similaires dans les parents d'une classe. Dans ce cas, afin de garantir une solution cohérente, le langage doit fournir un mécanisme pour réaliser une fusion ou un renommage des entités dupliquées.

2.9. Agrégation et composition

L'agrégation correspond au fait de référencer un objet dans un autre objet. Ainsi, un attribut d'une classe est du type d'une autre classe.

La figure suivante illustre la représentation en UML d'une relation d'agrégation entre les classes **Vehicule** et **Conducteur**.

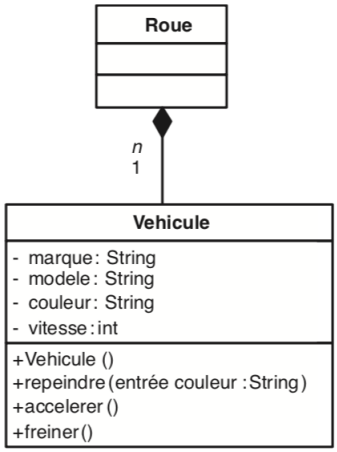


La classe Vehicule est enrichie par l'information relative à son conducteur principal. Cela se traduit par un attribut de classe, **conducteurPrincipal**, et éventuellement des méthodes de manipulation de cet attribut. L'agrégation peut également être réalisée par l'intermédiaire d'un attribut d'une classe correspondant à une collection.

La composition correspond à une agrégation forte. En effet, les cycles de vie des objets référencés suivent celui de l'objet contenant, si bien que lorsque ce dernier est détruit, les autres le sont également, contrairement à ce qui se produit avec l'agrégation.

La figure suivante illustre la schématisation d'une relation de composition entre la classe **Vehicule** et une classe Roue avec le langage UML.

Dans l'exemple ci-dessus, lorsqu'une instance de la classe Vehicule est détruite, les références à la classe **Roue** sont également détruites.



2.10. Le polymorphisme

Les mécanismes de **polymorphisme** permettent à une application de voir une instance sous différentes formes. Cette instance peut ainsi être considérée comme étant non seule- ment du type de sa classe, mais également des types de ses différentes classes mères. Une instance de notre classe **VehiculeMotorise** peut dès lors être vue comme une instance de sa classe mère Vehicule. La considérer comme **Vehicule** revient toutefois à restreindre le nombre de ses méthodes utilisables.

Le **polymorphisme** est intimement lié aux mécanismes de transtypage, qui permettent de convertir le type d'une instance, quand c'est possible, en un autre type.

Le code suivant donne un exemple de transtypage d'instances :

```
VehiculeMotorise voiture = (...);
Vehicule vehicule = (Vehicule)voiture; ❶
VehiculeMotorise laMemeVoiture = (VehiculeMotorise)vehicule; ❷
```

Au repère ❶, le transtypage ascendant **VehiculeMotorise** fonctionne correctement puisque la classe **Vehicule** est une classe mère de la classe **VehiculeMotorise**. Au repère ❷, le **transtypage** descendant est également correct puisque l'instance **transtypée** est de type **VehiculeMotorise**.

3. Classes JavaScript

Comme nous venons de le voir, JavaScript n'offre qu'un support partiel des concepts orientés objet mais permet d'utiliser différents types de classes. Certaines d'entre elles sont fournies directement par le langage, tandis que d'autres le sont par l'environnement d'exécution. Le

langage offre également au développeur la possibilité de définir ses propres classes par l'intermédiaire des fonctions.

Bien qu'il permette l'utilisation d'objets et dispose d'objets natifs, JavaScript n'est pas véritablement un langage orienté objet. Il ne fournit pas d'éléments de langage afin de supporter ce paradigme mais permet simplement d'émuler les différents principes de la programmation orientée objet.

JavaScript ne supporte donc pas véritablement la notion de classe et n'offre pas de mot-clé dédié class afin d'en définir. Pour mettre en œuvre les concepts de la programmation objet, il se fonde uniquement sur les mécanismes des fonctions et des closures ainsi que sur l'attribut prototype.

NB : Le langage JavaScript ne supporte pas la notion de classe afin de définir la structure des objets. Il se fonde à la place sur les principes des fonctions. Dans ce langage, les fonctions correspondent à des objets puisqu'elles y peuvent être référencées. C'est la raison pour laquelle certains auteurs préfèrent ne pas parler de classe JavaScript mais plutôt d'objet JavaScript, aussi bien pour l'entité définissant la structure d'un objet que pour ses instances. Par exemple, certains parlent de l'objet **XMLHttpRequest** plutôt que de la classe **XMLHttpRequest**.

