

Programmation Web Côté Client avec JavaScript et jQuery

Table des matières

1	Premiers pas en <i>JavaScript</i>	5
1.1	Balise <code><script></code> et Hello world en <i>JavaScript</i>	5
1.2	Types, variables et portée	6
1.3	Fonctions	6
1.4	Objets	7
1.5	Tableaux (le type <code>Array</code>)	11
1.6	Exemple : traitement d'un formulaire avec <i>jQuery</i>	13
2	Programmation Fonctionnelle et Objet en <i>JavaScript</i>	16
2.1	Le <i>Pattern</i> Module	16
2.2	Passage d'arguments par objets	19
2.3	Exemple de fabrique sommaire	20
2.4	Structuration d'une application	21
2.5	Exemple : un module <code>metier.regexUtil</code>	23
2.6	Module Métier <code>adresse</code>	27
2.7	Création d'un Module <code>myApp.view.adresse</code>	37
3	Constructeurs, Prototype et <i>Patterns</i> Associés	41
3.1	Constructeurs	41
3.2	Prototypes	42
3.3	Exemple : assurer l'implémentation d'interfaces	45
3.4	Fabrique d'Objets Métier avec prototype	47
3.5	<i>Patterns pseudo-classique</i> (à éviter)	55
4	Interfaces Hommes Machines (<i>IHM</i>)	58
4.1	Filtrage Basique des Inputs d'un Formulaire	58
4.2	<i>Pattern Mediator</i> pour le filtrage d'attributs	60
4.3	Exemple : génération automatique de formulaire d'adresse	64
5	Exemple d'Application Interactive	71
5.1	Principe de l'application et analyse fonctionnelle	71
5.2	Modèle de donnée	71
5.3	<i>Pattern Mediator</i> : centraliser les événements	74
5.4	Événements concernant les personnes	77
5.5	Événements concernant les Adresses	90

6	Requêtes Asynchrones et <i>API Restful</i>	99
6.1	Qu'est-ce qu'une requête asynchrone ?	99
6.2	Requête <i>Ajax</i>	100
6.3	Qu'est-ce qu'une <i>API REST</i> (ou systèmes <i>Restful</i>) ?	103
6.4	Exemple d' <i>API Restful</i>	103
6.5	Persistance avec <i>AJAX</i>	114
A	Graphisme avec les Canvas <i>HTML5</i>	125
A.1	Notion de <i>canvas</i>	125
A.2	Exemple d'animation dans un <i>canvas</i>	126
B	Programmation Événementielle en <i>JavaScript</i>	128
B.1	Rappel sur la Gestion d'Événements en <i>CSS</i>	128
B.2	Événements en <i>Javascript</i>	129
C	Gestion des fenêtres	134
C.1	Charger un nouveau document	134
C.2	Naviguer dans l'historique	135
C.3	Ouvrir une nouvelle fenêtre (popup)	136
D	<i>Document Object Model (DOM)</i>	137
D.1	Qu'est-ce que le <i>DOM</i> ?	137
D.2	Sélection et Manipulation de Base sur le <i>DOM</i>	138

Architectures client/serveur et *API*

Architecture d'une application multi plate-formes

Une application multi plate-formes contemporaine cherchera à se structurer suivant (voir figure 1) :

1. Une application sur un serveur (*API*) qui traitera les données et assurera la persistance ;
2. Une application sur chaque type de client, qui utilise ce serveur via des requêtes, et gère l'interface (par exemple une *Interface Homme Machine (IHM)*).

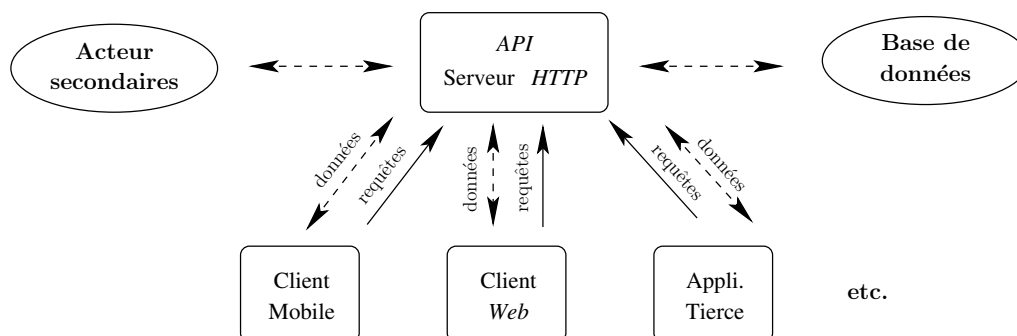


FIGURE 1 : La structure typique d'une application multi plate-formes

On aura dans la mesure du possible intérêt à limiter le plus possible le travail côté client pour les raisons suivantes :

1. L'implémentation côté client dépend de la plate-forme et l'implémentation sur le serveur constitue un forme de *factorisation* du code.
2. Sur certaines plate-formes, comme dans le cas des applications web en *JavaScript*, la sécurité et la confidentialité côté client sont très mauvaises, alors que nous pouvons implémenter la sécurité côté serveur.

Cependant, dans la mesure du possible, les opérations peu sensible, par exemple concernant l'ergonomie, se feront côté client pour limiter les coûts d'infrastructure (nombre de serveurs...) et améliorer la réactivité de l'application.

Le cas de l'application *Web*

Dans ce cours, nous étudions le développement d'applications *Web* (auxquelles on accède via un navigateur internet), avec une architecture client/serveur dans laquelle (voir la figure 2) :

- Notre *API* est un serveur *HTTP* implémenté en *PHP* avec une architecture *MVC* et *DAL*;
- Notre application côté client est en *JavaScript* (qui s'est imposé comme un langage standard côté client), et utilise la librairie *jQuery* pour la gestion des événements, des vues, et des interactions (requêtes et échange de données au format *JSON*) avec le serveur.

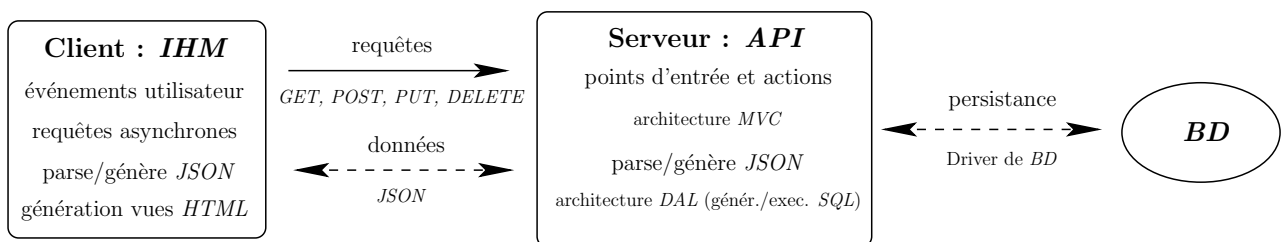


FIGURE 2 : L'architecture *client/serveur* de notre application *Web*

Chapitre 1

Premiers pas en *JavaScript*

1.1 Balise `<script>` et Hello world en *JavaScript*

Une première manière d’insérer un script *JavaScript* dans un fichier *HTML* est de mettre le code *JavaScript* dans une balise `<script></script>`. On a alors accès au document dans le code *JavaScript* et on peut sortir du code *HTML* :

exemples/bases/ex01_helloWorld.html

```
1 /<!doctype HHTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Hello World en Javascript</title>
6 </head>
7 <body>
8 <p>
9 <script>
10     document.write("Hello world !");
11 </script>
12 <p>
13 </body>
14 </html>
```

