10

Programmation orientée objet en JavaScript

Au sommaire de ce chapitre

- Du modèle procédural...
- ...à la programmation orientée objet
- Mettre en œuvre l'héritage
- Le contexte d'exécution
- Accéder aux propriétés et aux méthodes
- Définir des méthodes d'accès
- Isolation du code et bibliothèques
- Perfectionner l'animation de l'indicateur d'activité

Les chapitres précédents le laissent deviner, il est fort probable que JavaScript se retrouve au cœur du développement de vos applications web. Au fil du temps, les scripts côté client sont devenus un outil essentiel pour les développeurs, en raison des avancées matérielles, de l'évolution des attentes et, récemment, de nouvelles API qui apportent au contenu web les fonctionnalités recherchées.

Toutefois, pour employer efficacement ce langage, le développeur doit en comprendre certaines particularités, comme son typage faible et son modèle d'objet. Nous allons nous attarder sur l'aspect programmation orientée objet, car elle vous permettra de rendre vos programmes plus modulaires, plus souples et génériques grâce à une encapsulation de certaines API, et plus faciles à maintenir. Ce chapitre vous aidera également à comprendre les problèmes de contexte d'exécution que vous rencontrerez certainement lors de l'utilisation des gestionnaires d'événements ou des fonctions de rappel, deux constructions que nous emploierons énormément dans les prochains chapitres. Vous y découvrirez également comment résoudre efficacement les problèmes d'isolation de l'exécution du code lorsque vous devez mettre en œuvre des scripts tiers pour améliorer vos applications web ou lorsque vous proposez vos scripts à d'autres sites.

Du modèle procédural...

Le modèle de programmation procédurale est encore très répandu aujourd'hui. Comme en BASIC, il consiste à écrire des fonctions qui contrôlent l'état du programme en modifiant les variables et en changeant leur valeur. Voici la manière de procéder la plus courante :

```
var state = add(5, 6, 7);

function add(a, b, c) {
    return a + b + c;
}
```

Cette approche tend à multiplier le nombre de fonctions et de variables indépendantes et éventuellement redondantes. Pourtant, en JavaScript, il s'agit déjà de programmation orientée objet. En effet, lorsque vous définissez une fonction, le moteur de script crée une instance de l'objet `Function`. Voici donc l'équivalent objet du code précédent :

```
var add = new Function("a", "b", "c", "return a + b + c");
```

Le constructeur de `Function` prend en arguments une suite de chaînes de caractères, la dernière correspondant au code à exécuter, les premières, aux paramètres de la fonction définie. Il existe plusieurs manières de déclarer ces paramètres. Ils peuvent être donnés comme précédemment ou être regroupés dans la même chaîne, en les séparant par des virgules, de manière à former la liste des paramètres de la fonction :

```
var add = new Function("a, b", "c", "return a + b + c");
var add = new Function("a, b, c", "return a + b + c");
```

Le mot clé `function` est un raccourci très utile, car déclarer une fonction à l'intérieur d'une chaîne de caractères passée en paramètre à l'objet `Function` devient vite lourd.

En utilisant l'instanciation de l'exemple précédent, nous obtenons assez naturellement un code semblable au suivant :

```
var add = function(a, b, c) {
    return a + b + c;
}
```

Notez que vous pouvez également donner un nom à votre fonction afin de pouvoir y faire référence depuis l'intérieur de la fonction. Ce type de déclaration constitue la base de la création des objets.

...à la programmation orientée objet

La déclaration d'une fonction déclenche la création d'un objet interne associé à l'objet global de l'environnement d'hébergement. Avec JavaScript, dans le contexte du navigateur, il s'agit de l'objet `DOMWindow` du navigateur, accessible au travers de la propriété `window`. Autrement dit, la déclaration d'une fonction étend l'objet `DOMWindow` et elle devient accessible au travers de la propriété `window`, comme le serait n'importe quelle autre variable :

```
var state = window.add(5, 6, 7);
var result = window.state;
```

Grâce à cette possibilité d'ajouter des propriétés et des méthodes aux objets, nous allons créer un programme simple qui illustre les atouts de la programmation orientée objet.