Dans cet ouvrage, afin de simplifier nos explications et d'éviter toute confusion, nous avons fait le choix de désigner la structure des objets JavaScript par le terme de classe et les instances de ces classes par le terme d'objet.

Dans ce contexte, un objet ou une instance correspond à une collection d'attributs et de fonctions, appelées dans ce contexte ***méthodes***. Ils peuvent être ajoutés et supprimés dynamiquement au cours de son utilisation. Nous utilisons par la suite le terme ***classe*** pour désigner la structure de l'objet qui spécifie la liste de ses méthodes et attributs.

JavaScript fournit deux possibilités de définir un objet. La première consiste à utiliser la forme littérale d'un tableau associatif et la seconde à se fonder sur une fonction constructeur ou sur son attribut prototype. Ces deux derniers éléments peuvent éventuellement être utilisés conjointement.

Le tableau suivant recense les différents types de classes du langage.

Type	Description

Classes applicatives	JavaScript permet à l'application de créer ses propres classes et objets dans un souci de structuration et de modularité.
Classes de base JavaScript	fournit les classes relatives aux différents types natifs du langage.
Classes fournies par l'environnement d'exécution	L'environnement d'exécution fournit des classes aux applications JavaScript. Dans le cas d'une exécution dans un navigateur, par exemple, des instances de ces classes relatives aux éléments HTML contenus dans la page sont disponibles.
Classes préinstanciées	JavaScript pré-instancie deux classes particulières afin qu'elles puissent être utilisées lors de l'exécution. Les instances se nomment Global et Math. Global possède un fonctionnement particulier, que nous décrirons par la suite.

3.1. Classes de base

JavaScript offre différentes classes correspondant à la représentation objet des différents types primitifs et de certains éléments du langage, ainsi qu'à la classe racine de tous les objets et à différentes classes d'erreur.

Le tableau suivant récapitule l'ensemble de ces classes de base.

CLasses	Description
Array	Représentation sous forme d'objet d'un tableau JavaScript
Boolean	Représentation sous forme d'objet du type primitif booléen
Date	Classe représentant une date
Error	Classe d'erreur générique
EvalError	Classe relative aux erreurs survenant lors de l'interprétation de code JavaScript par l'intermédiaire de la fonction eval
Function	Représentation sous forme d'objet d'une fonction JavaScript

Number	Représentation sous forme d'objet d'un nombre
Object	Classe de base de toutes les classes
RangeError	Classe relative aux erreurs survenant lors de l'utilisation de nombres excédant la plage autorisée (supérieurs à MAX_VALUE ou inferieurs à MIN_VALUE)
ReferenceError	Classe relative aux erreurs survenant lors de l'utilisation d'une référencées incorrecte
RegExp	Classe permettant d'utiliser des expressions régulières
String	Classe représentant une chaîne de caractères
SyntaxError	Classe relative aux erreurs de syntaxe
TypeError	Classe relative aux erreurs de typage
URIError	Classe relative aux erreurs survenant lors d'une mauvaise utilisation des méthodes de traitements des URI de l'objet Global

3.1.1. La classe racine *Object*

La classe racine de tous les objets est la classe Object. Cette classe, la plus simple du langage, fournit ses attributs et méthodes à toutes les autres classes JavaScript. Les tableaux suivants récapitulent respectivement ses attributs et ses méthodes.

Attribut	Description
constructor	Référence la méthode utilisée pour créer une instance de la classe.
prototype	Sert à initialiser l'objet lors de son instanciation. Cet attribut est très important pour mettre en œuvre ses propres classes et l'héritage en JavaScript. Nous détaillons un peu plus loin dans ce chapitre la façons de l'utiliser.

Méthode	Paramètre	Description
hasOwnProperty	Nom de propriété	Permet de déterminer si l'objet possède un attribut ou une méthode dont le nom correspond au paramètre.

isPrototypeOf	Classe	Permet de déterminer si un objet est le prototype d'un autre.
propertyIsEnumerable	Nom de la propriété de l'objet	Permet de déterminer si l'attribut dont le nom est passé en paramètre peut être énuméré par l'intermédiaire d'une boucle for...in.
toString	-	Représentation de l'objet sous forme de chaîne de caractères
valueOf	-	Valeur associée à l'objet. Cette méthode est utile dans le cas d'objets relatifs à des types primitifs. Sinon, elle renvoie la même valeur que la méthode toString.

La classe Object permet de définir facilement des instances vides, comme dans le code suivant :

var o = new Object();

L'objet précédemment créé est équivalent à l'objet littéral suivant:

var o = {};

3.1.2. Classes relatives aux types primitifs

Les classes relatives aux types primitifs du langage JavaScript sont les classes **Boolean, Number** et **String**.