Une première manière d’insérer un script *JavaScript* dans un fichier *HTML* est de mettre le code *JavaScript* dans un fichier `.js` séparé, qui est inclus dans le *HTML* au niveau du header par une balise `<script src='...'></script>`.

exemples/bases/ex02_helloWorld.html

```
1 /<!doctype HHTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Hello World en Javascript</title>
6 <script src="/ex02_helloWorld.js"></script>
7 </head>
8 <body>
9 </body>
10 </html>
```

exemples/bases/ex02_helloWorld.js

```
1 alert("Hello World !");
```

Dans ce dernier cas, on n'a pas accès au document dans le fichier *JavaScript*, mais il y a d'autres avantages (factorisation et mise en cache du code *JavaScript* partagé entre plusieurs pages *HTML* par exemple).

1.2 Types, variables et portée

Le *JavaScript* est un langage faiblement typé, car on n'indique pas le type des variables lors de la déclaration. Lors de la déclaration des variables, le type est fixé implicitement par le type de la donnée affectée à la variable.

La déclaration de la variable peut contenir ou non le mot clef **var**. Une variable déclarée avec le mot clef **var** (on parle de déclaration *explicite*) est locale à la fonction où la variable est déclarée. Une variable déclarée sans le mot clef **var** (on parle de déclaration *implicite*) est globale.

Il n'y a pas, contrairement au *C++* ou *Java*, de visibilité locale à un bloc. Une variable déclarée n'importe où dans une fonction est visible dans toute la fonction au moins. Pour cette raison, on déclarera systématiquement les variables locales à la fonction au début du corps de la fonction, contrairement aux bonnes pratiques dans d'autres langages où on déclare la variable au plus près de son point de première utilisation.

Dans les programmes assez gros structurés en modules ou *packages*, on peut créer en *JavaScript* l'équivalent d'un *namespace* par un patron de conception consistant à mettre le code de l'ensemble d'un module dans le corps de définition d'une fonction ou dans un littéral définissant un objet (voir plus loin pour la notion d'objet).

1.3 Fonctions

Les fonctions en *JavaScript* sont déclarées par le mot clef **function**. c'est un type de données comme un autre, et une fonction peut ainsi être affectée à une variable. Voici un exemple de fonction qui calcule le prix *TTC* d'un produit à partir de son prix hors taxes. Comme les paramètres des fonctions ne sont pas typés, on peut vérifier le type des paramètres dans la fonction et éventuellement renvoyer une exception si le type du paramètre effectif n'est pas le bon.

exemples/bases/ex03_function.html

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Fonctions</title>
6 </head>
7 <body>
8 <p>
9 <script>
10 // Déclaration d'une variable de type fonction
11 var calculPrixTTC = function (prixHT, tauxTVA){
12     if (!(typeof prixHT == "number") || !(typeof tauxTVA == "number")){
```

```
13     throw new Error("Function calculPrixTTC appelée avec paramètre incorrect.")
14   }
15   return prixHT*(1.0+tauxTVA/100.0);
16 };
17
18 // Appel correct de la fonction :
19 try{
20   document.write("Prix TTC : " + calculPrixTTC(180.0, 19.6));
21 }catch (err){
22   alert(err);
23 }
24
25 // Appel incorrect de la fonction déclenchant une exception :
26 try{
27   document.write("Prix TTC : " + calculPrixTTC("coucou", 19.6));
28 }catch (err){
29   alert(err);
30 }
31 </script>
32 <p>
33 </body>
34 </html>
```

Notons que l'on peut aussi déclarer une fonction un peu comme en *PHP* de la manière suivante :

```
1 function myFunction(myParam){
2   return (myParam < 0);
3 }
```

mais la fonction est alors globale (son nom existe dans tout le programme). La bonne pratique consiste à déclarer les éléments d'un programme de sorte qu'ils aient la portée la plus locale possible, donc à déclarer la fonction avec le mot clé **var** comme dans le premier exemple de fonction ci-dessus.

1.4 Objets

Un *objet JavaScript* rassemble plusieurs propriétés, qui peuvent être des données, d'autres objets, ou encore des fonctions, alors appelées *méthodes*. Un objet n'est ni tout à fait une structure comme en *C*, ni tout à fait une classe comme dans un *langage objet classique*. Par exemple, un objet *JavaScript* n'a pas de visibilité (privée, public) pour ses propriétés. Par ailleurs, le principal mécanisme d'héritage en *JavaScript* se fait par la notion de *prototype* et est très différent de l'héritage dans les langages objet classiques. Là encore, on peut mimer une notion de visibilité via des patrons de conception.

Les noms de propriétés peuvent être

- Soit une chaîne de caractère (comme "nom de propriété!") quelconque (respecter les doubles *quotes* dans un tel cas).

- Soit des noms légaux (commençant par une lettre suivi par une suite de lettres, chiffres, et *underscores* (caractère `_`) auquel cas les doubles *quotes* sont optionnelles pour désigner le nom.

1.4.1 Création d'un objet au moyen d'un littéral

On peut créer un nouvel objet par un littéral, en définissant ses propriétés des accolades `{}`. On met alors chaque nom de propriété suivi d'un `:` suivi de la valeur de la propriété. Les propriétés ainsi construites sont séparées par des virgules.

exemples/bases/ex04_objectLiteral.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Objets</title>
6 </head>
7 <body>
8 <p>
9 <script>
10 // Littéral définissant un objet :
11 var produit = {
12   "denomination" : "Notebook sous Ubuntu 4 cores 2.0GB",
13   "prixHT" : 180.0,
14   "tauxTVA" : 19.6
15 };
16
17 // Déclaration d'une fonction avec un paramètre :
18 var calculPrixTTC = function(prod){
19   // Test d'existence des propriétés de l'objet :
20   if ("prixHT" in prod && "tauxTVA" in prod){
21     return prod.prixHT*(1.0+prod.tauxTVA/100.0);
22   }else{
23     // Rejet d'une exception personnalisée :
24     // On rejette un objet avec une prop. "name" et une prop. "message".
25     throw {name: "Bad Parameter",
26           message: "Mauvais type de paramètre pour la fonction calculPrixTTC"};
27   }
28 };
29
30 // Essai d'appel de la fonction
31 try{
32   document.write("Prix TTC du produit \""+produit.denomination
33                 +"\" : "+calculPrixTTC(produit));
34 } catch (e) { // affichage de l'exception personnalisée.
35   alert("Une erreur \"" + e.name + "\" s'est produite :\n"
36         + e.message);
37 }
38 </script>
39 <p>
40 </body>
41 </html>

```

Un objet peut contenir des propriétés qui sont de type **function**. On parle alors de *méthode* de l'objet. Dans une méthode, on accède aux propriétés de l'objet grâce à l'identificateur **this**, désignant l'objet auquel appartient la méthode.