Un premier objet personnalisé

En programmation orientée objet, tous les objets dérivent de l'objet de base `Object`, comme c'est le cas dans .NET ou Java. Toutes les propriétés et méthodes de l'objet `Object` sont donc également disponibles dans l'objet dérivé ; la méthode `toString()` en est un bon exemple. Par conséquent, la création de notre objet débute par une instanciation d'`Object` à l'aide du mot clé `new`. Nous pouvons ensuite l'étendre avec le code suivant :

```
var animal = new Object();

/* Propriétés supplémentaires. */
animal.family = "Sans nom";
animal.noise = "silencieux";
animal.legs = 4;

/* Nouvelle méthode. */
animal.makeNoise = function() {
    console.log("Cri de l'animal " + this.family + " : " + this.noise);
}
```

L'invocation de la méthode et la lecture des propriétés de l'objet ne posent aucune difficulté :

```
animal.makeNoise();
console.log(animal.legs);

--- résultat ---
> Cri de l'animal Sans nom : silencieux
> 4
```

Dans nos exemples, nous adoptons les conventions de nommage du langage, c'est-à-dire l'écriture Camel Case, avec une première lettre en majuscule pour la définition de l'objet et une première lettre en minuscule pour les propriétés.

Le mot clé `this` donne accès au contexte d'exécution courant. Dans le code précédent, `this` correspond à l'instance de l'objet. Il nous permet d'atteindre les propriétés définies et de les employer pour construire une chaîne de caractères. Le contexte d'exécution est une notion importante sur laquelle nous reviendrons plus en détail bientôt.

Voici une autre manière d'écrire le code de l'exemple précédent :

```
var animal = {
    family: "Sans nom",
    noise: "silencieux",
    legs: 4,

    makeNoise: function() {
        console.log("Cri de l'animal " + this.family + " : " + this.noise);
    }
};
```

Cette façon de créer des objets se nomme *initialiseur d'objet*. Elle permet de créer dynamiquement des objets anonymes et, par exemple, de retourner plusieurs valeurs à partir d'une fonction. Toutefois, elle n'offre pas la souplesse apportée par d'autres langages orientés objet, car la définition de l'objet n'est pas facilement réutilisable.

Utiliser un constructeur approprié

L'`animal` que nous avons créé n'est pas très vivant. Pour y remédier, nous allons rendre notre structure d'objet réutilisable en ajoutant un constructeur à l'aide d'un objet `Function`.

ECMAScript ne dispose pas de la notion de classes mais utilise des constructeurs pour allouer l'objet et initialiser ses propriétés. C'est précisément ce que fait le code suivant :

```
/* Constructeur. */
var Animal = function() {
    /* Propriétés. */
    this.family = "Sans nom";
    this.noise = "silencieux";
    this.legs = 4;

    /* Méthode. */
    this.makeNoise = function() {
        console.log("Cri de l'animal " + this.family + " : " + this.noise);
    }
}
```

Cela nous permet d'obtenir une instance d'`Animal`, de nouveau en utilisant le mot clé `new` :

```
/* Créer une première instance d'Animal. */
var animal1 = new Animal();
animal1.family = "Moule";
```

```
/* Créer une nouvelle instance d'Animal. */
var animal2 = new Animal();
animal2.family = "Plancton";

animal1.makeNoise();
animal2.makeNoise();

--- résultat ---
> Cri de l'animal Moule : silencieux
> Cri de l'animal Plancton : silencieux
```

Nous pouvons donc utiliser notre objet autant de fois que nécessaire avec des instances totalement indépendantes, chacune avec ses propres valeurs des propriétés. Toutefois, cette version n'est pas optimale car la syntaxe de déclaration des méthodes oblige le moteur à instancier un nouvel objet `Function` pour la méthode `makeNoise()` de chaque instance au lieu de faire référence à l'objet. L'occupation mémoire est alors plus importante, ce qui peut constituer un problème si l'objet est employé à de nombreuses reprises.

Meilleures performances avec les prototypes

ECMAScript ajoute la propriété `prototype` pour résoudre ce problème d'instanciations multiples. Chaque constructeur dispose de cette propriété pour mettre en œuvre l'héritage et les propriétés partagées. Voici donc comment améliorer notre objet :

```
/* Constructeur. */
var Animal = function() {
    this.family = "Sans nom";
    this.noise = "silencieux";
    this.legs = 4;
}

/* Une manière plus efficace de définir une méthode. */
Animal.prototype.makeNoise = function() {
    console.log("Cri de l'animal " + this.family + " : " + this.noise);
}
```

La méthode définie sera partagée par toutes les instances d'`Animal` et ne sera instanciée qu'une seule fois. Cela reste vrai pour toutes les propriétés définies de cette manière.