Les méthodes de ces classes peuvent être également utilisées sur les types primitifs, considérés comme des pseudo-objets par JavaScript.

3.1.3. Classes d'éléments du langage

JavaScript permet de mettre en œuvre des tableaux. Le langage permet également de construire des tableaux par l'intermédiaire de la classe **Array**.

3.1.4. Classes pré-instanciées

JavaScript pré-instancie deux classes pour ses applications, dont les noms de leurs instances

sont **Global** et **Math**.

Global est utilisée implicitement par le langage, dont les méthodes « globales » ne sont apparemment rattachées à aucune instance. Cette dernière ne peut donc être utilisée explicitement.

Les méthodes de l'instance **Global** sont les suivantes :

- **eval() :** afin d'interpréter une chaîne de caractères contenant du JavaScript.
- **parseXXX() :** permet de convertir des chaînes de caractères en des types primitifs. Par exemple, pour les entiers, la méthode est **parseInt**.
- Méthodes de type **isXXX()** : permettent de vérifier la véracité d'un objet pour un critère, comme, pour les nombres, les méthodes **isFinite()** ou **isNaN**
- Méthodes **escape()** et **unescape()** : afin de convertir des chaînes pour HTML
- Constructeurs des classes relatives aux types primitifs, décrites précédemment

Toutes ces méthodes sont rattachées à l'instance **Global.** Bien que cette dernière ne soit pas accessible ni utilisable directement dans les applications, ses méthodes sont utilisables telles quelles, sans avoir à être préfixées par **Global.**

L'instance Math offre des constantes mathématiques ainsi que des méthodes permettant de réaliser des opérations mathématiques.

Le tableau suivant récapitule les principales d'entre elles.

Méthode	Paramètre	Description
Abs	Un nombre	Retourne la valeur absolue d'un nombre.
acos, asin, atan	Un nombre	Permettent de calculer respectivement les arc cosinus, sinus et tangente en radians.
Ceil	Un nombre	Retourne l'arrondi supérieur d'un nombre.
cos, sin, tan	Un nombre	Permettent de calculer les cosinus, sinus et tangente d'un nombre.
exp, log	Un nombre	Permettent de calculer respectivement l'exponentiel et le logarithme d'un nombre.

Floor	Un nombre	Retourne l'arrondi inferieur d'un nombre.
Max	Deux nombres	Retourne la plus grande valeur des deux paramètres.
Min	Deux nombres Retourne	la plus petite valeur des deux paramètres.
pow, sqrt	Un nombre	Permettent de calculer respectivement le carré et la racine carrée d'un nombre.
Random	-	Retourne un nombre pseudo-aléatoire.
Round	Un nombre	Retourne l'arrondi d'un nombre.

Le code suivant illustre l'utilisation des trois premières :

```
var nombre = 34.567;
// Affiche 35
alert("Ceil : "+Math.ceil(nombre));
// Affiche 34
alert("Floor : "+Math.floor(nombre));
// Affiche 35
alert("Round : "+Math.round(nombre));
```

3.2. Mises en œuvre de classes personnalisées

L'avantage d'un langage supportant les concepts de base de la programmation orientée objet est qu'il permet au développeur de créer ses propres classes afin de structurer une application.

JavaScript offre cette possibilité, bien qu'il ne supporte pas vraiment la notion de classe, puisque aucun élément de langage spécifique ne permet de définir une classe. Leur mise en œuvre est cependant possible en utilisant le support des **fonctions** et des **closures**, décrites au chapitre précédent et en s'appuyant sur le concept de prototype décrit dans cette section.

JavaScript ne fournissant pas une manière unique de mettre en œuvre les concepts de la programmation objet, plusieurs techniques permettent d'implémenter des classes et de définir des liens d'héritage.

3.2.1. Mise en œuvre de classes

Rappelons que JavaScript ne permet pas d'implémenter tous les mécanismes objet décrit en début de chapitre, puisqu'il ne fournit pas de mot-clé class afin de créer une classe.

De plus, le support de la visibilité de type privée n'est que partiel et dépend de la façon dont sont implémentées les classes. Dans tous les cas, le langage supporte la visibilité de type public, sans toutefois garantir les principes d'encapsulation.

3.2.1.1. Le mot-clé this

Le langage JavaScript fournit le mot-clé **this**, qui offre une référence, dans une fonction, à l'objet sur lequel cette dernière est appelée. De ce fait, il est possible d'avoir accès à tous les attributs et méthodes de l'objet dans le code d'une fonction.