exemples/bases/ex05_objectMethod.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Objets</title>
6 </head>
7 <body>
8 <script>
9   var produit = {
10     "denomination" : "Notebook sous Ubuntu 4 cores 2.0GB",
11     "prixHT" : 180.0,
12     "tauxTVA" : 19.6,
13
14     // Définition d'une méthode :
15     getPrixTTC : function() {
16       return this.prixHT*(1.0+this.tauxTVA/100.0);
17     }
18   };
19
20   // Fonction dans le contexte global
21   var getHtmlObjet = function(objet){
22     var chaine = "";
23     // Parcours de toutes les propriétés de l'objet (style "foreach") :
24     for (nom in objet){
25       chaine += "objet[" + nom + "] = " + objet[nom] + "<br />";
26     }
27     return chaine;
28   };
29
30   // appel d'une fonction définie dans le contexte global :
31   document.write("<p>" + getHtmlObjet(produit) + "</p>");
32
33   // appel d'une méthode :
34   document.write("<p>Prix TTC : " + produit.getPrixTTC() + "</p>");
35 </script>
36 </body>
37 </html>

```

Une méthode d'objet *JavaScript* n'est pas tout à fait comme une méthode d'un langage à objet classique, car la méthode *JavaScript* existe en autant d'exemplaires qu'il y a d'instance des objets. Nous verrons plus loin la notion de *prototype*, qui permet de créer des méthodes qui existent en un seul exemplaire pour toute une classe d'objets ayant les mêmes propriétés.

exemples/bases/ex06_nestedObjects.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Objets</title>
6 </head>

```

```

7 <body>
8 <script>
9   var produit = {
10     denomination : "Notebook sous Ubuntu",
11     prixHT_base : 180.0,
12     tauxTVA : 20.0,
13     // Objet "niché" dans un sur-objet
14     options : {
15       processor : "Intel 4 cores 2.5Ghz",
16       memory : "4GB",
17       "prix supplémentaire HT" : 50.0,
18       getHtml : function() {
19         return this.processor + " " + this.memory +
20           " (supplément : " + this["prix supplémentaire HT"] + " €euro;) ";
21       }
22     },
23     // Définition d'une méthode :
24     getHtml : function() {
25       return this.denomination +
26         "<br/>prix TTC tout compris : "
27         + (this.prixHT_base + (this.options["prix supplémentaire HT"] ||
28           0.0))
29         *(1.0+this.tauxTVA/100.0)
30         + " €euro;<br/>" + this.options.getHtml() + "<br/>";
31     }
32   };
33   // appel d'une méthode :
34   document.write("<p>" + produit.getHtml() + "</p>");
35 </script>
36 </body>
37 </html>

```

1.4.2 Constructeur d'objets, mot réservé new

On peut créer un objet via le constructeur `Object`. Voici un exemple où l'on crée un objet qui représente un produit. On crée ensuite une fonction qui calcule le prix *TTC* de ce produit après avoir testé l'existence d'attributs.

exemples/bases/ex07_objectNew.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Objets</title>
6 </head>
7 <body>
8 <p>
9 <script>
10   var produit = new Object();
11   produit.denomination = "Notebook sous Ubuntu 4 cores 2.0GB";
12   produit.prixHT = 180.0;
13   produit.tauxTVA = 20.0;
14

```

```

15  var calculPrixTTC = function(prod){
16      if ( "prixHT" in prod && "tauxTVA" in prod){
17          return prod.prixHT*(1.0+prod.tauxTVA/100.0);
18      }else{
19          throw new Error("Mauvais type de paramètre pour la fonction calculPrixTTC"
20              );
21      }
22  }
23  document.write("Prix TTC du produit \""+produit.denomination+"\" : "+
24      calculPrixTTC(produit));
25  </script>
26  <p>
27  </body>
28  </html>

```

Dans la mesure du possible, il est préférable de définir les objets *JavaScript* par des littéraux car ça peut être plus efficace que la construction dynamique avec le constructeur `Object`.

1.5 Tableaux (le type Array)

1.5.1 Notion d'Array et construction

Dans les langages classique, un tableau est une séquence d'éléments, contigus en mémoire, avec un accès aux éléments par un indice entier. En *JavaScript*, les tableaux sont des objets dont les propriétés sont automatiquement nommées avec les chaînes '0', '1', '2'. Ces tableaux possèdent certains avantages des objets, comme par exemple la possibilité d'avoir des éléments de types différents, mais sont significativement plus lents que les tableaux classiques.

Un tableau peut être créé par un littéral, entre crochets [].

exemples/bases/ex08_arrayLiterals.html

```

1  <!doctype HTML>
2  <html lang="fr">
3  <head>
4  <meta charset="UTF-8" />
5  <title>Tableaux</title>
6  </head>
7  <body>
8  <p>
9  <script>
10     // Déclaration d'une array sous forme de littéral
11     var tab = [1, 3, "coucou", 6];
12     tab[4]=9; // ajout d'un élément
13     for (var i=0 ; i<tab.length ; i++){
14         document.write(tab[i]+"<br/>");
15     }
16 </script>
17 <p>
18 </body>
19 </html>

```

Un tableau peut aussi être créé par le constructeur d'*Array*. Celui-ci peut prendre en argument soit le nombre de cases du tableau à allouer, soit les éléments du tableau initialisés lors

de la création du tableau. On peut toujours ajouter des éléments au tableau par une simple affectation de ces éléments et la mémoire évolue automatiquement.

exemples/bases/ex09_arraysNew.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Tableaux</title>
6 </head>
7 <body>
8 <p>
9 <script>
10     var tab = new Array(1, 3, "coucou", 6);
11     tab.push(9); // ajout d'un élément
12     for (var i=0 ; i<tab.length ; i++){
13         document.write(tab[i]+ "<br/>");
14     }
15 </script>
16 <p>
17 </body>
18 </html>

```

De même que pour les objets, il est préférable de définir les tableaux *JavaScript* par des littéraux car ça peut être plus efficace que la construction dynamique avec le constructeur `Array`.

1.5.2 Quelques méthodes prédéfinies sur le type `Array`

- Suppression du dernier élément (l'élément supprimé est retourné par la méthode) :
`function array.pop();`
- Suppression du premier élément (l'élément supprimé est retourné par la méthode) :
`function array.shift();`
- Suppression d'une partie des éléments :
`function array.splice(firstElementKey, numberOfElementsToRemove);`
- Ajout d'un ou plusieurs élément(s) à la fin :
`function array.push(element1, element2...);`
- Tri d'un tableau : `function array.sort(compareFuntion);`
 où `compareFuntion` est une fonction permettant de comparer deux éléments du tableau qui a pour prototype : `function compareFuntion(a, b);`
 et renvoie 0 si les éléments sont égaux, un nombre négatif si `a` est strictement inférieur à `b`, et un nombre positif si `a` est strictement supérieur à `b`.
 Pour les chaînes de caractère, on peut utiliser la méthode `string.localCompare(that)` (similaire à `strcmp`).

1.6 Exemple : traitement d'un formulaire avec *jQuery*

1.6.1 Qu'est-ce que *jQuery* ?

La librairie *jQuery*, dont on peut obtenir le code et la documentation sur api.jquery.com, permet de simplifier la gestion de différents aspects d'une application côté client en *JavaScript* :

- Gestion des événements utilisateur ;
- Récupération des valeurs saisies dans un formulaire ;
- Manipulation du document via le *DOM* ;
- Requêtes asynchrones (transfert de données entre serveur et client en dehors du chargement initial de la page) ;
- Codage des données pour la transfert (par exemple *JSON*).

