Éric Sarrion

Prototype et Scriptaculous

Dynamiser ses sites web avec JavaScript

Comptant parmi les cinq bibliothèques JavaScript les plus utilisées du développement web, le tandem de bibliothèques Prototype et Scriptaculous permet au développeur de créer des sites compatibles avec les différents types de navigateurs. Qu'il s'agisse de produire des effets visuels avancés, d'interroger le serveur par Ajax (sans réafficher une nouvelle page), de permettre la saisie automatique des champs (autocomplétion avec liste de valeurs proposées), ou d'autoriser le *drag and drop* d'éléments sur la page ou encore de proposer des curseurs, une ou deux lignes de code suffisent.

Cet ouvrage explique comment exploiter toutes les possibilités et la souplesse que ces deux bibliothèques offrent dans les développements de sites web, tant sous PHP que pour Ruby on Rails. Vos sites web ne seront plus les mêmes !

EYROLLES

Code éditeur : G85408
ISBN : 978-2-212-85408-4
© Eyrolles 2011

Éric Sarrion

Prototype et Scriptaculous

Dynamiser ses sites web avec JavaScript

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands-Augustins, 75006 Paris.
© Groupe Eyrolles, 2011, ISBN : 978-2-212-85408-4

Avant-propos

Prototype et Scriptaculous sont des bibliothèques JavaScript qui permettent de développer pour tous les navigateurs sans se soucier des différences liées aux spécificités des moteurs d'exécution. En fournissant une manière de développer uniforme et en enrichissant les possibilités des navigateurs, ces bibliothèques vont vous permettre d'améliorer l'expérience utilisateur qu'offrirait un site web traditionnel. Le but de ce livre est de vous faire découvrir toutes ces possibilités qui feront que vos sites web ne seront plus pareils qu'avant...

Pourquoi cet ouvrage ?

Avez-vous remarqué que très peu de documentation sur Prototype et Scriptaculous était disponible, aussi bien en français qu'en anglais ? De plus, les quelques livres écrits sur ce sujet datent déjà de 2007, alors que la dernière version de nos bibliothèques est sortie fin 2009 pour Scriptaculous, et début 2010 pour Prototype ! Autant dire que la documentation actuellement disponible, en dehors de ce livre, est obsolète.

Pourtant, tout montre que de nombreux développeurs les utilisent, car elles simplifient énormément la programmation JavaScript. Une documentation à jour, des exemples fonctionnels, et surtout, un guide d'apprentissage progressif sont des éléments nécessaires pour développer en exploitant 100 % des possibilités offertes par ces outils. Ce livre est, nous l'espérons, adapté à ce besoin et vous fera gagner un temps énorme !

Plan de l'ouvrage

Ce livre est découpé en deux parties : Prototype puis Scriptaculous. Nous commençons par l'étude de la bibliothèque Prototype car c'est elle qui est au cœur de Scriptaculous. Ainsi, Prototype nous aide plutôt à mieux utiliser le langage JavaScript, en nous procurant diverses classes, tandis que Scriptaculous est orienté vers une meilleure gestion de l'interface utilisateur. Ce livre respecte donc cette logique, et nous vous suggérons d'étudier ces parties dans cet ordre.

Le **chapitre 1** présente Prototype dans son ensemble, et vous montre ce que vous allez découvrir dans les chapitres le concernant.

Le **chapitre 2** explique le nouveau mécanisme offert par Prototype pour créer ses propres classes JavaScript. Les anciens mécanismes, internes au langage JavaScript, sont bien sûr toujours accessibles, mais les facilités offertes par Prototype vous les feront bien vite oublier !

Le **chapitre 3** vous présente certaines classes créées ou améliorées par Prototype. Vous découvrirez une nouvelle manière d'effectuer des boucles, et vous oublierez l'ancienne méthode avec `while` ou `for`.

Le **chapitre 4** vous fait découvrir qu'une fonction traditionnelle est aussi un objet de classe `Function`. Prototype a aussi amélioré ce mécanisme...

Le **chapitre 5** concerne la classe `Element`. Elle regroupe l'un des principaux usages de Prototype, permettant de manipuler facilement l'arborescence du document HTML.

Le **chapitre 6** traite les événements et leur gestion. Grâce à Prototype, vous gagnerez en efficacité et pourrez créer de nouveaux événements !

Le **chapitre 7** regroupe la description de quelques classes bien pratiques lorsqu'on veut observer les clics dans les formulaires de saisie. Prototype les a écrites pour nous, apprenons à les exploiter !

Le **chapitre 8** termine l'étude de Prototype en décrivant les possibilités offertes pour « faire de l'Ajax », c'est-à-dire pour lier notre programme JavaScript avec un programme sur un serveur.

Le **chapitre 9** concerne la bibliothèque Scriptaculous. Il décrit ses possibilités, qui sont détaillées dans les chapitres suivants.

Le **chapitre 10** traite les effets possibles par Scriptaculous. Un effet est un moyen d'animer les éléments de la page, uniquement avec quelques lignes de code JavaScript !

Le **chapitre 11** présente le mécanisme d'auto-complétion, qui permet de faciliter la saisie, en affichant une liste de suggestions correspondant aux lettres saisies dans le champ.

Le **chapitre 12** montre les mécanismes de drag & drop. Vous pouvez ainsi facilement déplacer des objets dans la page et les déposer ailleurs, en quelques lignes !

Le **chapitre 13** explique un des composants les moins connus de Scriptaculous, à savoir les sliders. Ils permettent de représenter visuellement des données, comme une distance, une date, etc.