Voici une notion importante : toutes les propriétés définies à l'aide de `prototype` *seront partagées par toutes les instances de l'objet. Par conséquent, toute modification de la valeur d'une telle propriété aura un impact sur toutes les instances.*

Puisque cette syntaxe implique un code plus long, vous devez toujours rechercher un bon équilibre entre la concision du code et l'utilisation de la mémoire. Cependant, faire référence à l'objet `Function` au lieu d'en créer une instance permet d'améliorer les performances, car une opération d'allocation mémoire est coûteuse. L'Inspecteur web de Safari vous permettra d'étudier de plus près cet aspect (voir Chapitre 3).

Mettre en œuvre l'héritage

Bien que JavaScript soit un langage orienté objet, il se fonde non pas sur le concept de classe, comme C++ ou Java, mais sur les constructeurs et les prototypes, chaque constructeur disposant d'une propriété `prototype`. Le moteur affecte implicitement à cette propriété une référence à une instance d'`Object`, avec pour conséquence un héritage des caractéristiques d'`Object`. Cette instance sera partagée par toutes les instances de votre objet, ce qui explique pourquoi toutes les méthodes définies avec `prototype` seront également partagées (elles étendent `prototype`).

Héritage par prototype

Pour mettre en œuvre dans vos propres objets un héritage à base de prototype, vous devez simplement redéfinir `prototype` de manière à faire référence à votre objet plutôt qu'à `Object`. Voici par exemple comment créer un animal plus spécifique :

```
/* Constructeur. */
var Dog = function() {
    this.family = "Chien";
    this.noise = "aboiement";
};

/* Dog dérive d'Animal. */
Dog.prototype = new Animal;

/* Ajouter une nouvelle méthode à Dog. */
Dog.prototype.showLegs = function() {
    console.log("L'animal " + this.family + " a " + this.legs + " pattes");
}
```

Nous définissons un nouvel objet nommé `Dog`, dont nous modifions les propriétés par défaut dans le constructeur de manière à les adapter au chien. D'autres propriétés sont déjà définies dans le constructeur d'`Animal`, comme le montre le bloc de code suivant. Le script précédent commence par créer une instance d'`Animal` qui est affectée à la propriété `prototype` de l'objet `Dog` au moment de l'évaluation du code. L'appel au constructeur de `Dog` se fera uniquement après l'initialisation, à l'aide du mot clé `new` :

```
var dog = new Dog();
dog.makeNoise();
dog.showLegs();

--- résultat ---
> Cri de l'animal Chien : aboiement
> L'animal Chien a 4 pattes
```

La propriété `property` existe bien qu'elle n'ait pas été définie dans le constructeur de `Dog` et la méthode `makeNoise()` retourne une chaîne de caractères avec les valeurs appropriées pour `family` et `noise`. La preuve est faite que `Dog` hérite d'`Animal`.

Propriétés partagées

Certes, limiter l'utilisation mémoire et la multiplication des instances de fonctions est un objectif louable, mais, en utilisant l'héritage, vous risquez de constater amèrement que toutes les définitions des propriétés de l'objet hérité (tableaux, objets anonymes, etc.) suivent également ce principe et qu'elles sont donc partagées par toutes les instances de votre objet. Ajoutons, par exemple, un tableau à l'objet initial pour mémoriser la couleur des yeux d'un animal :

```
var Animal = function() {
    ...
    this.eyeColor = [];
}
```

Nous pouvons ensuite créer deux chiens avec des yeux de couleur différente :

```
var dog1 = new Dog();
dog1.eyeColor.push("marron", "marron");

var dog2 = new Dog();
dog2.eyeColor.push("bleu", "marron");

console.log(dog1.eyeColor);
console.log(dog2.eyeColor);

--- résultat ---
> ["marron", "marron", "bleu", "marron"]
> ["marron", "marron", "bleu", "marron"]
```

Vous le constatez, nous avons deux chiens, avec chacun pas moins de quatre yeux. Pour y remédier, nous pouvons définir une méthode `initialize()` qui initialisera les propriétés sensibles. Le code suivant permet de retourner des informations plus pertinentes :

```
var Animal = function() {
    ...
    this.initialize();
}

Animal.prototype.initialize = function() {
    this.eyeColor = [];
}

var Dog = function() {
    ...
    this.initialize();
}

--- résultat ---
> ["marron", "marron"]
> ["bleu", "marron"]
```