Pour utiliser *jQuery*, il suffit d'insérer son code dans un script, via une balise (remplacer `x.xx.x` par le numéro de version utilisé sur jquery.com) :

```
<script src="https://code.jquery.com/jquery-x.xx.x.js"></script>
```

Pour travailler *offline*, on peut utiliser *jQuery* en local après téléchargement dans le répertoire courant :

```
<script src="./jquery-x.xx.x.js"></script>
```

Les méthodes de *jQuery* peuvent être appelées (avec des argument `args`) par l'abréviation `$(args)`.

1.6.2 Récupérer, filtrer et afficher les données d'un formulaire

Le script suivant récupère les données d'un formulaire, les filtre par expressions régulières, et les affiche en modifiant le *DOM*.

Plus précisément, le script réalise les opération suivantes :

- Déclaration d'un gestionnaire (fonction `afficheDonneesForm`) de l'événement `submit` du formulaire ayant `formStudentData` pour *ID* ;
- Dans cette fonction `afficheDonneesForm`,
 - Récupération des valeurs saisies dans les éléments ayant `nom` et `annee` pour *ID*, qui sont respectivement un *input* et un *select*.
 - Test sur la forme (expression régulière, champs obligatoire,...) sur les valeurs des champs `nom` et `année` du formulaire (à l'aide d'un littéral de type expression régulière entre slashes `/.../`).
 - Ajout dans le `` ayant `spanResultat` pour *ID* du code *HTML* du résultat de la saisie (affichage du nom et de l'année, ou le cas échéant un message d'erreur).
 - Empêcher le comportement par défaut de la soumission du formulaire (appel du script `action` côté serveur lors du *click* sur l'*input* de type `submit`).

Voici le fichier *HTML* :

[examples/bases/ex10_jQueryForm.html](#)

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Formulaires avec jQuery</title>
6 </head>
7 <body>
8     <form id="formStudentData">
9         <p>
10             <label for="nom">Nom</label>
11             <input name="nom" id="nom"/>
12         </p>
13         <p>
14             <select name="annee" id="annee">
15                 <option value="choisissez" selected disabled>— choisissez —</option>
16                 <option value="premiere">Première année</option>
17                 <option value="deuxieme">Deuxième année</option>
18             </select>
19         </p>
20         <p>
21             <input type="submit" value="Valider"/>
22         </p>
23     </form>
24
25     <p>
26         <strong>données saisies&nbsp;  ;</strong><br/>
27
28         <!-- Les résultats de la saisie du formulaire vont s'afficher dans ce span
29             -->
30         <span id="spanResultat"></span>
31     </p>
32 <script src="jquery.js"></script>
33 <script src="ex10_jQueryForm.js"></script>
34 </body>
35 </html>

```

Il est recommandé de mettre, dans la mesure du possible, le script à la fin du document, car cela limite le coût et le délai des chargements et *parsing* de la librairie *jQuery* lors d'un premier chargement (ou rafraîchissement) de la page. Voici le fichier *JavaScript* :

exemples/bases/ex10_jQueryForm.js

```

1 // Méthode de récupération et affichage des inputs de formulaire
2 // à l'aide de jQuery :
3 var afficheDonneesForm = function(event){
4     // récupération (via jQuery) de la valeur de l'input d'ID "nom"
5     var nom = $("#nom").val();
6     // récupération (via jQuery) de la valeur du select d'ID "annee"
7     var annee = $("#annee").val();
8
9     // test de champs obligatoires et expressions régulières
10    if (annee=="choisissez" ||
11        !/^([a-zA-ZÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæœ,èéêëìíîïðñòóôõ÷øùúûÄÅäüýþÿ
        \s"'-]{1,}$)/

```

```
12     .test(nom)){
13     $("#spanResultat").html("Problème : forme d'un champs incorrect.");
14 }else{
15     $("#spanResultat").html("<em>Nom : </em>" + nom + "<br />" +
16         "<em>Annee : </em>" + annee );
17 }
18 // Éviter d'appeler l'"action" par défaut () script PHP, etc...)
19 //du formulaire lors du submit
20 event.preventDefault();
21 };
22
23 // Gestion de l'événement submit du formulaire.
24 // On définit afficheDonneesForm comme gestionnaire (handler)
25 //de l'événement
26 $("#formStudentData").on("submit", afficheDonneesForm);
```


Chapitre 2

Programmation Fonctionnelle et Objet en *JavaScript*

On distingue en *JavaScript* deux catégories de *patterns* (et éventuellement des *patterns* hybrides) :

- Les *patterns* dits *fonctionnels* s'appuient sur les aspects de *JavaScript* en tant que *langage fonctionnel*. Autrement dit, ces *patterns* exploitent les propriétés des fonctions *JavaScript* en tant que données, ainsi que la portée des variables dans ces fonctions.
- Les *patterns* dits *prototypaux* s'appuient sur les aspects de *JavaScript* en tant que *langage prototypal*. Ceci est lié à une propriété que possèdent tous les objets *JavaScript*, appelée le **prototype**. Le prototype permet de partager des propriétés entre plusieurs objets, et il conduit naturellement à des notions d'héritage. Il permet aussi d'*augmenter* les objets pour leur rajouter des propriétés, bien après que ces objets aient été définis, y compris sur les types de base comme **String**.

Nous commencerons par voir un certain nombre de *patterns* fonctionnels, qui permettent de faire de la programmation objet avec des notions comme la visibilité, la structuration d'une application en modules (ou *packages*), des fabriques, ou encore des *patterns* permettant le découplage des composants d'une application à base d'événements, ou comme *subscriber/publisher*.

Ces *patterns* peuvent paraître déconcertant au premier abord pour un développeur habitué aux langages objet classiques. Avec un peu d'habitude, on en vient à considérer que *JavaScript* est un excellent langage objet, très expressif et très souple. Cependant, certains problèmes de conception du langage, qui n'ont pu être corrigés pour assurer la compatibilité ascendante, nécessitent quelques précautions, sous la forme de bonnes habitudes.

2.1 Le *Pattern* Module

Le *pattern Module* permet de créer des composants qui peuvent jouer le rôle que jouent les classes dans les langages objet classiques. Il permet, entre autre, de créer des données et méthodes privées, et une interface publique avec d'autres données et méthodes, qui sont accessibles de l'extérieur, et qui peuvent, elles, accéder aux propriétés privées.

Le *pattern* consiste à créer une fonction. Les données et méthodes privées sont simplement des variables locales de la fonction. Elles ne sont donc pas visibles du monde extérieur à la

fonction. La fonction renvoie un objet, qui constitue l'interface publique du module, dont les propriétés (données, objets ou fonctions) accèdent aux variables privées. Lorsque l'objet est retourné, on ne peut plus accéder directement aux variables locales de la fonction, mais celles-ci restent vivantes (leur cycle de vie ne se termine pas) tant que l'objet retourné qui s'y réfère n'est pas lui-même détruit. Autrement dit, on peut continuer à manipuler ces variables locales au travers des méthodes de l'interface publique.

exemples/objet/ex01_modulePattern.js