Un fonction peut être affectée à une variable. Elle peut également être rattachée à un objet en l'affectant à un attribut d'un objet. Le point important à comprendre est qu'à ce moment-là, l'objet référencé par le mot-clé **this** change pour correspondre à l'objet dont dépend maintenant la fonction.

Le code suivant illustre la mise en œuvre de ce mécanisme :

```
function maFonction() {
        return this.monAttribut;
}
```

Le code de la fonction **maFonction()** permet de renvoyer la valeur de l'attribut **monAttribut** de l'objet appelant la méthode. Nous pouvons remarquer que la fonction n'a pas besoin d'être rattachée à une classe lors de sa création. Par contre, lors de son exécution, elle doit être exécutée sur un objet sous peine que la valeur de **this.monAttribut** soit **undefined**.

Le code suivant décrit l'utilisation de la fonction **maFonction()** précédente pour un objet :

```
function maFonction(msg) {
        return msg+" : "+this.monAttribut;
}
var premierObjet = new Object();
premierObjet.monAttribut = "mon attribut";
premierObjet.maMethode = maFonction;
var retour = premierObjet.maMethode("Valeur de l'attribut");
var secondObjet = new Object();  secondObjet.maMethode = test ;
secondObjet.maMethode("Valeur de l'attribut");
```

3.2.1.2. Appels de fonctions d'un objet

Comme nous l'avons vu précédemment, JavaScript gère les fonctions en tant qu'objets de type Function. Cette classe possède les méthodes call et **apply**, qui se révèlent particulière- ment utiles lorsqu'elles sont utilisées conjointement avec le mot-clé **this**. Elles permettent alors d'exécuter une fonction pour un objet donné sans affecter la fonction à l'objet.

Ces deux méthodes permettent également de passer des paramètres à ce moment et de retourner la valeur de retour de la fonction appelée.

L'unique différence entre ces méthodes réside dans la façon de passer les paramètres. La première, **call,** prend en paramètre l'objet cible suivi de la liste des paramètres de la fonction à appeler, comme le montre le code suivant :

```
function maMethode(msg) {
        return msg+" : "+this.monAttribut;
}
var o = new Object();
o.monAttribut = "mon attribut";
var retour = maMethode.call(o, "Valeur de l'attribut");
```

La seconde, **apply**, prend en paramètre **l'objet cible**, suivi d'un tableau contenant la liste des paramètres de la fonction à appeler, comme dans le code suivant :

```
function maFonction(msg) {
        return msg+" : "+this.monAttribut;
}
var o = new Object();
o.monAttribut = "mon attribut";
var parametres = ["Valeur de l'attribut"];
var retour = maFonction.apply(o, parametres);
```

3.2.1.3. L'attribut *prototype*

L'attribut prototype est un attribut particulier que possèdent toutes les classes JavaScript. Il est utilisé lors de l'instanciation des objets afin de spécifier un modèle structurel pour leur création. Les différents attributs et méthodes présents dans l'attribut sont assignés à l'objet nouvellement

créé.

L'attribut **prototype** permet ainsi de spécifier tous les attributs et méthodes communs à toutes les instances de la classe.

NB : Attribut prototype et bibliothèque prototype : Il ne faut pas confondre l'attribut prototype, qui est défini par le langage JavaScript, avec la bibliothèque prototype, qui étend le langage afin de le rendre plus facile à utiliser.

Cette fonctionnalité permet de faire pointer toutes les instances d'une classe sur les mêmes méthodes et évite de devoir dupliquer leurs codes lors des différentes instanciations des objets. Cela allège la mémoire des applications JavaScript fondées sur des objets.

L'attribut **prototype** offre également la possibilité d'affecter aux attributs des objets des valeurs par défaut.

Le code suivant illustre la mise en œuvre de cette technique afin d'ajouter une méthode à une classe (repère ❶) et de spécifier la valeur par défaut d'un attribut (repère ❷) pour une classe :

```
Object.prototype.maMethode = function() { ❶
        (...)
};
Object.prototype.monAttribut = "Valeur par défaut ";❷
var o = new Object();
o.maMethode();
```

L'utilisation de prototype peut se faire à n'importe quel moment dans un script JavaScript. Tous les objets dont l'attribut prototype de la classe correspondante a été modifié sont impactés, et ce, même s'ils ont été instanciés précédemment.

```
var monInstance = new Object();
 Object.prototype.maFonction = function() {
        return "test de late binding";
};
var retour = monInstance.maFonction();
```

3.2.1.4. Techniques de création de classes

Avec le langage JavaScript, il est possible de créer des classes en se fondant sur les

caractéristiques des fonctions et closures ainsi que sur le mot-clé **this** et l'attribut prototype.