Le **chapitre 14** termine notre étude par la possibilité d'inclure des sons dans nos pages. Un très court chapitre, mais avec beaucoup de potentiel pour nos pages !

À qui s'adresse cet ouvrage ?

Cet ouvrage est destiné à :

- Tous les développeurs, chefs de projet qui veulent exploiter les possibilités offertes par JavaScript pour réaliser des sites web innovants, car tous les concepts de programmation modernes (datant de 2007 et après) y sont utilisés pour concevoir des sites nouvelles générations.
- Tous les étudiants en école d'informatique en retireront également un grand bénéfice...

Remerciements

Sans la gentillesse de Sandrine Paniel et Muriel Shan Sei Fan, et l'expertise de Romain Pouclet, ce livre n'aurait sûrement pas la même qualité. Qu'ils soient remerciés ici.

Table des matières

PREMIÈRE PARTIE
Prototype 1

Installation de Prototype 2
Aperçu général de Prototype 2

CHAPITRE 1
Créer des classes 3

Créer des classes sans Prototype 3
Créer des classes avec Prototype 4
Héritage 5
Le mot-clé $super 8
Ajouter dynamiquement des méthodes à une classe 9
Les méthodes superclass et subclasses 11

CHAPITRE 2
Classes générales 13

Classe Object 13
Méthodes gérant les propriétés de l'objet 13
Méthodes booléennes 16
Classe Enumerable 17
Itération sur les éléments de la collection 17
Méthodes principales 18
Exemples d'utilisation des méthodes 20
Créer une classe qui hérite de Enumerable 22
Classe Array 23
Création d'un tableau 24
Méthodes principales 25
Exemples d'utilisation des méthodes 26
Classe Hash 26
Création d'une table de hachage 27

Méthodes principales 27
Exemples d'utilisation des méthodes 28
Classe ObjectRange 30
Création d'un intervalle 30
Méthodes principales 32
Classe Number 32
Classe String 33
Méthodes principales 33
Exemples d'utilisation des méthodes 34
Classe PeriodicalExecuter 35

CHAPITRE 3
Classe Function 39
Fonctionnement standard de la classe Function 39
Création d'un objet Function 39
Propriété this 40
Propriété prototype 41
Propriété arguments 42
Propriété caller 44
Méthodes de base 45
Méthodes définies par Prototype 46
Méthode bind (obj, arg1, ..., argN) 46
Méthode bindAsEventListener (obj, arg1, ..., argN) 48
Méthode wrap () 51

CHAPITRE 4
Classe Element 53
Gestion des éléments sans Prototype 53
Éléments et attributs 53
DOM (Document Object Model) 54
Création dynamique d'éléments HTML et texte 55
Méthodes d'insertion dans la page 56
Accès direct aux éléments de la page 57
Propriétés principales 59
Propriétés communes aux éléments HTML et texte. 59
Propriétés spécifiques aux éléments HTML. 59
Propriété spécifique aux éléments texte 59
Exemple : création dynamique dans la page 59
Gérer les éléments avec Prototype 61
Sélecteurs CSS 62
Sélecteurs simples 62

Sélecteurs d'attributs 62
Sélecteurs de classe 63
Sélecteurs d'id 64
Pseudo-classes 65
:link et :visited 65
:focus 65
:first-child 65
:last-child 66
:nth-child(an+b) 67
:only-child 68
:not 68
Combinateurs 68
Combinateur de descendance 68
Combinateur filial 69
Combinateur d'adjacence directe 69
Combinateur d'adjacence indirecte 69
Création d'un élément HTML 70
Éléments étendus 71
Formes d'écriture des méthodes de la classe Element 71
Accéder aux éléments de l'arborescence 72
Exemples d'utilisation 73
Mettre à jour le contenu d'un élément 76
Exemples d'utilisation 77
Modification de l'état d'un élément 79
Classes CSS de l'élément 80
Style de l'élément 80
Élément visible/caché 80
Opacité de l'élément 81
Attributs de l'élément 81
Autres caractéristiques 81
Exemples d'utilisation 82
Positionnement et dimensions sur l'écran 85
Exemples d'utilisation 86
Gestion des événements 89
Ajouter nos propres méthodes à la classe Element 89
Exemple d'utilisation 91
Classe Element.Layout 93
Propriétés définies dans Element.Layout.PROPERTIES 93
Méthodes de la classe Element.Layout 97
Méthodes de la classe Element 97

CHAPITRE 5
Gérer les événements 99
Gérer les événements sans Prototype 99
Classe Event 100
Définitions des codes des touches du clavier 101
Élément sur lequel survient l'événement 101
Coordonnées de l'événement 101
Gestion des boutons de la souris 102
Propagation de l'événement 102
Événements étendus 102
Observation de l'événement 103
Exemples d'utilisation 104
Déplacement d'un élément dans la page 104
Affichage d'un menu 106
Classe Event.Handler 108
Exemple d'utilisation 110
Classe Element 112
Exemple d'utilisation 112

CHAPITRE 6
Classes observateurs des formulaires 115
Classe Form.Element.Observer 115
Classe Form.Observer 117
Classe Form.Element.EventObserver 120
Classe Form.EventObserver 123

CHAPITRE 7
Classes Ajax 127
Mise en place d'un serveur 127
Mise en place d'un serveur PHP 128
Mise en place d'un serveur Ruby on Rails 129
Classe Ajax.Request 130
Options disponibles dans Ajax.Request 130
Code pour un serveur PHP 131
Code pour un serveur Ruby on Rails 133
Traiter la réponse du serveur (paramètre xml) 134
Récupérer et utiliser responseText 134
Récupérer responseXML 136
Utiliser responseXML 139
Utiliser responseText ou responseXML ? 141