```
1 var mySecretModule = function(defaultSecretValue){
2
3   // donnée privée avec une valeur par défaut
4   var myPrivateSecret = ((defaultSecretValue &&
5                           typeof defaultSecretValue === "string")
6                           && defaultSecretValue)
7                           || "";
8
9   // Méthode privée :
10  var myRegexTestMethod = function(chaine){
11    return (typeof chaine === "string") && /^[a-z]*$/i.test(chaine);
12  };
13
14  // On crée un objet qui va être rendu public
15  // Cet objet va être retourné, mais pas les données privées.
16  var publicInterface = {
17
18    // Les éléments publics sont les propriétés de publicInterface
19
20    // Donnée publique :
21    donneePublique : 'donnée par défaut ',
22
23    // Setter
24    setSecret : function(secretValue){
25      // Test d'expression régulière
26      if (myRegexTestMethod(secretValue)){
27        myPrivateSecret = secretValue;
28      }else{
29        throw {
30          name : "IllegalArgumentException",
31          message : "Le secret " + secretValue + " est invalide."
32        };
33      }
34    },
35
36    // Accesseur :
37    getSecret : function(){
38      return myPrivateSecret;
39    },
40
41  }; // Fin de publicInterface
42
43  return publicInterface;
44 }
```

exemples/objet/ex01_modulePattern.html



```

1  /<!doctype HTML>
2  <html lang="fr">
3  <head>
4  <meta charset="UTF-8" />
5  <title>Pattern "Module"</title>
6
7  </head>
8  <body>
9  <p>
10 <script src="./ex01_modulePattern.js"></script>
11 <script>
12     var secretModule = mySecretModule("initSecret");
13
14     document.write("donneePublique : " + secretModule.donneePublique + "<br/>");
15
16     // Modification de la donnée publique
17     secretModule.donneePublique = "nouvelle donnée publique";
18     document.write("donneePublique : " + secretModule.donneePublique + "<br/>");
19
20     // Accesseur de secret :
21     document.write("Secret : " + secretModule.getSecret() + "<br/>");
22
23     // Tentative de modifier le secret
24     try{
25         secretModule.setSecret("abcde");
26         document.write("Secret : " + secretModule.getSecret() + "<br/>");
27     } catch (e) {
28         document.write("Erreur de type " + e.name + "<br/>Message : " + e.message +
29             "<br/>");
30     }
31
32     try{
33         secretModule.setSecret("abcde567");
34         document.write("Secret : " + secretModule.getSecret() + "<br/>");
35     } catch (e) {
36         document.write("Erreur de type " + e.name + " : " + e.message + "<br/>");
37     }
38 </script>
39 </p>
40 </body>
41 </html>

```

Lé mécanisme du langage essentiel pour ce *pattern* est la portée (*scope*) des variables locales à une fonction, qui s'étend aux sous-fonctions de la fonction, et à leurs sous-fonctions...

2.2 Passage d'arguments par objets

En *JavaScript*, il est souvent plus pratique, plutôt que de passer une série de paramètres, ce qui oblige à tenir compte de l'ordre de ces paramètres, de donner en argument à une fonction les données dans les propriétés d'un objet, soit construit à la volée, soit construit auparavant.

Ce *pattern* offre souvent plus de souplesse que la manière classique. Dans l'exemple suivant, la fonction génère le code *HTML* de l'objet passé en paramètre, sans savoir de quel type d'objet il s'agit. On l'utilise ensuite pour afficher une adresse.

exemples/objet/ex02_affichageObjetBasic.js

```

1  /**
2   * Crée un chaîne de caractère lisible qui représente l'objet.
3   * On suppose que toutes les propriétés de l'objet sont de type chaîne ou nombre.
4   * (elles peuvent être automatiquement converties en chaîne)
5   */
6  var objectToHtmlTable = function(spec){
7
8      var chaine = "<table><tbody>";
9      // Parcours des propriétés de l'objet spec passé en argument
10     for (propertyName in spec){
11         // La propriété est définie et non vide.
12         // Elle ne vient pas du prototype de l'objet
13         // Ce n'est pas une fonction
14         if (spec[propertyName] && spec.hasOwnProperty(propertyName)
15             && typeof spec[propertyName] !== "function"){
16             // Concaténation à une chaîne. Les nombres sont convertis.
17             chaine += '<tr><td style="text-align : right;"><em>' + propertyName + " :</em></td>' +
18                     "<td>" + spec[propertyName] + "</td></tr>";
19         }
20     };
21     chaine += "<tbody></table>";
22     return chaine;
23 };

```



exemples/objet/ex02_affichageObjetBasic.html

```

1  <!doctype HTML>

```

```

2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Affichage d'Objets Générique</title>
6
7 </head>
8 <body>
9 <p>
10 <script src="/ex02_affichageObjetBasic.js"></script>
11 <script>
12     document.write(objectToHtmlTable({
13         id: "0f3ea759b1",
14         numeroRue: "2 bis",
15         rue: "Rue de la Paix",
16         complementAdresse: "",
17         codePostal: "63000",
18         ville: "Clermont-Ferrand",
19         pays: "France"
20     }));
21 </script>
22 <p>
23 </body>
24 </html>

```

2.3 Exemple de fabrique sommaire

Dans l'exemple suivant, une fabrique, suivant un *pattern* module très sommaire, construit un objet de type adresse (éventuellement partiellement rempli), en sélectionnant les propriétés que l'objet en paramètre qui sont dans une liste.

Cet exemple est plutôt un *exemple d'école* et nous verrons plus loin des exemple plus complets, les propriété de l'objet retourné manipulant des données (attributs) privées.

exemples/objet/ex03_methodLiteralParam.js

```

1 var fabriqueAdresseVersion1 = function(spec){
2     // Objet à retourner initialement vide
3     var adresse = {};
4
5     var listeProprietes = ["id", "numeroRue", "rue", "complementAdresse",
6                             "codePostal", "ville", "pays"];
7
8     // Parcours des propriétés de l'objet spec passé en argument
9     for (propertyName in spec){
10         if (spec.hasOwnProperty(propertyName)){
11             // Si la propriété existe dans le type adresse :
12             if (listeProprietes.indexOf(propertyName) >= 0){
13                 adresse[propertyName] = spec[propertyName];
14             }else{
15                 throw {name: "UnknownPropertyException",
16                         message: "Propriété de l'adresse inconnue."};
17             }
18         }
19     }
20 }

```