Il existe plusieurs façons de créer des classes en JavaScript. Afin de décrire leur mise en œuvre, nous allons utiliser une classe StringBuffer, qui offre la possibilité de concaténer des chaînes de caractères.

Une classe peut être définie en se fondant uniquement sur les fonctions, les closures et le mot-clé this. La classe est alors définie par l'intermédiaire d'une fonction dont le nom est celui de la classe. Cette fonction correspond au constructeur. Elle peut contenir des **variables locales**, précédées du mot-clé **var**, et des **variables de classe**, correspondant aux attributs, préfixées par **this**. Ces variables peuvent correspondre aussi bien à des attributs (repère ❶) qu'à des méthodes (repères ❷), comme l'illustre le code suivant :

```
function StringBuffer() {
this.chainesInternes = []; ❶
this.append = function(chaine) { ❷
        this.chainesInternes.push(chaine);
};
this.clear = function() { ❷
        this.chainesInternes.splice(0, this.chainesInternes.length);
};
this.toString = function() { ❷
        return this.chainesInternes.join("");
};
}
```

L'utilisation de cette classe se réalise de la manière suivante :

```
var sb = new StringBuffer();
sb.append("première chaîne");
sb.append(" et ");
sb.append("deuxième chaîne ");
var resultat = sb.toString();
//resultat contient « première chaîne et deuxième chaine »
```

Le problème avec cette première approche est que le code de chaque méthode est dupliqué à chaque instanciation. Comme indiqué précédemment, l'attribut prototype permet de résoudre ce problème. La technique décrite ici ne peut cependant être utilisée que lorsque peu d'instances de la classe sont utilisées ou si des attributs ou des méthodes privées doivent être mis en œuvre.

Une bonne pratique consiste à définir les méthodes de la classe précédente à l'aide de l'attribut prototype de la classe StringBuffer. Ainsi, le code de la classe est désormais le suivant :

```
function StringBuffer() {
        this.chainesInternes = [];
}
StringBuffer.prototype.append = function(chaine) {
        this.chainesInternes.push(chaine);
};
StringBuffer.prototype.clear = function() {
        this.chainesInternes.splice(0, this.chainesInternes.length);
};
StringBuffer.prototype.toString = function() {
        return this.chainesInternes.join("");
};
```

Comme les attributs ne sont pas partagés par les différentes instances de la classe, ils sont toujours définis dans le constructeur qui a la responsabilité de les initialiser spécifiquement pour les instances. Leur définition avec l'attribut prototype permet cependant de leur affecter une valeur par défaut.

Cette façon de définir une classe à l'aide de l'attribut prototype est celle communément utilisée. Son principal inconvénient est de ne pas définir toute la classe dans le constructeur. Il est possible de l'améliorer afin de spécifier les méthodes sur l'attribut prototype lors d'un premier appel au constructeur.

Nous utilisons pour cela une propriété du constructeur, comme l'illustre le code suivant :

```
function StringBuffer() {
this.chainesInternes = [];
        if( typeof StringBuffer.initialized == "undefined" ) {
                StringBuffer.prototype.append = function(chaine) {
                                this.chainesInternes.push(chaine);
                };
                StringBuffer.prototype.clear = function() {
                        this.chainesInternes.splice(0, this.chainesInternes.length);
                };
                StringBuffer.prototype.toString = function() {
                        return this.chainesInternes.join("");
                };
                StringBuffer.initialized = true;
        }
}
```

Il est également possible d'initialiser l'attribut prototype grâce à un objet. Ce dernier peut être défini sous forme littérale, comme nous le verrons à la section relative à JSON. Nous verrons également que certaines techniques de mise en œuvre de l'héritage s'appuient sur ce principe.

3.2.1.5. Classes et visibilité

Comme indiqué précédemment, le niveau de visibilité public est le seul supporté complètement par le langage JavaScript.

Il est néanmoins possible de supporter le niveau de visibilité privé dans certains cas. En JavaScript, les attributs et méthodes privés doivent nécessairement être définis dans le constructeur de la classe en tant que variables. Afin que les méthodes publiques puissent les utiliser, il est indispensable que toute la classe soit définie dans son constructeur. D'après la section précédente, deux approches vérifient cette condition.

La première n'utilise pas l'attribut prototype et définit attributs et méthodes directement dans le constructeur. Les méthodes privées sont alors des méthodes internes au constructeur et ne sont pas rattachées à une propriété de la classe. Les attributs privés sont eux définis en tant que variables locales simples du constructeur.