Transmettre des paramètres au serveur 141
Code pour un serveur PHP 142
Code pour un serveur Ruby on Rails 143
Classe Ajax.Updater 145
Propriété options.insertion 145
Propriété options.evalScripts 148
Classe Ajax.PeriodicalUpdater 150
Exemples d'utilisation 151
Affichage de l'heure du serveur 151
Code pour un serveur PHP 151
Code pour un serveur Ruby on Rails 152
Utilisation de start () et stop () 153

DEUXIÈME PARTIE
Scriptaculous 157

Installation de Scriptaculous 158
Contenu de Scriptaculous 159

CHAPITRE 8
Effets visuels 161
Paramètres d'un effet visuel 161
Étapes d'exécution d'un effet visuel 163
Classe Effect.Base 168
Méthodes de gestion des effets visuels 169
Méthodes événementielles liées aux effets visuels 170
Classe Effect.Opacity 170
Classe Effect.Move 172
Classe Effect.Scale 175
Classe Effect.Highlight 177
Classe Effect.Morph 179
Classe Effect.Transform 183
Classe Effect.Parallel 185
Influence de l'option transition dans l'effet 190
Effets visuels standards 194
Effect.Fade 195
Effect.Appear 196
Effect.Puff 197
Effect.BlindUp 198
Effect.BlindDown 199

Effect.SwitchOff 200
Effect.DropOut 201
Effect.Shake 201
Effect.SlideDown 202
Effect.SlideUp 203
Effect.Squish 204
Effect.Grow 206
Effect.Shrink 207
Effect.Pulsate 208
Effect.Fold 209
Méthodes ajoutées à la classe Element 210
Création d'un nouvel effet visuel 211
Création d'une classe d'effet visuel 211
Création d'une fonction d'effet visuel 214
Ajout de l'effet dans les méthodes de la classe Element 216
Ajout d'un effet sous forme de classe d'effet 216
Ajout d'un effet sous forme de fonction d'effet 217
Effet multiple 217

CHAPITRE 9
Auto-complétion 219
Classe Autocompleter.Base 219
Fonctionnement de base 220
Propriétés de la classe Autocompleter.Base 220
Options de baseInitialize () 221
Exemple d'utilisation 223
Classe Ajax.Autocompleter 228
Fonctionnement de base 228
Propriétés de la classe Ajax.Autocompleter 228
Options de la classe Ajax.Autocompleter 229
Exemple d'utilisation 229
Code pour un serveur PHP 230
Code pour un serveur Ruby on Rails 233
Classe Autocompleter.Local 236
Fonctionnement de base 236
Options de la classe Autocompleter.Local 237
Exemple d'utilisation 238
Classe Ajax.InPlaceEditor 241
Principes du fonctionnement 242
Propriétés de la classe Ajax.InPlaceEditor 242
Méthode de la classe Ajax.InPlaceEditor 242

Les options de new 244
Le pointeur de la souris touche l'élément 245
Le pointeur de la souris sort de l'élément 245
L'utilisateur clique sur l'élément 245
On valide la saisie 248
On annule la saisie 249
Récapitulatif des options 250
Exemple d'utilisation 252
Classe Ajax.InPlaceCollectionEditor 255
Fonctionnement de base 255
Utilisation des options 255
Correction d'un bogue sous Internet Explorer 256
Exemple d'utilisation 257

CHAPITRE 10
Drag & drop 261

Classe Draggable 261
Fonctionnement de base 261
Propriétés de la classe 262
Méthode de la classe 262
Les options disponibles 264
On clique sur l'élément 264
On déplace la souris 264
On relâche la souris 267
On appuie sur Échap 268
Récapitulatif des options 268
Exemple d'utilisation 270
Objet Droppables 273
Fonctionnement de base 273
Méthodes de l'objet Droppables 274
Propriétés de l'objet Droppables 274
Les options employées lors de l'appel de Droppables.add 274
Étude de Droppables.show (pointer, élément_déplacé) 275
Étude de Droppables.fire (event, élément_déplacé) 277
Récapitulatif des options 277
Exemple d'utilisation 278
Amélioration du programme 283
Objet Sortable 284
Fonctionnement de base 285
Méthodes de l'objet Sortable 286
Utilisation des options 286

Étude de Sortable.create (element, options) 287
Actions effectuées lors du survol d'un élément autorisé 290
Actions effectuées lors du relâchement de la souris 291
Récapitulatif des options 291
Exemples d'utilisation 293
Tri dans une seule liste 293
Tri entre plusieurs listes 297

CHAPITRE 11
Sliders **301**
Fonctionnement de base 301
Constructeur de la classe Control.Slider 306
Méthodes de la classe Control.Slider 307
Propriétés de la classe Control.Slider 307
Utilisation des options 308
Cliquer sur un curseur 308
Cliquer sur l'axe 308
Obtention de la valeur d'un curseur 308
Appel de options.onChange (valeurs, objSlider) 309
Déplacer un curseur (directement ou sur l'axe) 309
Obtention de la valeur d'un curseur 309
Appel de options.onSlide (valeurs, objSlider). 309
Arrêter le déplacement d'un curseur 309
Appel de options.onChange (valeurs, objSlider) 310
Utiliser la méthode setValue (valeur, indice_curseur) 310
Obtention de la valeur d'un curseur 310
Appel de options.onChange (valeurs, objSlider) 310
Récapitulatif des options 310
Exemples d'utilisation 312
Afficher la valeur d'un curseur 312
Utiliser plusieurs curseurs 314
Régler l'opacité d'une image à l'aide d'un curseur 316

CHAPITRE 12
Sons **319**
Inclure et lire des fichiers son 319
Activer et désactiver la lecture 320

Conclusion **321**

Index **323**

Première partie

Prototype

La bibliothèque Prototype est le cœur des bibliothèques Prototype et Scriptaculous. Elle offre une nouvelle façon de programmer en JavaScript, plus simple et plus agréable que la façon traditionnelle.