Cette méthode évite que l'objet dérivé ne récupère ce que son parent contient et initialise. La réinitialisation des propriétés peut se faire directement dans le constructeur de `Dog`, mais la facilité de maintenance du code en pâtirait car il faudrait corriger `Dog` après chaque modification de la définition d'`Animal`. Cependant, redéfinir une propriété dans un objet hérité ne signifie pas la perte de la valeur initiale :

```
Animal.prototype.initialize = function() {
    this.eyeColor = ["valeur initiale"];
}
...
console.log(dog1.eyesColor);
delete dog1.eyesColor;       // Supprimer la propriété créée avec initialize().
console.log(dog1.eyesColor); // Retourner la propriété d'Animal.
console.log(dog2.eyesColor); // Les autres instances ne sont pas affectées.

--- résultat ---
> ["marron", "marron"]
> ["valeur initiale"]
> ["bleu", "marron"]
```

L'objet `dog1` est instancié, ce qui crée une nouvelle propriété `eyeColor` sur le prototype. Le code précédent retourne donc la couleur correcte pour les yeux de notre premier chien. Ensuite, nous effaçons cette propriété à l'aide de l'opérateur `delete`. Lors de l'accès suivant à la propriété, le moteur de script remonte la hiérarchie de prototypes jusqu'à trouver une propriété de même nom. Si elle existe, il retourne alors sa valeur. C'est pourquoi nous obtenons le contenu du tableau défini dans `Animal`.

En général, ce problème d'initialisation n'apparaît pas avec les propriétés de type primitif, comme `Boolean` ou `Number`, car la technique la plus répandue consiste à redéfinir des valeurs directement sur la propriété. *A contrario*, il est fréquent d'employer des méthodes comme `push()` ou `unshift()` pour remplir un tableau ou comme `pop()` ou `shift()` pour en extraire des valeurs. Dans ce cas, il faut manipuler directement l'instance partagée de l'objet `Array`, non la propriété.

La chaîne des prototypes

ECMAScript définit le concept de *chaîne des prototypes*. Nous l'avons indiqué précédemment, la propriété `prototype` permet de remonter dans la chaîne des propriétés héritées par les objets. C'est précisément ce que fait le moteur de script lorsqu'il invoque une méthode. Ainsi, lors de l'appel à `dog.makeNoise()`, il vérifie si cette méthode est définie dans `Dog`. Dans la négative, il remonte la chaîne jusqu'à trouver, ou non, la méthode invoquée. Ce fonctionnement peut être exploité pour invoquer des méthodes sur l'objet de base. Pour cela, il suffit d'obtenir le prototype de base et d'appeler la méthode dans le contexte d'exécution adéquat :

```
var Dog = function() {
    this._base = this.constructor.prototype;
    ...
}
```

```
Dog.prototype.makeNoise = function() {
    console.log("Version modifiée...");
    this._base.makeNoise.call(this);
}

var dog = new Dog();
dog.makeNoise();

--- résultat ---
> Version modifiée...
> Cri de l'animal Chien : aboiement
```

Nous profitons de la propriété `constructor` interne pour accéder au prototype sans dépendre de l'objet initial. Pour la valeur de `_base`, nous aurions pu utiliser `Animal.prototype`. La propriété `_base` nous sert à invoquer `makeNoise()` à l'aide de la méthode `call()` de l'objet `Function`. Cette méthode nous permet de préciser le contexte d'exécution et donc de manipuler les valeurs des propriétés de l'objet courant, non celles de l'objet de base.

Le contexte d'exécution

En JavaScript, le contexte d'exécution est primordial. Il détermine les éléments accessibles au moment de l'exécution d'une partie du code et de l'accès aux fonctions. La méthode `call()` de l'objet `Function` attend au moins en premier paramètre un contexte d'exécution. Viennent ensuite d'autres paramètres séparés par des virgules et passés à la fonction invoquée.