Le code suivant illustre la mise en œuvre de la classe **StringBuffer** précédente avec des attributs et des méthodes privées :

```
function StringBuffer() {
var chainesInternes = [];¨1
function addInInternalTable(chaine) {¨2
        chainesInternes.push(chaine);
}
function clearInternalTable() {¨2
        chainesInternes.splice(0,
        this.chainesInternes.length);
}
this.append = function(chaine) {¨3
        addInInternalTable(chaine);
};
this.clear = function() {¨3
        clearInternalTable();
};
this.toString = function() {¨4
        return chainesInternes.join("");
};
}
```

Le repère ❶ identifie l'attribut privé de la classe, puisqu'il est défini en tant que variable pour le constructeur par l'intermédiaire du mot-clé var. Les repères ❷ indiquent les méthodes privées, puisqu'elles sont définies uniquement par le mot-clé function et ne sont pas affectées à des variables préfixées par this. Les méthodes identifiées par le repère ❸ sont publiques et utilisent une méthode privée. La méthode identifiée par le repère ❹ est publique et utilise un attribut privé.

La seconde approche s'appuie sur l'attribut prototype, mais son initialisation s'effectue dans le constructeur de la classe.

```
function StringBuffer() {
        var chainesInternes = [];①
function addInInternalTable(chaine) {②
        chainesInternes.push(chaine);
}
function clearInternalTable() {②
        chainesInternes.splice(0,
        this.chainesInternes.length);
}
if( typeof StringBuffer.initialized == "undefined" ) {
        StringBuffer.prototype.append = function(chaine) {③
                addInInternalTable(chaine);
        };
        StringBuffer.prototype.clear = function() {③
                clearInternalTable();
        };
        StringBuffer.prototype.toString = function() {④
                return chainesInternes.join("");
        };
        StringBuffer.initialized = true;

}

}
```

NB : Notons que les techniques utilisant l'attribut prototype à l'extérieur du constructeur ne peuvent mettre correctement en œuvre des éléments avec une visibilité autre que publique. Dans ce cas, les méthodes publiques ne pourraient pas appeler les méthodes privées.

3.2.2. L'héritage de classes

À l'instar de la création de classes, le langage JavaScript ne définit pas de mot-clé spécifique (par exemple**, extends**) afin de réaliser l'héritage de classes. De ce fait, il existe plusieurs manières de faire hériter une classe d'une autre, dont certaines permettent de mettre en œuvre l'héritage multiple.

Comme nous l'avons vu à la section précédente, les éléments d'une classe sont définis à deux niveaux :

- Les attributs au niveau du constructeur, puisque leurs valeurs sont spécifiques aux instances. Leurs valeurs par défaut peuvent être cependant spécifiées dans l'attribut

prototype.

- Les méthodes au niveau du constructeur ou de l'attribut prototype, avec les implications décrites précédemment.

Comme JavaScript comprend les objets comme une liste non ordonnée d'attributs et de méthodes, la mise en œuvre de l'héritage correspond à une copie de ces derniers de la ou des classes mères dans la sous-classe.

3.2.2.1. Appel du constructeur de la classe mère

La première technique permettant de mettre en œuvre l'héritage en JavaScript consiste à s'appuyer sur le constructeur de la classe mère pour initialiser la sous-classe. Elle s'appuie sur le fonctionnement du mot-clé this, qui référence l'objet sur lequel est appelée une méthode Il s'agit d'affecter la fonction correspondant au constructeur de la classe mère à une méthode quelconque de la classe fille. De ce fait, l'appel de cette fonction initialise cette dernière classe en se fondant sur le constructeur de la classe mère. Le code suivant illustre la mise en œuvre de ce mécanisme :

```
function StringBuffer() {
   this.chainesInternes = [];
}
StringBuffer.prototype.append = function(chaine) {
   this.chainesInternes.push(chaine);
};
StringBuffer.prototype.clear = function() {
   this.chainesInternes.splice(0, this.chainesInternes.length);
};
StringBuffer.prototype.toString = function() {
   return this.chainesInternes.join("");
};
function ExtendedStringBuffer() {
   this.parent = StringBuffer; ❶      this.parent();
   delete this.parent;
   (...)
}
```

Le repère ❶ indique l'utilisation du constructeur de la clase mère.

Notons qu'il convient de libérer la variable contenant le constructeur de la classe mère puisqu'elle ne correspond pas à un élément de la classe fille. Cette variable n'est d'ailleurs plus utilisée par la suite. Dans l'exemple ci-dessus, une fois la méthode parent appelée, la classe **ExtendedStringBuffer** possède un attribut nommé **chainesInternes**.