Installation de Prototype

La bibliothèque Prototype est disponible sur http://www.prototypejs.org. Elle est constituée d'un fichier `prototype.js`, que l'on peut inclure dans la page HTML de la façon suivante.

Inclusion du fichier prototype.js dans la page HTML

```
<html>
<head>
<script src = "prototype.js"></script>
</head>

<body>

</body>
</html>
```

Quelle version de Prototype ?

La version utilisée dans les pages qui suivent correspond à la plus récente actuellement (version 1.7 RC2).

SYNTAXE **Balise <script>**

Vous remarquerez que nous n'indiquons pas l'attribut type dans la balise `<script>`. En effet, cet attribut n'est pas obligatoire pour que le navigateur comprenne le code HTML que nous lui indiquons. Dans un souci de simplification, nous ne l'indiquerons pas dans cet ouvrage.

Aperçu général de Prototype

Une fois la bibliothèque Prototype incluse dans la page HTML, vous avez accès aux nouvelles fonctionnalités suivantes :

- enrichissement des classes natives de JavaScript, telles que `Object`, `Array`, `Number`, `String`, `Function` ;
- ajout de nouvelles classes : `Enumerable`, `Hash`, `ObjectRange` ;
- accès au DOM *(Document Object Model)* facilité à travers la classe `Element`, et aux événements JavaScript grâce à la classe `Event` ;
- mise en place d'une observation des champs de formulaires à l'aide des classes observateurs créées par Prototype ;
- simplification de l'utilisation d'Ajax *(Asynchronous JavaScript And XML)* grâce aux classes spécialisées.

L'ensemble des classes et enrichissements apportés par Prototype sont étudiés dans les chapitres suivants.

1

Créer des classes

JavaScript permet de définir ses propres classes, mais pour réaliser ceci la syntaxe est peu pratique. Prototype a créé un nouveau mécanisme simplifiant la mise en œuvre, ajoutant également la possibilité de créer des classes dérivées à partir de classes mères.

Afin de bien montrer les différentes façons de procéder avec ou sans Prototype, nous étudions ces deux possibilités ci-après.

Créer des classes sans Prototype

Une classe JavaScript est en fait une fonction classique, dont les objets s'instancient par `new nom_de_la_fonction (liste de paramètres)`. Voici un exemple :

Création d'une classe Rectangle et utilisation

```
<script>
// déclaration de la classe Rectangle
function Rectangle (height, width)
{
  this.height = height;
  this.width = width;
  this.surface = function ()
  {
    return this.width * this.height;
```

```
  }
}

// utilisation de la classe Rectangle : création d'un objet
var r = new Rectangle (2, 3);
alert ("Rectangle de hauteur : " + r.height +     // 2
       ", de largeur : " + r.width +              // 3
       ", et de surface : " + r.surface ());      // 6
</script>
```

Nous avons défini une classe `Rectangle`, possédant les propriétés `height` et `width` pour chacun des objets qui seront créés pour cette classe. On a aussi défini la méthode `surface ()`, qui retourne la surface du rectangle. Cela correspond à l'utilisation du mot-clé `this` à l'intérieur de la fonction.

L'inconvénient de cette syntaxe est le suivant : bien que fonctionnelle, elle ne permet pas de distinguer aisément une définition de fonction classique d'une définition d'une classe. Bien sûr l'utilisation du mot-clé `this` dans le code de la fonction peut nous alerter sur l'utilisation qui sera faite de celle-ci, mais nous aimerions finalement que le mot-clé `function` ne soit pas utilisé pour définir une classe, mais plutôt le mot-clé `class` qui est plus significatif.

Créer des classes avec Prototype

Comme le mot-clé `class` n'existe pas dans le langage JavaScript, Prototype a contourné le problème en créant l'objet `Class`, auquel il a ajouté la méthode `create ()` permettant de créer une classe. L'exemple précédent peut alors s'écrire :

Une classe créée avec Prototype

```
<script src = prototype.js> </script>
<script>
// déclaration de la classe Rectangle
var Rectangle = Class.create ( {
  initialize : function (height, width) // constructeur
  {
    this.height = height;
    this.width = width;
  },
  surface : function ()
  {
    return this.width * this.height;
  }
});
```

```
// utilisation de la classe Rectangle : création d'un objet
// -> identique avec ou sans Prototype
var r = new Rectangle (2, 3);
alert ("Rectangle de hauteur : " + r.height +
       ", de largeur : " + r.width +
       ", et de surface : " + r.surface ());
</script>
```

`Class.create ()` permet de créer un objet de classe `Function` (cette classe est définie dans JavaScript), que l'on peut ensuite instancier par `new`. L'utilisation qui est faite de la fonction retournée par `Class.create ()` est donc similaire à celle que l'on effectuait sans Prototype.

Le paramètre `obj` utilisé dans `Class.create (obj)` est un objet définissant les méthodes et propriétés de la classe. Il peut posséder une méthode constructeur des objets de la classe (facultatif), dont le nom est alors obligatoirement `initialize ()`. Les autres méthodes de la classe sont définies à la suite, ici selon la syntaxe JSON (*JavaScript Object Notation*).