```

21   return adresse;
22 };

```

exemples/objet/ex03_methodLitteralParam.html

```

1  <!doctype HTML>
2  <html lang="fr">
3  <head>
4  <meta charset="UTF-8" />
5  <title>Affichage Générique d'Objets</title>
6
7  </head>
8  <body>
9  <p>
10 <script src="./ex02_affichageObjetBasic.js"></script>
11 <script src="./ex03_methodLitteralParam.js"></script>
12 <script>
13     // création d'une instance
14     var adresse = fabriqueAdresseVersion1({
15         id : "0f3ea759b1",
16         numeroRue : "2 bis",
17         rue : "Rue de la Paix",
18         complementAdresse : "",
19         codePostal : "63000",
20         ville : "Clermont-Ferrand",
21         pays : "France"
22     });
23
24     document.write(objectToHtmlTable(adresse));
25 </script>
26 <p>
27 </body>
28 </html>

```

2.4 Structuration d'une application

L'un des principaux défauts de *JavaScript* est sa tendance à créer, parfois sans faire exprès, des variables globales, ce qui a tendance à créer des interactions involontaires entre des parties du code qui n'ont rien à voir, ce qui génère des *bugs* difficiles à débusquer...

Nous allons voir maintenant comment rédiore les nombres de variables globales de notre programme à une seule variable, ici appelée **myApp**, qui contient toute notre application.

L'objet **myApp**, initialement, ne contient que deux méthodes :

- Une méthode **addModule** qui permet d'ajouter un objet quelconque (de type **Objet**, **Function**, **Array**, etc.) sous la forme de propriété de l'application.
- Une méthode **init**, qui permet de rajouter un ensemble de propriétés prédéfinies, sans avoir à les créer une par une.

exemples/objet/ex04_structureApplication.js

```

1  /** Définition d'une variable application.
2   * L'application est initialement vide et ne comporte que la fonctionnalité

```

```

3  * permettant d'ajouter des modules.
4  * Une méthode init() permet d'initialiser plusieurs modules.
5  *
6  */
7  var myApp = {
8    /** Méthode qui ajoute un module à notre application
9     * Un module peut être n'importe quel objet qui contient
10    * des données ou des méthodes...
11    * @method addModule
12    * @param {(Object | function | string | regex | number)} moduleObject
13    *          - un objet ou valeur quelconque à ajouter à notre application.
14    */
15    addModule : function(moduleName, moduleObject){
16      if (typeof moduleName === "string" &&
17          /^[a-z]{1,}[a-z0-9\_\-]*$/i.test(moduleName)){
18        this[moduleName] = moduleObject;
19      }else{
20        throw {
21          name : "IllegalArgumentException",
22          message : "Impossible de créer le module : nom " + moduleName
23                  + " illégal"
24        }
25      }
26    },
27
28    /** Ajoute toutes les propriétés d'un objet à notre application.
29     * @method init
30     * @param {Object} spec - objet contenant les propriétés à ajouter.
31     */
32    init : function(spec){
33      for (propertyName in spec){
34        if (spec.hasOwnProperty(propertyName)){
35          this.addModule(propertyName, spec[propertyName]);
36        }
37      }
38    }
39  };
40
41  // Initialisation de l'application avec un module metier
42  // metier est initialement vide.
43  myApp.init({
44    metier : {}
45  });

```

Nous utilisons maintenant ce squelette d'application et nous créons dans notre application un module **metier**.

Nous utilisons ensuite le *pattern apply* qui nous permet d'utiliser la méthode `myApp.addModule` en prenant comme **"this"** un autre objet que `myApp`.

En appliquant donc la méthode `myApp.addModule` en prenant `myApp.metier` comme **"this"**, nous créons un sous-module de `myApp.metier`, appelé `myApp.metier.sousModule`. Ce sous-module contient une propriété **essai**.

exemples/objet/ex04_structureApplication.html

```

1  <!doctype HTML>
2  <html lang="fr">

```

```

3 <head>
4 <meta charset="UTF-8" />
5 <title>Classes</title>
6 <style>
7 table {border-collapse : collapse}
8 tbody tr td,th {border-style : solid; text-align : center; padding : 4pt;}
9 </style>
10 </head>
11 <body>
12 <p>
13 <script src="ex04_structureApplication.js"></script>
14 <script>
15 // ajout d'une propriété au métier :
16 myApp.metier.coucou = "test";
17
18 // Ajout d'un sous-module au module myApp.metier
19 // On applique (pattern apply) addModule en prenant this = myApp.metier
20 myApp.addModule.apply(myApp.metier ,
21                       [ "sousModule",
22                         {essai : "Je suis la propriété \"essai\" du sous module"}
23                       ]);
24
25 // Ajout d'une méthode mainFunction
26 myApp.addModule( "mainFunction", function() {
27   document.write( "Fonction myApp.mainFunction <br/>" );
28   document.write( "myApp.metier.coucou : " + myApp.metier.coucou + "<br/>" );
29   document.write( "myApp.metier.sousModule.essai : " + myApp.metier.sousModule.
30     essai );
31 });
32
33 // Exécution de la méthode mainFunction
34 myApp.mainFunction();
35 </script>
36 <p>
37 </body>
38 </html>

```

2.5 Exemple : un module `metier.regexUtil`

L'exemple suivant montre l'utilisation du *pattern* Module pour créer un sous-module métier utilitaire pour tester des expressions régulières courantes :

- Expressions formées avec les caractères du langage courant dans une des langues dont les accents sont normalisés dans la norme ISO 8859 – 1 (*Latin-1*, Europe occidentale), admettant aussi les guillemets, apostrophes et traits d'union (tiret haut).
- Mêmes caractère que la précédente mais admettant en outre les chiffres.
- Mêmes caractère que la précédente mais admettant en outre les caractères de ponctuation (; . , ! ? :) et les parenthèses.

Trois expressions régulières constantes (donc pré-compilées) sont définies comme données statiques (en un seul exemplaire) privées. L'interface fournit trois méthodes pour tester ces

expressions régulières sur une chaîne, avec éventuellement des conditions de longueur minimale ou maximale sur la chaîne (exemple : champs obligatoire...).

exemples/objet/ex05_modulePatternRegex.js

```

1  /**
2   * Ajoute au métier un objet qui est l'interface publique
3   * d'une fonction qui suit le "pattern module".
4   * La fonction retourne son interface publique qui est un objet.
5   * Cet objet est ajouté comme sous-module au module "metier".
6   *
7   * @module regexUtil
8   * @augments myApp.metier
9   */
10 myApp.addModule.apply(myApp.metier, [ "regexUtil", function() {
11
12     //////////////////////////////////////
13     // Propriétés et méthodes "statiques" privées
14
15     /**
16      * Expression régulière constante pour la langue naturelle (et espaces)
17      * @constant
18      * @private
19      */
20     var regexLatin1
21         = /^[a-zA-ZÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæ,èéêëìíîïðñòóôõ÷øùú
22             Åäÿþÿ\s"'\-]*$/i;
23
24     /**
25      * Expression régulière constante pour la langue naturelle et chiffres
26      * @constant
27      * @private
28      */
29     var regexLatin1WithDigits =
30         /^[a-zA-ZÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæ,èéêëìíîïðñòóôõ÷øùú
31             Åäÿþÿ\s"'\-0-9]*$/i;
32
33     /**
34      * Expression régulière constante pour la langue naturelle et chiffres ou
35      * ponctuation
36      * @constant
37      * @private
38      */
39     var regexLatin1WithDigitsPunctuation =
40         /^[a-zA-ZÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæ,èéêëìíîïðñòóôõ÷øùú
41             Åäÿþÿ\s"'\-0-9|;|.|!|?|:|(|) ]*/i;
42
43     /**
44      * Valide une expression régulière sur une chaîne (avec conditions de
45      * longueur)
46      * @method validateRegex
47      * @param {Object} spec - objet contenant les données du test à effectuer
48      * @param {function} spec.regexTest - fonction de test qui renvoie true en
49      *     cas de succès
50      *                                     et un message d'erreur en cas d'échec
51      * @param {string} spec.chaine - chaîne de caractères à tester
52      * @param {number} [minLength=0] - longueur minimale pour la chaîne

```