Cette technique est fastidieuse et peu naturelle. L'utilisation des méthodes call et **apply** des fonctions s'avère beaucoup plus intéressante en ce qu'elles offrent la possibilité de définir au moment de l'exécution l'objet sur lequel est appelée la méthode.

Le code précédent peut être amélioré de la manière suivante :

```
function StringBuffer() {
        this.chainesInternes = [];
}
(...)
function ExtendedStringBuffer() {
        StringBuffer.call(this);¨❶
        (...)
}
```

Le repère ❶ indique toujours l'utilisation du constructeur de la clase mère.

Dans le cas où les méthodes de la classe StringBuffer sont définies par l'intermédiaire de l'attribut prototype en dehors du constructeur, ces dernières ne sont pas héritées avec cette technique.

3.2.2.2. Utilisation du prototype de la classe mère

L'utilisation de l'attribut **prototype** afin de mettre en œuvre l'héritage peut se réaliser de deux manières :

- En écrasant le contenu de l'attribut **prototype** de la classe fille avec une instance de la classe mère ;
- En recopiant les éléments de l'attribut **prototype** de la classe mère dans celui de la classe fille.

La première technique correspond au chaînage de prototype et consiste à initialiser l'attribut prototype de la classe fille avec une instance de la classe mère. Elle permet d'affecter tous les éléments contenus dans la classe mère à l'attribut **prototype** de la classe fille. Par contre, elle ne permet pas de supporter l'héritage multiple.

Le code suivant illustre sa mise en œuvre :

```
function StringBuffer() {
        this.chainesInternes = [];
}
function ExtendedStringBuffer() {
        (...)
}
ExtendedStringBuffer.prototype = new StringBuffer();
```

L'ordre dans lequel sont réalisés les différents traitements peut avoir une incidence importante sur les méthodes de la classe fille. En effet, les méthodes de la classe fille ajoutées à l'attribut prototype avant sa réinitialisation ne sont pas présentes au final.

Le code suivant illustre ce problème :

```
function StringBuffer() {
        this.chainesInternes = [];
}
function ExtendedStringBuffer() {
        (...)
}
ExtendedStringBuffer.prototype.uneMethode = function() {
        (...)
}
ExtendedStringBuffer.prototype = new StringBuffer();
//La méthode uneMethode n'existe plus dans le prototype
```

La seconde technique consiste à copier manuellement les éléments présents dans l'attribut prototype de la classe mère dans celui de la classe fille.

Le code suivant illustre la mise en œuvre de ce principe :

```
function StringBuffer() {
        this.chainesInternes = [];
}
(...)
function ExtendedStringBuffer() {
        (...)
}
for( element in StringBuffer.prototype ) {
        ExtendedStringBuffer.prototype[element] = StringBuffer.prototype[element];
}
```

Cette technique ne permet pas à la classe fille d'hériter des éléments de la classe définis dans le constructeur de la classe mère. Il convient donc de combiner l'approche fondée sur le constructeur de la classe mère avec celle fondée sur l'attribut prototype.

3.2.2.3. Combinaison des deux techniques

Afin de réaliser efficacement l'héritage de classes avec JavaScript, la combinaison des deux techniques précédentes est nécessaire.

Le code suivant en donne un exemple de mise en œuvre :

```
//Définition de la classe mère
function StringBuffer() {
        this.chainesInternes = [];
}
//Spécification des méthodes de la classe mère
StringBuffer.prototype.append = function(chaine) {
        this.chainesInternes.push(chaine);
};
StringBuffer.prototype.clear = function() {
        this.chainesInternes.splice(0, this.chainesInternes.length);
};
StringBuffer.prototype.toString = function() {
        return this.chainesInternes.join("");
};
```

```
//Définition de la classe fille function
ExtendedStringBuffer() {
        StringBuffer.call(this); ❶
}
//Copie des méthodes de la classe mère
for( var element in StringBuffer.prototype ) { ❷
        ExtendedStringBuffer.prototype[element]  = StringBuffer.prototype[element];
}
//Spécification des méthodes de la classe fille
StringBuffer.prototype.append = function(chaine) {
        this.chainesInternes.push(chaine);
};
```

L'appel du constructeur de la classe mère est réalisé dans le constructeur de la classe fille (repère ❶). La copie des éléments de l'attribut prototype de la classe mère dans celle de la classe fille se fait au repère ❷.