Héritage

L'héritage est un concept orienté objet permettant à une classe d'hériter des méthodes et propriétés définies dans une classe dite parent (ou mère). Par exemple on pourrait définir la classe `Carre` qui est en fait un cas particulier de `Rectangle`. Grâce à l'héritage, nous n'aurons pas à redéfinir la méthode `surface ()` dans la classe `Carre`, qui sera héritée de celle définie dans `Rectangle`.

La syntaxe de création d'une classe dérivée se fait par la méthode `Class.create ()`.

```
var Carre = Class.create (Rectangle, {
  // nouvelles méthodes sous forme JSON
  methode1 : function (...)
  {
  },

  methode2 : function (...)
  {
  }
});
```

Dans ce cas, le premier argument de `Class.create ()` est le nom de la classe parent, tandis que le deuxième est un objet contenant la définition des nouvelles méthodes

de la classe créée. Selon le principe de la programmation orientée objet, les anciennes méthodes de la classe parent sont accessibles dans la nouvelle classe (sinon cela n'aurait aucun intérêt !).

Créons la classe `Carre` dérivant de `Rectangle`, qui possédera en plus la méthode `perimetre ()` :

```
<html>
<script src = prototype.js> </script>
<script>
var Rectangle = Class.create ( {
  initialize : function (height, width)
  {
    this.height = height;
    this.width = width;
  },
  surface : function ()
  {
    return this.width * this.height;
  }
});

var Carre = Class.create (Rectangle, {
  perimetre : function ()
  {
    return 4 * this.height;
  }
});

var c = new Carre (2, 2);
alert ("Carre de hauteur : " + c.height +
       ", de largeur : " + c.width +
       ", de périmètre : " + c.perimetre () +
       ", et de surface : " + c.surface ());
</script>
</html>
```

La classe `Carre`, héritant de `Rectangle`, a accès aux méthodes de `Rectangle` mais aussi aux nouvelles méthodes qu'elle a créées.

Intéressons-nous à la création d'un objet `Carre` :

```
var c = new Carre (2, 2);
```

Reconnaissons que cette façon de procéder n'est pas très bonne, car nous devons indiquer la largeur et la hauteur du carré, alors que c'est bien entendu la même valeur ! Il serait beaucoup plus pertinent d'écrire :

```
var c = new Carre (2);      // largeur d'un seul côté !
```

Cela nécessite de réécrire le code de la méthode `initialize ()` dans la classe `Carre`, sinon c'est la méthode `initialize (height, width)` définie dans `Rectangle` qui sera utilisée. On écrira donc :

Classe Carre avec constructeur spécifique

```
<script src = prototype.js> </script>
<script>
var Rectangle = Class.create ( {
  initialize : function (height, width)
  {
    this.height = height;
    this.width = width;
  },
  surface : function ()
  {
    return this.width * this.height;
  }
});

var Carre = Class.create (Rectangle, {
  initialize : function (size)   // création d'un Carre
  {
    this.height = size;
    this.width = size;
  },
  perimetre : function ()
  {
    return 4 * this.height;
  }
});

var c = new Carre (2);
alert ("Carre de hauteur : " + c.height +
       ", de largeur : " + c.width +
       ", de périmètre : " + c.perimetre () +
       ", et de surface : " + c.surface ());
</script>
```

C'est mieux, mais pas encore parfait ! En effet, pourquoi, dans le constructeur de `Carre`, devrions-nous réécrire ce qui avait été précédemment écrit pour `Rectangle` ? Ici le code est simple donc ce n'est pas très compliqué à réécrire, mais dans le cas d'un

code plus complexe, cela oblige à dupliquer les mêmes parties de code à différents endroits. D'où l'utilisation du mot-clé `$super` ci-dessous.

Le mot-clé $super

`$super` est une variable créée par Prototype qui est utilisée comme premier paramètre dans une méthode définie dans une classe, et correspond à la méthode de même nom de la classe parente. Par exemple, si nous écrivons :

```
var Carre = Class.create (Rectangle, {
  initialize : function ($super, size)
  {
    $super (size, size); // construire à partir de Rectangle
  }
});
```

`$super` désigne ici une référence à la fonction `initialize ()` définie dans la classe `Rectangle.` On peut donc s'en servir pour construire un objet `Carre`, en utilisant le constructeur défini dans `Rectangle`.

Nous avons utilisé `$super` ici dans la méthode `initialize ()`, mais il peut s'utiliser pour toute autre méthode redéfinie dans une classe héritée. L'exemple ci-après définit une nouvelle méthode `surface (scale)`, qui utilise la méthode `surface ()` définie dans `Rectangle`, mais multiplie le résultat par `scale.`

```
<script src = prototype.js> </script>
<script>
var Rectangle = Class.create ( {
  initialize : function (height, width)
  {
    this.height = height;
    this.width = width;
  },
  surface : function ()
  {
    return this.width * this.height;
  }
});

var Carre = Class.create (Rectangle, {
  initialize : function ($super, size)
  {
    $super (size, size);
  },
```

```
  perimetre : function ()
  {
    return 4 * this.height;
  },
  surface : function ($super, scale)
  {
    return $super () * scale;
  }
});

var c = new Carre (2);
alert ("Carre de hauteur : " + c.height +
   ", de largeur : " + c.width +
   ", de périmètre : " + c.perimetre () +
   ", et de surface : " + c.surface (10));   // 40 au lieu de 4 !
</script>
```