Utiliser les méthodes *call()* et *apply()*

Lors de l'invocation d'une fonction, le nombre de paramètres à passer n'est pas toujours précisément connu. Par ailleurs, fixer explicitement les paramètres de la méthode `call()` ne plaide pas en faveur de la facilité de maintenance ni de la réutilisabilité du code. En voici un exemple :

```
Animal.prototype.initialize = function() {
    ...
    this.colors = [];
}

Animal.prototype.setColors = function(color1, color2) {
    this.colors.splice(0, this.eyesColor.length);
    this.colors.push(color1);
    this.colors.push(color2);
}

Dog.prototype.setColors = function(color1, color2) {
    console.log("Ajout de couleurs : " + color1 + ", " + color2);
    this._base.setColors.call(this, color1, color2);
}

dog.setColors("blanc", "marron");
```

Il montre clairement que l'approche n'est pas très souple, car seules deux couleurs peuvent être attribuées à un animal. Pour corriger le problème, il suffit d'utiliser un nombre variable de paramètres, dont l'analyse se fait dans une boucle avec la propriété `arguments` :

```
Animal.prototype.setColors = function() {
    this.colors.splice(0, this.eyesColor.length);

    for (var i = 0; i < arguments.length; i++) {
        this.colors.push(arguments[i]);
    }
}
```

Pour résoudre le problème rencontré avec la fonction `setColors()` de l'objet dérivé d'`Animal`, nous pouvons utiliser la méthode `apply()` de l'objet `Function`. À l'instar de `call()`, elle prend en argument le contexte d'exécution, mais, contrairement à `call()`, les paramètres sont donnés non plus dans une liste mais dans un tableau. Notre code peut ainsi être optimisé :

```
Animal.prototype.setColors = function() {
    this.colors.splice(0, this.eyeColor.length);
    this.colors.push.apply(this.colors, arguments);
}

Dog.prototype.setColors = function() {
    console.log("Ajout de couleurs : " + arguments.join(", "));
    this._base.setColors.apply(this, arguments);
}
```

Puisque nous connaissons le type des arguments attendus par la fonction, nous n'avons pas besoin de les indiquer dans sa déclaration. La seule inconnue est leur nombre. Dans la méthode `setColors()` de l'objet `Animal`, nous invoquons `apply()` sur `push()`, qui prend initialement en paramètre la liste des éléments à ajouter à un tableau. Dans ce cas, puisque le contexte n'a pas besoin d'être modifié mais précisé, nous utilisons `this.colors` pour le désigner.

Ensuite, nous simplifions l'appel à la méthode de base depuis `Dog` en remplaçant `call()` par `apply()`, car le nombre de paramètres n'a alors plus aucune importance. Nous sommes donc en mesure d'appeler n'importe quelle méthode de l'objet de base quelle que soit sa signature.

La méthode `apply()` sera très utile dans le développement des applications web dès lors qu'elles se fondent sur des listes dont la taille varie dynamiquement. Elle peut par exemple servir à gérer un tableau d'objets dont les données permettent d'afficher une liste de courriers électroniques et de laisser l'utilisateur consulter d'autres messages en utilisant un bouton MESSAGES SUIVANTS, comme dans l'application Mail de l'iPhone. Dans ce cas, `apply()` supprime la boucle car les objets peuvent être simplement ajoutés à la liste à l'aide de la méthode `push()` de l'objet `Array`.

Cette solution est plus efficace, car la procédure est gérée en natif par le moteur de script et n'a pas besoin d'être réalisée dans une boucle.

Prendre soin du contexte d'exécution

Tout cela est plutôt simple, mais il est parfois difficile de déterminer l'origine d'un dysfonctionnement dans les parties du code, notamment lorsqu'elles impliquent des références à des méthodes. Par exemple, en utilisant la première définition de `Dog`, vous pourriez penser que la fonction `funcRef()` retourne la même chose que précédemment.

```
var funcRef = dog.makeNoise;
funcRef();

--- résultat ---
> Cri de l'animal undefined : undefined
```

Ce n'est pas le cas car les propriétés ne sont pas disponibles et ont donc la valeur `undefined`. En effet, la référence est sur la fonction `makeNoise()`. Même si `makeNoise()` est désignée au travers d'une instance de `Dog`, elle n'est liée à aucun contexte. Dans ce cas, `dog.makeNoise` est une référence sur une instance de la fonction.

En appelant `dog.makeNoise()`, le contexte est fixé par `dog`. Puisque la méthode est invoquée sur `dog` et que le contexte correspond à l'appelant de la fonction, il s'agit de l'instance de `Dog` et le résultat est celui attendu.

En revanche, lors de la définition de `funcRef`, `dog` n'a servi qu'à accéder à la référence sur l'instance de la fonction. Le moteur a ensuite quitté ce contexte (`dog`) pour exécuter l'instruction suivante. Lors de l'appel à `funcRef()`, le *contexte appelant* correspond au contexte par défaut, c'est-à-dire l'objet global `window`. Voici comment le vérifier :

```
...
var family = "Oiseau";
funcRef();

--- résultat ---
> Cri de l'animal Oiseau : undefined
```

Ce type de problème est récurrent dans les applications web, car les références sont souvent employées pour définir des gestionnaires d'événements et des minuteurs. En utilisant `addEventListener()`, le contexte est toujours celui de l'objet auquel l'écouteur est associé. Avec `setTimeout()` et `setInterval()`, qui sont des méthodes de l'objet `window`, le contexte sera toujours `window`.