Cette implémentation n'est pas encore complètement satisfaisante, car le code des différentes classes n'est pas regroupé dans leurs constructeurs respectifs. Avec la technique permettant de spécifier les valeurs de l'attribut prototype dans le constructeur, la spécification de l'héritage est plus concise.

Comme cette opération ne doit être réalisée qu'une seule fois, nous utilisons une propriété rattachée au constructeur afin de détecter si elle a déjà été réalisée.

Le code suivant illustre la mise en œuvre de ce mécanisme :

```
//Définition de la classe mère
function StringBuffer() {
	this.chainesInternes = [];
//Spécification des méthodes de la classe mère
	if( typeof  StringBuffer.initialized == "undefined" ) {
		StringBuffer.prototype.append = function(chaine) {
			this.chainesInternes.push(chaine);
		};
		StringBuffer.prototype.clear = function() {
			this.chainesInternes.splice(0, this.chainesInternes.length);
		};
		StringBuffer.prototype.toString = function() {
			return this.chainesInternes.join("");
		};
		StringBuffer.initialized = true;
	}

}
```

```
//Définition de la classe fille
function ExtendedStringBuffer() {
	//Appel du constructeur de la classe mère
	StringBuffer.call(this);
	//Spécification des méthodes de la classe fille
	if( typeof ExtendedStringBuffer.initialized == "undefined" ) { ❷
		//Copie des méthodes de la classe mère
		for( var element in StringBuffer.prototype ) { ❶
			ExtendedStringBuffer.prototype[element]  =
StringBuffer.prototype[element];
		}
		ExtendedStringBuffer.prototype.append = function(chaine) {
			this.chainesInternes.push(chaine);
		};
		ExtendedStringBuffer.initialized = true; ❷
	 }
}
```

La copie des éléments de l'attribut prototype de la classe mère dans la classe fille se réalise désormais dans le constructeur de cette dernière (repère ❶). Elle n'est réalisée qu'une seule fois grâce à la propriété **initialized** rattachée au constructeur de la classe **ExtendedStringBuffer** (repères ❷).

3.2.2.4. L'héritage multiple

L'héritage multiple peut être mis en œuvre avec JavaScript de la même manière que l'héritage simple. Il faut simplement prendre garde de ne pas écraser l'attribut **prototype** lors de la réalisation des différents héritages.

Une bonne pratique consiste à copier les différents éléments contenus dans les attributs **prototype** des classes mères dans l'attribut **prototype** de la classe fille.

Le code suivant illustre la mise en œuvre de l'héritage multiple pour la classe **ExtendedStringBuffer**, dont les classes mères sont StringBuffer et **StringSpliter** :

```
function StringBuffer() { ❶
	this.chainesInternes = [];
	//Spécification des méthodes de la première classe mère
	if( typeof StringBuffer.initialized == "undefined" ) {
		StringBuffer.prototype.append = function(chaine) {
			this.chainesInternes.push(chaine);
		};
		StringBuffer.prototype.clear = function() {
			this.chainesInternes.splice(0, this.chainesInternes.length);
		};
		StringBuffer.prototype.toString = function() {
			return this.chainesInternes.join("");
		};
		StringBuffer.initialized = true;
	}
}
function StringSpliter() { ❷
	if( typeof StringSpliter.initialized == "undefined" ) {
		StringSpliter.prototype.split = function(chaine) {
			this.chainesInternes = chaine.splite(" ");
		};
		StringSpliter.initialized = true;
	}
}
```

```
function ExtendedStringBuffer() { ❸
        StringBuffer.call(this); ❹
        StringSpliter.call(this); ❹
        if( typeof ExtendedStringBuffer.initialized == "undefined" ) { ❹
                //Copie des méthodes de la première classe mère
                for( var element in StringBuffer.prototype ) {
                        ExtendedStringBuffer.prototype[element] =
                        StringBuffer.prototype[element];
                }
                //Copie des méthodes de la seconde classe mère
                for( var element in StringSpliter.prototype ) {
                        ExtendedStringBuffer.prototype[element] =
                        StringSpliter.prototype[element];
                }
                ExtendedStringBuffer.prototype.append = function(chaine) {
                        this.chainesInternes.push(chaine);
                };
        }
}
```

Les repères ❶, ❷ et ❸ pointent respectivement sur les définitions des classes mères StringBuffer et **StringSpliter** et de la classe fille **ExtendedStringBuffer**. Les repères ❹ illustrent l'appel aux constructeurs des classes mères ainsi que l'initialisation de l'attribut prototype de la classe fille à partir de ceux des classes mères.

Rappelons que l'héritage multiple doit être utilisé avec prudence puisqu'il peut introduire des problèmes dans certains cas.