Ajouter dynamiquement des méthodes à une classe

La description des classes `Rectangle` et `Carre` que nous avons précédemment créées était figée : les méthodes étaient définies dans ces classes lors de leur création. Prototype permet d'ajouter dynamiquement de nouvelles méthodes à une classe déjà définie. On utilise alors `NomClasse.addMethods ()`, `NomClasse` étant le nom de la classe pour laquelle on ajoute les méthodes :

```
Rectangle.addMethods ({
  // nouvelles méthodes sous forme JSON
  methode1 : function (...)
  {
  },
  methode2 : function (...)
  {
  }
});
```

Ajoutons dynamiquement la méthode `perimetre ()` dans la classe `Rectangle` :

```
<script src = prototype.js> </script>
<script>
var Rectangle = Class.create ( {
  initialize : function (height, width)
  {
    this.height = height;
    this.width = width;
  },
```

```
  surface : function ()
  {
    return this.width * this.height;
  }
});

// ajout dynamique de la méthode perimetre ()
// à la classe Rectangle
Rectangle.addMethods ({
  perimetre : function ()
  {
    return (this.height + this.width) * 2;
  }
});

var r = new Rectangle (2, 3);
alert ("Rectangle de hauteur : " + r.height +
       ", de largeur : " + r.width +
       ", de périmètre : " + r.perimetre () +     // 10
       ", et de surface : " + r.surface ());

</script>
```

Cette fonctionnalité est très puissante, car elle permet de plus d'ajouter les méthodes non seulement à la classe, mais à toutes les instances d'objets de cette classe qui ont pu être créées avant. On pourrait donc dans notre exemple précédent créer l'objet `r` avant l'ajout de la méthode dans la classe `Rectangle` : bien que l'objet ait été créé selon l'ancien modèle de la classe, il bénéficiera tout de même des nouvelles méthodes ajoutées. Il sera donc également possible d'écrire :

```
// d'abord on crée l'objet
var r = new Rectangle (2, 3);

// puis ajout dynamique de la méthode perimetre ()
// à la classe Rectangle
Rectangle.addMethods ({
  perimetre : function ()
  {
    return (this.height + this.width) * 2;
  }
});

alert ("Rectangle de hauteur : " + r.height +
       ", de largeur : " + r.width +
       ", de périmètre : " + r.perimetre () +     // 10
       ", et de surface : " + r.surface ());
```

Les méthodes superclass et subclasses

Prototype ajoute pour chaque classe que nous créons les méthodes `superclass` et `subclasses` :

- `superclass` permet de récupérer la classe parente. On récupère donc la variable qui a servi à définir la classe (par exemple `Rectangle` si la classe a été définie par `var Rectangle = Class.create ()`).
- `subclasses` contient un tableau de toutes les classes qui ont dérivées de celle-ci. On récupère donc, sous forme de tableau, les variables qui ont servi à définir les classes dérivées.

Dans cet exemple, nous définissons la classe `Rectangle`, puis les classes `Carre` et `Carre2` qui héritent de `Rectangle`.

Utilisation de superclass et subclasses

```
<script src = prototype.js> </script>
<script>
var Rectangle = Class.create ( {
  initialize : function (height, width)
  {
    this.height = height;
    this.width = width;
  },
  surface : function ()
  {
    return this.width * this.height;
  }
});

var Carre = Class.create (Rectangle, { });
var Carre2 = Class.create (Rectangle, { });

alert (Rectangle.subclasses[0] == Carre);     // true
alert (Rectangle.subclasses[1] == Carre2);    // true
alert (Carre.superclass == Rectangle);        // true
alert (Carre2.superclass == Rectangle);       // true

</script>
```

Les améliorations apportées par Prototype pour la création de classes d'objets rendent obsolète l'utilisation des anciens mécanismes de base JavaScript.

2

Classes générales

Prototype facilitant la création de classes, comme nous l'avons vu au chapitre précédent, ses concepteurs en ont profité pour créer de nombreuses classes utilitaires et pour améliorer des classes natives de JavaScript.

Classe Object

La classe `Object` est une classe standard du langage JavaScript. Elle est la classe parent de toutes les autres, car c'est la classe de base permettant de créer les objets JavaScript.

La classe `Object` de JavaScript a été enrichie au moyen de méthodes statiques. Chacune de ces méthodes possède en premier paramètre l'objet sur lequel elle s'applique.

Méthodes gérant les propriétés de l'objet

Ces méthodes ont une action (de lecture ou d'écriture) sur les propriétés définies dans l'objet. Par exemple, récupérer l'ensemble des noms de propriétés, leurs valeurs, etc. Pour expliquer ces méthodes, nous utilisons ici un objet `obj`.

Définition d'un objet

```
var obj = {
  nom : "Sarrion",
  prenom : "Eric",
  age : function ()
  {
    var year = new Date ().getFullYear ();   // année YYYY
    return year - 1961;
  }
};
```

Nous avons défini dans cet objet le nom, le prénom et une fonction calculant l'âge à partir de l'année de naissance.

Remarquons que nous avons utilisé la notation JSON (*JavaScript Object Notation*), et que nous pouvons également l'écrire comme suit :

Définition d'un objet (sans utiliser la notation JSON)

```
var obj = new Object ();
obj.nom = "Sarrion";
obj.prenom = "Eric";
obj.age = function ()
{
  var year = new Date ().getFullYear ();   // année YYYY
  return year - 1961;
};
```

Utilisons maintenant les méthodes offertes par Prototype.