```

47      * @param {number} [maxLength] - longueur maximale pour la chaîne (défaut :
48      *   illimité)
49      * @return {boolean} true si les conditions sont satisfaites, false sinon.
50      */
51      var validateRegex = function(spec){
52          if (typeof spec.chaine === "string"
53              && (!spec.minLength || spec.chaine.length >= spec.minLength)
54              && (!spec.maxLength || spec.chaine.length <= spec.maxLength)
55          ){
56              return spec.regex.test(spec.chaine);
57          }
58          return false;
59      };
60
61      //////////////////////////////////////
62      // Interface publique du module
63
64      /**
65       * Création d'un objet contenant les données et méthodes publiques
66       * (les propriétés publiques sont retournées par la fonction "module").
67       */
68      var publicInterfaceRegex = {
69
70          /**
71           * méthode publique : test d'expression du langage avec chiffres
72           * @method testRegexLatin1
73           * @param {Object} spec - objet contenant les données du test à effectuer
74           * @param {string} spec.chaine - chaîne de caractères à tester
75           * @param {number} [minLength=0] - longueur minimale pour la chaîne
76           * @param {number} [maxLength] - longueur maximale pour la chaîne
77           * @return {boolean|string} true si les conditions sont satisfaites, un
78           *   message d'erreur sinon.
79           */
80          testRegexLatin1 : function(spec){
81              // Ajout d'une propriété à spec (augmentation)
82              spec.regex = regexLatin1;
83              return validateRegex(spec);
84          },
85
86          /**
87           * méthode publique : test d'expression du langage avec chiffres
88           * @method testRegexLatin1WithDigits
89           * @param {Object} spec - objet contenant les données du test à effectuer
90           * @param {string} spec.chaine - chaîne de caractères à tester
91           * @param {number} [minLength=0] - longueur minimale pour la chaîne
92           * @param {number} [maxLength] - longueur maximale pour la chaîne
93           * @return {boolean|string} true si les conditions sont satisfaites, un
94           *   message d'erreur sinon.
95           */
96          testRegexLatin1WithDigits : function(spec){
97              // Ajout d'une propriété à spec (augmentation)
98              spec.regex = regexLatin1WithDigits;
99              return validateRegex(spec);
100          },

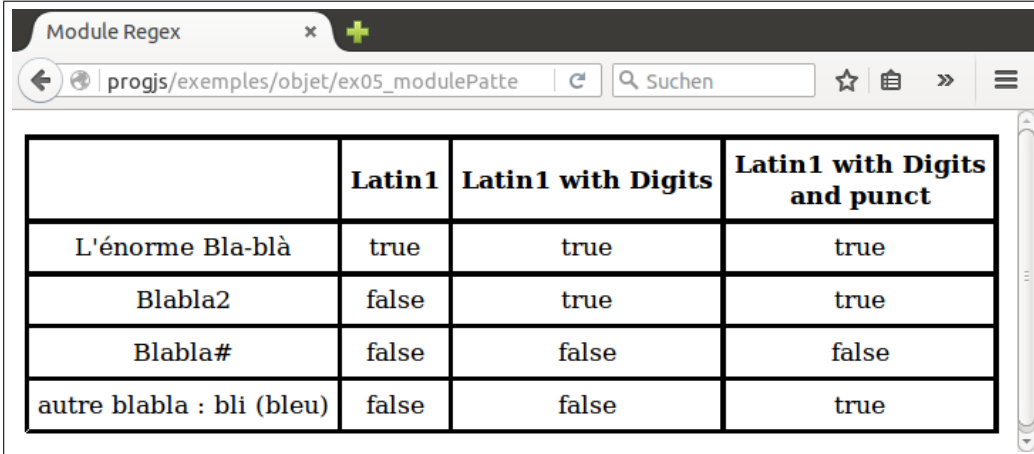
```

```

100  /**
101   * méthode publique : test d'expression du langage avec chiffres
102   * @method testRegexLatin1WithDigitsPunctuation
103   * @param {Object} spec - objet contenant les données du test à effectuer
104   * @param {string} spec.chaine - chaîne de caractères à tester
105   * @param {number} [minLength=0] - longueur minimale pour la chaîne
106   * @param {number} [maxLength] - longueur maximale pour la chaîne
107   * @return {boolean/string} true si les conditions sont satisfaites, un
    message d'erreur sinon.
108   */
109  testRegexLatin1WithDigitsPunctuation : function(spec){
110    // Ajout d'une propriété à spec (augmentation)
111    spec.regex = regexLatin1WithDigitsPunctuation;
112    return validateRegex(spec);
113  }
114
115  }; // fin de publicInterfaceRegex
116
117  // On retourne l'objet contenant l'interface publique.
118  return publicInterfaceRegex;
119
120  }()) // fin ET APPEL de la fonction qui crée l'objet "regexUtil"
121
122  ); // fin de l'appel "apply" de la méthode myApp.addModule
123  // (ajout de l'objet publicInterfaceRegex au métier, sous le nom regexUtil)

```

Le fichier *HTML* réalise des tests des méthodes du module `regexUtil` sur un jeu de chaînes, et affiche les résultats dans une table.



The screenshot shows a web browser window titled "Module Regex". The address bar displays "progjs/exemples/objet/ex05_modulePatte". The page content is a table with four columns: an empty header cell, "Latin1", "Latin1 with Digits", and "Latin1 with Digits and punct". There are four rows of test data.

	Latin1	Latin1 with Digits	Latin1 with Digits and punct
L'énorme Bla-blà	true	true	true
Blabla2	false	true	true
Blabla#	false	false	false
autre blabla : bli (bleu)	false	false	true

exemples/objet/ex05_modulePatternRegex.html

```

1  <!doctype HTML>
2  <html lang="fr">
3  <head>
4    <meta charset="UTF-8" />
5    <title>Module Regex</title>
6    <link rel="stylesheet", href="ex05_modulePatternRegex.css"/>
7  </head>
8  <body>
9    <!-- Création de l'application vide avec deux méthodes -->
10   <script src="ex04_structureApplication.js"></script>

```