Fixer le contexte approprié avec les gestionnaires et les fonctions de rappel

Au Chapitre 7, nous avons créé un indicateur d'activité à l'aide d'un canevas et de JavaScript. Dans les fonctions écrites, vous aurez peut-être noté que nous avons placé la valeur de `this` dans la variable `that`, puis encapsulé l'appel dans une fonction anonyme. Cette technique permet d'activer le contexte approprié et donc d'exécuter la méthode sur l'instance convenable. Voici, page suivante, la partie correspondante du code, la méthode `animate()` et l'objet `BigSpinner` :

```
BigSpinner.prototype.animate = function() {
    /* Déjà en cours d'exécution ? sortir... */
    if (this.timer) {
        return;
    }

    /* Le contexte d'exécution (this) est l'objet window avec setInterval().
       Mémoriser le contexte approprié dans la variable "that" et démarrer
       le minuteur. */
    var that = this;
    this.timer = window.setInterval(function() {
        that.draw();
    }, 100);
}
```

La variable `that` est locale. Mais, puisqu'elle est référencée dans le minuteur, elle est conservée en mémoire et reste valide jusqu'à ce que le minuteur soit annulé par la méthode `clearInterval()`. Il s'agit d'une solution classique pour appeler une méthode par référence. Pour faciliter l'emploi de cette technique dans votre code, nous pouvons ajouter une nouvelle méthode à l'objet natif `Function` de manière à lier une exécution à un contexte précis. Notez que tous les objets natifs peuvent être étendus. Nous devons également vérifier l'existence d'une telle méthode, car elle est prévue dans la cinquième version des spécifications ECMAScript. Voici notre version simplifiée :

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(ctx) {
        var that = this;

        return function() {
            that.apply(ctx, arguments);
        }
    }
}
```

Aussi pratique que cette solution puisse être, sachez qu'un nouvel objet `Function` est instancié lors de chaque appel à cette fonction. Par conséquent, si vous devez utiliser la référence à de nombreuses reprises, associez-la à une variable ou à une propriété de manière à réduire la consommation mémoire. Prenez également en compte le fait que la mémorisation de la référence est indispensable pour supprimer un gestionnaire d'événements. Le code suivant ne produit pas le résultat attendu, car les instances passées aux méthodes `addEventListener()` et `removeEventListener()` diffèrent :

```
/* Écouter l'événement "load". */
document.body.addEventListener("click",
        une.methode.bind(unParametre), false);

/* Supprimer l'écouteur, mais rien ne se passe. */
document.body.removeEventListener("click",
        une.methode.bind(unParametre), false);
```

Il faut à la place procéder de la manière suivante :

```
/* Obtenir une référence. */
var ref = une.methode.bind(unParametre);

/* Écouter l'événement "load". */
document.body.addEventListener("click", ref, false);

/* L'écouteur peut être trouvé et supprimé. */
document.body.removeEventListener("click", ref, false);
```

Dans cette version, les références manipulées dans les méthodes de l'écouteur correspondent à la même instance. La seconde permet de trouver la méthode employée dans la première. Ce point est évidemment important, notamment lorsque des écouteurs sont ajoutés et retirés en permanence pour mettre en place des animations (voir le chapitre précédent sur les transitions CSS). Bien entendu, tant que l'écouteur n'est pas supprimé, l'instance de l'objet est toujours référencée, l'écouteur est toujours en fonctionnement et l'occupation mémoire et la charge processeur augmentent en permanence, car une quantité de code toujours plus importante est exécutée pour un même événement.

Accéder aux propriétés et aux méthodes

Ce chapitre le montre, la syntaxe `objet.propriété` facilite l'accès aux propriétés. C'est également le cas pour les méthodes, car ce sont en réalité des propriétés qui font référence à des instances de l'objet `Function`. Toutes ces propriétés sont mémorisées sous forme de couple clé-valeur dans un tableau associatif. Elles sont donc accessibles à l'aide de la syntaxe `objet["propriété"]`, très utile notamment pour définir les caractéristiques d'un personnage dans un jeu de rôle. Cette approche est comparable à la méthode `setAttribute()` du DOM.