Récupérer les noms des propriétés/méthodes d'un objet

```
alert (Object.keys (obj));
                // "nom", "prenom", "age"
```

Récupérer les valeurs des propriétés/méthodes d'un objet

```
alert (Object.values (obj));
                // "Sarrion", "Eric", function () { ... }
```

Transformer sous forme d'une chaîne JSON

```
alert (Object.toJSON (obj));
                // {"nom": "Sarrion", "prenom": "Eric"}
```

Lors de cette transformation, les propriétés associées à des fonctions ne sont pas récupérées (ici la propriété `age` n'a pas été récupérée). Si une valeur de propriété est elle-même un objet, celui-ci est inséré dans la chaîne après transformation en JSON.

Transformer sous forme d'une chaîne sérialisée

```
alert (Object.toQueryString (obj));
                // nom=Sarrion&prenom=Eric&age=function%20()...
```

L'objet est sérialisé sous la forme « propriété=valeur », chaque bloc étant séparé du suivant par `&`. Cette méthode est utile lorsque l'on désire transmettre des informations sérialisées, par exemple par Ajax vers un serveur.

Les données sont sérialisées et encodées en UTF-8 (les espaces sont par exemple remplacés par `%20`). Les définitions des méthodes sont aussi sérialisées, cependant les propriétés dont la valeur est elle-même un objet ne sont pas prises en compte.

Dupliquer un objet

```
alert (Object.keys (Object.clone (obj)));
                // "nom", "prenom", "age"
alert (Object.values (Object.clone (obj)));
                // "Sarrion", "Eric", function () { ... }
```

Ajouter des propriétés à un objet

```
var obj1 = { adresse : { rue : "Champs Elysées",
                         ville : "Paris" } };
Object.extend (obj, obj1);
alert (Object.keys (obj));
     // "nom", "prenom", "adresse"
alert (Object.toJSON (obj));
     // {"nom": "Sarrion", "prenom": "Eric",
     // "adresse": {"rue": "Champs Elysées", "ville": "Paris"}}
```

Lors de `Object.extend (firstObj, secondObj)`, les propriétés et valeurs du deuxième objet `secondObj` sont ajoutées au premier objet `firstObj`.

```
var obj1 = { adresse : { rue : "Champs Elysées",
                         ville : "Paris" } };
Object.extend (obj, obj1);

obj.adresse.ville = "Versailles";
alert (obj1.adresse.ville);     // "Versailles" !
```

> **ATTENTION Par référence ou par valeur ?**
>
> Les valeurs des propriétés ajoutées le sont par référence et non par valeur. Ceci implique que si une valeur de propriété est un objet (en fait une référence vers une zone mémoire), c'est cette adresse qui sera ajoutée dans le premier objet : l'adresse pointera la même zone mémoire dans les deux objets. Donc toute modification de valeur d'un élément de cette zone sera répercutée pour les deux objets.

Méthodes booléennes

Ces méthodes sont utilitaires, elles permettent de savoir si l'objet est d'une classe donnée. Elles retournent `true` ou `false` selon que la condition est vraie ou fausse.

L'objet est-il un élément DOM ?

```
var obj = document.createElement ("div");
alert (Object.isElement (obj));    // true
```

L'objet est-il un tableau ?

```
var obj = [1, 2, 3];
alert (Object.isArray (obj));    // true
```

L'objet est-il une table de hachage ?

```
var obj = new Hash ({ nom : "Sarrion", prenom : "Eric" });
alert (Object.isHash (obj));    // true
```

L'objet est-il une fonction ?

```
var obj = function (a) { return a * a };
alert (Object.isFunction (obj)); // true
```

L'objet est-il une chaîne de caractères ?

```
var obj = new String ("Eric");
alert (Object.isString (obj));    // true
var obj = "Eric";
alert (Object.isString (obj));    // true
```

L'objet est-il une valeur numérique ?

```
var obj = 3.14;
alert (Object.isNumber (obj));    // true
```

Classe Enumerable

La classe `Enumerable` est l'une des plus fondamentales de Prototype. Elle regroupe sous forme de méthodes des moyens d'accès aux données d'une collection. L'intérêt est que la collection n'est pas d'un type particulier figé, mais peut être n'importe quelle collection d'objets (un tableau, une table de hachage, un ensemble, etc.).

En fait, la classe `Enumerable` est ce qu'on appelle un *mixin*. Elle ne s'utilise pas telle quelle (c'est-à-dire qu'on ne crée pas d'objets de classe `Enumerable`), mais elle s'intègre dans d'autres classes en leur permettant d'hériter de ses méthodes, sans avoir à les réécrire. Les classes `Array` et `Hash` que nous décrivons à la suite héritent de la classe `Enumerable`, et bénéficient donc des méthodes décrites ci-dessous.

Itération sur les éléments de la collection

Le parcours classique d'une collection d'objets (ici un tableau) se fait au moyen d'une boucle `for ()` ou `while ()`. Cette méthode devient obsolète lorsque la collection hérite de `Enumerable` (mais elle reste néanmoins accessible).

Ancienne méthode de parcours d'un tableau

```
var tab = ["Eric", "Jacques", "Pierre", "Edmond"];
for (var i = 0; i < tab.length; i++)
  alert (tab[i]);   // "Eric", "Jacques", "Pierre", "Edmond"
```

Nouvelle méthode de parcours d'un tableau

```
var tab = ["Eric", "Jacques", "Pierre", "Edmond"];
tab.each (function (e)   // e : élément du tableau
{
  // on dispose de l'élément du tableau,
  // à nous d'en faire l'usage désiré !
  alert (e); // "Eric", "Jacques", "Pierre", "Edmond"
});
```

La méthode `each ()` appliquée sur le tableau prend en paramètre une fonction de callback qui jouera le rôle d'un itérateur. Cette fonction prend comme premier paramètre l'élément courant du tableau (ici la variable `e`, qui vaudra successivement chacun des éléments du tableau). L'avantage de cette façon de procéder est que nous n'utilisons plus l'indice `i` pour parcourir la boucle, ni la condition de fin de boucle basée sur le nombre d'éléments contenus dans la collection récupérée par `tab.length`.

Si vous désirez connaître l'indice `i` de l'élément `e` dans la collection, Prototype le transmet en deuxième argument de la fonction.

Accès à l'élément et à son indice

```
var tab = ["Eric", "Jacques", "Pierre", "Edmond"];
tab.each (function (e, i)   // e : élément du tableau
{                           // i : indice de l'élément
  // on dispose de l'élément du tableau et son indice
  // à nous d'en faire l'usage désiré !
  alert (e); // "Eric", "Jacques", "Pierre", "Edmond"
  alert (i); // 0, 1, 2, 3
});
```

Méthodes principales

La méthode `each ()` précédente parcourt toute la collection, élément après élément. Il pourrait également être utile d'avoir une méthode qui fournisse le plus grand élément du tableau, ou une autre méthode qui fasse un tri en ne retournant que les éléments pairs du tableau. Tout ceci (et bien plus) est possible grâce aux méthodes fournies par `Enumerable`.

- `all (function (e, i) { ... })` : parcourt la collection en vérifiant que la fonction retourne `true` pour tous les éléments de la collection. Le retour de `all ()` est `true` dans ce cas, sinon `false`.
- `every (function (e, i) { ... })` : est synonyme de `all ()`.
- `any (function (e, i) { ... })` : parcourt la collection en vérifiant que la fonction retourne `true` pour au moins un des éléments de la collection. Le retour de `any ()` est `true` dans ce cas, sinon `false`.
- `some (function (e, i) { ... })` : est synonyme de `any ()`.
- `collect (function (e, i) { ... })` : parcourt la collection en mettant dans un tableau chaque retour de la fonction. Si la fonction ne retourne rien pour un élément de la collection, `undefined` est mis dans le tableau. Le tableau (contenant donc autant d'éléments que la collection) est ensuite retourné par `collect ()`.
- `map (function (e, i) { ... })` : est synonyme de `collect ()`.
- `detect (function (e, i) { ... })` : parcourt la collection et retourne le premier élément pour lequel la fonction retourne `true`, sinon retourne `undefined` (pour indiquer qu'aucun élément ne satisfait la condition).
- `find (function (e, i) { ... })` : est synonyme de `detect()`.
- `inject (init, function (acc, e, i) { ... })` : parcourt la collection en transmettant à la fonction un objet `acc` (pour accumulateur) dont la valeur initiale est `init`. La valeur retournée par chaque appel de la fonction correspond à la

valeur de `acc` qui sera transmise au prochain appel. La méthode `inject ()` retourne finalement la dernière valeur de `acc`.

- `each (function (e, i) { ... })` : parcourt la collection. La fonction permet de faire un traitement pour chaque élément transmis en argument.
- `findAll (function (e, i) { ... })` : parcourt la collection et retourne dans un tableau tous les éléments pour lesquels la fonction retourne `true`. Le tableau retourné peut être vide (si la fonction ne retourne `true` pour aucun élément).
- `select (function (e, i) { ... })` : est synonyme de `findAll()`.
- `filter (function (e, i) { ... })` : est synonyme de `findAll()`.
- `grep (regexp, function (e, i) { ... })` : parcourt la collection en transmettant à la fonction tous les éléments qui satisfont l'expression régulière `regexp`. Les valeurs retournées par la fonction sont mises dans un tableau qui est retourné par `grep ()`. Si la fonction ne retourne rien pour un élément transmis à la fonction, `undefined` est mis dans le tableau.
- `min (function (e, i) { ... })` : parcourt la collection et retourne la plus petite valeur retournée par la fonction.
- `max (function (e, i) { ... })` : parcourt la collection et retourne la plus grande valeur retournée par la fonction.
- `partition (function (e, i) { ... })` : parcourt la collection et la divise en deux tableaux retournés par `partition ()`. Le premier tableau contient tous les éléments pour lesquels la fonction retourne `true`, tandis que le second tableau contient tous les éléments pour lesquels la fonction retourne `false`.
- `reject (function (e, i) { ... })` : parcourt la collection et retourne dans un tableau tous les éléments excepté ceux pour lesquels la fonction retourne `true`. C'est le résultat inverse de `findAll ()`.
- `sortBy (function (e, i) { ... })` : parcourt la collection et retourne un tableau trié selon l'entier (positif ou négatif) indiqué en retour de la fonction. On peut ainsi ordonner la collection selon un certain ordre.
- `include (obj)` : `true` si l'objet appartient à la collection, sinon `false`.
- `member (function (e, i) { ... })` : est synonyme de `include ()`.
- `toArray ()` : transforme la collection en objet `Array`.
- `entries (function (e, i) { ... })` : est synonyme de `toArray ()`.
- `invoke ("methodName", arg1, ..., argN)` : parcourt la collection et appelle pour chaque élément `e` de la collection la méthode `e.methodName (arg1, ..., argN)`. Le paramètre `methodName` est une chaîne de caractères, les arguments `arg1`, `argN` sont facultatifs. Le retour de `invoke ()` est un tableau contenant le retour de chaque appel de `e.methodName (arg1, ..., argN)`.