```
var Character = function(name) {
    this.patronym = name || "Inconnu";

    this.characteristics = {
        stamina: 10,
        mana: 10,
        skill: 10,
        health: 100
    };
}

Character.prototype.setCharacteristic = function(prop, value) {
    if (this.characteristics.hasOwnProperty(prop)) {
        if (value >= 0 && value <= 250) {
            this.characteristics[prop] = value;
        }
    }
}
```

```
Character.prototype.showCharacteristics = function() {
    console.log("Caractéristiques de " + this.patronym + " :");
    for (var prop in this.characteristics) {
        console.log("=>" + prop + " : " + this.characteristics[prop]);
    }
}

var paladin = new Character("Danis");
paladin.setCharacteristic("health", 200);
paladin.setCharacteristic("mana", 0);
paladin.showCharacteristics();

--- résultat ---
> Caractéristiques de Danis :
> => stamina : 10
> => mana : 0
> => skill : 10
> => health : 200
```

Pour éviter que l'objet des caractéristiques ne grandisse artificiellement, nous vérifions l'existence de la propriété concernée dans `setCharacteristic()`. Nous vérifions également que la valeur se trouve dans un intervalle raisonnable. Le contrôle de l'existence de la propriété évite également les erreurs liées à la casse des caractères, car les propriétés sont toujours sensibles à la casse.

Dans notre exemple, nous avons utilisé la syntaxe des tableaux associatifs pour parcourir les propriétés dans une instruction `for...in` et pour afficher leur valeur. Faites toutefois attention car si dans nos exemples les propriétés sont affichées dans leur ordre de déclaration, ce n'est pas nécessairement le cas. Lorsque vous parcourez les propriétés d'un objet, ne faites aucune hypothèse sur un ordre précis.

Définir des méthodes d'accès

Dans notre exemple précédent, nous avons utilisé une méthode pour fixer les valeurs de plusieurs propriétés. Cette solution est comparable à l'utilisation d'un mutateur dans d'autres langages orientés objet. JavaScript permet lui aussi de définir des accesseurs (*getter*) et des mutateurs (*setter*), bien qu'ils soient d'un intérêt limité car, contrairement à d'autres langages plus complexes, les propriétés d'un objet sont nécessairement publiques. Il n'existe aucun moyen natif pour définir des propriétés privées utilisables uniquement dans un objet précis. Par conséquent, alors que dans certains langages on essaiera de protéger l'accès à certaines variables de manière à faciliter la maintenance du code, il est fréquent en JavaScript de laisser les variables d'une instance d'objet accessibles au code client.

Lorsque l'on programme en JavaScript, il faut dissocier les éléments d'un objet existant afin de les manipuler comme des propriétés plutôt que de passer par une méthode. Les accesseurs apportent également un contrôle sur les accès et sur les valeurs affectables *via* une propriété. Par exemple, si nous souhaitons créer un objet qui compte le temps nécessaire à l'exécution

d'une fonction afin de comparer deux techniques, nous pouvons donner accès à la fonction de conversion des millisecondes en secondes au travers d'une propriété créée avec un accesseur :

```
/* Définition de l'objet. */
var Timing = function() {
    this.elapsed = 0;
}

Timing.prototype.test = function(iter, func) {
    var time = new Date();

    for (var i = 0; i < iter; i++) {
        func();
    }
    this.elapsed = new Date() - time;
}

Timing.prototype.__defineGetter__("seconds",
    function() { return this.elapsed / 1000; });

/* Utiliser l'objet. */
var timer = new Timing();

/* Tester la rapidité de Math.floor(). */
timer.test(1000000, function() { var a = Math.floor(10.6) });
console.log(timer.seconds);

/* Tester une version plus courte/rapide. */
timer.test(1000000, function() { var a = (10.6 << 0) });
console.log(timer.seconds);

--- résultat ---
> 0.035
> 0.011
```

À l'aide de la méthode `__defineGetter__`, nous créons une propriété en lecture seule qui nous permet d'obtenir le temps écoulé en secondes. Cette méthode prend en premier paramètre le nom de la propriété à créer, la fonction à appeler étant précisée en second paramètre. Toutefois, cela permet uniquement de lire la valeur. Pour être en mesure de la modifier, nous devons également définir un mutateur. La définition de cette fonction se fait de manière comparable à un accesseur, en passant uniquement une valeur à la fonction associée :

```
Timing.prototype.__defineSetter__("reset",
    function(value) {
        if (value === true) {
            this.elapsed = 0
        }
    });
```

Les accesseurs peuvent être ajoutés à un objet, comme dans notre exemple, ou dynamiquement à une instance. Nous pouvons également les utiliser directement dans des initialiseurs d'objets, à l'aide des mots clés `get` et `set` devant la propriété concernée :

```
var something = {
    prop: "valeur",
    get uneProp() { return this.prop },
    set uneProp(value) { this.prop = value }
};
```

Cette solution sera parfois utile, mais sachez que ces accesseurs seront plus lents que des méthodes qui procèdent aux mêmes opérations, à leur tour plus lentes que des propriétés normales. Ces accesseurs ne sont pas standard, mais ils sont pris en charge par Safari Mobile et de nombreux autres navigateurs. Ils risquent donc de se révéler plus utiles dans les très grandes applications JavaScript, où des contrôles stricts sur les propriétés garantissent une exécution correcte du code.

Isolation du code et bibliothèques

Le code doit souvent être isolé autant que possible des autres scripts d'une page ou d'un site web. L'objectif est d'éviter tout conflit avec du code tiers ou avec les bibliothèques fournies à d'autres applications web. Pour cela, nous pouvons avoir recours à la *chaîne de portée*. Elle définit l'environnement, lié au contexte d'exécution, dans lequel une fonction s'exécute. Autrement dit, il s'agit de la liste des objets analysés lors de l'évaluation d'une propriété. Chaque appel à une fonction ajoute de nouveaux éléments à la chaîne.

Isoler du code

L'isolation du code est plutôt simple à réaliser : il suffit d'entourer la fonction par des parenthèses de manière à forcer l'évaluation et à lancer ensuite l'exécution comme pour n'importe quelle référence de fonction :

```
(function() {
    /* Le code ici. */
})();
```

Tout ce qui est déclaré à l'intérieur de la fonction est placé dans la chaîne de portée et est inaccessible depuis les fonctions externes, car le contexte est lié à cette fonction. Tant que la définition d'une nouvelle variable se fait avec le mot clé `var`, il n'existe aucun risque d'écraser des propriétés externes. En l'absence de ce mot clé, une recherche est effectuée dans la chaîne de portée et elle risque de trouver une déclaration correspondante en dehors de la fonction :

```
var variable1 = "La variable externe 1.";
var variable2 = "La variable externe 2.";

function someFunc() {
    console.log("Appel de la fonction externe someFunc().");
}
```

```
(function() {
    var variable0 = "Inaccessible depuis le code externe.";
    var variable1 = "La variable interne 1.";
    variable2 = "Mauvais... variable externe 2 remplacée.";

    function someFunc() {
        console.log("Appel de la fonction interne someFunc().");
        console.log("Suis-je dans le contexte 'window' : " + (this == window));
    }

    /* Début des traces... */
    someFunc();
    window.someFunc();
})();

/* ...poursuite ici. */
console.log(variable0);
console.log(variable1);
console.log(variable2);
someFunc();

--- résultat ---
> Appel de la fonction interne someFunc().
> Suis-je dans le contexte 'window' : true
> Appel de la fonction externe someFunc().
x ReferenceError: Can't find variable: variable0
> La variable externe 1.
> Mauvais... variable externe 2 remplacée.
> Appel de la fonction externe someFunc().
```

Le code précédent définit deux variables externes et une fonction globale nommée `someFunc`. Vient ensuite du code qui est isolé et exécuté immédiatement. Notez que la seconde variable interne n'écrase pas son équivalent externe car elle est définie à l'aide du mot clé `var`. En revanche, une valeur est affectée à la dernière variable sans impliquer le mot clé `var`. Le moteur recherche donc une propriété de ce nom dans la chaîne de portée et, s'il en trouve une, lui affecte la valeur donnée. Enfin, `someFunc()` n'est pas écrasée car sa définition équivaut à la suivante :

```
var someFunc = function() { ... }
```

Elle est placée dans la chaîne de portée et découverte sur la pile dès l'appel à `someFunc()`. Bien entendu, il est possible d'appeler la méthode `someFunc()` définie auparavant en précisant l'objet auquel elle est associée, c'est-à-dire `window` dans cet exemple. En dehors d'une fonction, le mot clé `var` fonctionne différemment : il définit les variables et les fonctions sous forme de propriétés de l'objet global.