```

11 <!-- Création de sous-module regexUtil de myApp.metier -->
12 <script src="/ex05_modulePatternRegex.js"></script>
13
14 <!-- Ajout d'un main et exécution -->
15 <script>
16   // Ajout d'une méthode mainFunction
17   myApp.addModule("mainFunction", function() {
18     // Chaînes pour les tests d'expressions régulières :
19     var tabChaines = [ "L'énorme Bla-blà", "Blabla2", "Blabla#", "autre blabla
20                       : bli (bleu)" ];
21     var i;
22     // Raccourci par copie de référence :
23     var regexUtil = myApp.metier.regexUtil;
24     var texte = "<table><thead>" +
25                 "<tr><th></th><th>Latin1</th><th>Latin1 with Digits</th>" +
26                 "<th>Latin1 with Digits<br />and punct</th></tr></thead><tbody>"
27                 ;
28     for (i = 0 ; i < tabChaines.length ; i++){
29       texte += "<tr><td>" + tabChaines[i] + "</td>"
30              + "<td>" + regexUtil.testRegexLatin1({
31                  chaine : tabChaines[i]
32                }) + "</td>"
33              + "<td>" + regexUtil.testRegexLatin1WithDigits({
34                  chaine : tabChaines[i]
35                }) + "</td>"
36              + "<td>" + regexUtil.testRegexLatin1WithDigitsPunctuation({
37                  chaine : tabChaines[i]
38                }) + "</td>"
39              ;
40     }
41     texte += "</tbody></table>";
42
43     document.write(texte);
44   }); // fin de myApp.addModule("mainFunction")
45
46   // Lancement de l'application :
47   myApp.mainFunction();
48 </script>
49 </body>
50 </html>

```

2.6 Module Métier adresse

Nous créons maintenant un sous-module `myApp.metier.adresse`. Celui-ci comporte une partie privée, avec notamment une donnée membre statique `propertiesPatterns`, qui définit la structure d'une adresse. Ici, `propertiesPatterns` définit, pour chaque propriété, une méthode de test par expression régulière de validité de la valeur, et une propriété `labelText` à afficher pour indiquer à l'utilisateur de quelle propriété il s'agit (typiquement : texte de `label` associé à un `input` dans un formulaire). On pourrait facilement adapter le code pour permettre des propriétés calculées.

L'interface propose quelques méthodes statiques utilitaires, comme l'accès à la liste des noms de propriétés, aux sonnées `labelText`, ou le test d'expression régulière d'une propriété.

exemples/objet/ex06_moduleMetierAdresse.js

```

1  /**
2   * Définit les propriétés générale des objets métiers représentant des adresses.
3   * On ajoute au métier un objet qui est l'interface publique d'une fonction qui
4     suit le pattern "module".
5   * La fonction retourne son interface publique qui est un objet.
6   * Cet objet est ajouté comme sous-module au module "metier".
7   *
8   * Dans cet objet, on ne trouve pas pour le moment les propriétés d'instance.
9     Celles-ci seront ajoutées par "augmentation".
10  *
11  * @module adresse
12  * @augments myApp.metier
13  */
14  myApp.addModule.apply(myApp.metier, [ "adresse", function() {
15
16      // Raccourci (alias) vers le module de regex
17      var regexUtil = myApp.metier.regexUtil;
18
19      //////////////////////////////////////
20      // Propriétés et méthodes "statiques" privées
21
22      /**
23       * Définit la structure des objets de type adresse :
24       * propriétés attendues, forme de ces données...
25       *
26       * @constant
27       * @private
28       *
29       * @property {Object} id - Propriétés de l'identifiant unique de l'instance
30       * @property {Object} numéroRue - Propriétés du numéro de la rue
31       * @property {Object} rue - Propriétés du nom de la rue/place
32       * @property {Object} complementAdresse - Propriétés du complément Lieu dit/
33         Bâtiment...
34       * @property {Object} codePostal - Propriétés du code postal
35       * @property {Object} ville - Propriétés du nom de la ville
36       * @property {Object} numéroRue - Propriétés du nom du pays
37       */
38
39      var propertiesPatterns = {
40        id : {
41          regexTest : function(chaine){
42            if (/^[0-9a-f]{10}$/i.test(chaine) === true){
43              return true;
44            }else{
45              return "L'identifiant doit comporter 10 chiffres hexa.";
46            }
47          },
48          labelText : "Identifiant"
49        },
50        numeroRue : {
51          regexTest : function(chaine){
52            if (regexUtil.testRegexLatin1WithDigits({
53              chaine : chaine,
54              maxLength : 15
55            }) === true)
56            {
57              return true;
58            }
59          }
60        }
61      };
62    });
63  }

```

```

54         }else{
55             return "Le numéro de la rue contient au plus 15 caractères, "
56                 + "lettres, tirets et guillemets ou chiffres.";
57         }
58     },
59     labelText : "Numéro"
60 },
61 rue : {
62     regexTest : function(chaine){
63         if (regexUtil.testRegexLatin1WithDigits({
64             chaine : chaine,
65             minLength : 1,
66             maxLength : 255
67         }) == true)
68         {
69             return true;
70         }else{
71             return "le nom de la rue/place, obligatoire ne contient que "
72                 + "des lettres, tirets et guillemets ou chiffres.";
73         }
74     },
75     labelText : "rue/place"
76 },
77 complementAdresse : {
78     regexTest : function(chaine){
79         if (regexUtil.testRegexLatin1WithDigitsPunctuation({
80             chaine : chaine,
81             maxLength : 255
82         }) == true)
83         {
84             return true;
85         }else{
86             return "le complément d'adresse ne contient que des lettres, "
87                 + "tirets et guillemets ou chiffres.";
88         }
89     },
90     labelText : "Lieu dit, Bâtiment, BP"
91 },
92 codePostal : {
93     regexTest : function(chaine){
94         if (/^[0-9]{5}$/.test(chaine) == true){
95             return true;
96         }else{
97             return "Le code postal doit comporter 5 chiffres décimaux.";
98         }
99     },
100    labelText : "Code Postal"
101 },
102 ville : {
103     regexTest : function(chaine){
104         if (regexUtil.testRegexLatin1({
105             chaine : chaine,
106             minLength : 1,
107             maxLength : 255
108         }) == true)
109         {

```

```

110         return true;
111     }else{
112         return "La ville, obligatoire, ne contient que des lettres, "
113             + "tirets et guillemets.";
114     }
115 },
116 labelText : "Ville"
117 },
118 pays : {
119     regexTest : function(chaine){
120         if (regexUtil.testRegexLatin1({
121             chaine : chaine,
122             minLength : 1,
123             maxLength : 255
124         }) == true)
125         {
126             return true;
127         }else{
128             return "Le pays, obligatoire, ne contient que des lettres, "
129                 + "tirets et guillemets.";
130         }
131     },
132     labelText : "Pays"
133 }
134 }; // fin de l'objet propertiesPatterns
135
136 /**
137  * Tableau contenant la liste des propriétés attendues d'une instance d'
138  *   adresse. Le tableau est précalculé lors de l'initialisation.
139  * @member
140  * @private
141  */
142 var propertyList = function(){
143     var liste = [];
144
145     // Parcours des propriétés de l'objet propertiesPatterns.regexTest
146     // qui correspondent aux propriétés de l'adresse
147     for (var propertyName in propertiesPatterns){
148         // Ne pas considérer les propriétés "héritées" du prototype.
149         if (propertiesPatterns.hasOwnProperty(propertyName)){
150             liste.push(propertyName);
151         }
152     }
153
154     return liste;
155 }(); // appel immédiat de la fonction anonyme.
156
157 //////////////////////////////////////
158 // Interface publique du module
159
160 /**
161  * Création d'un objet contenant les données et méthodes publiques
162  * (les propriétés publiques sont retournées par la fonction "module").
163  */
164 var publicInterfaceAdresse = {

```

```

165
166  /**
167   *  Renvoie la liste des propriétés attendues des instances d'adresse.
168   *  @method getPropertyList
169   */
170  getPropertyList : function(){
171    return propertyList;
172  },
173
174  /**
175   *  Renvoie le texte de description de la propriété attendue des instances d'
176   *  adresse.
177   *  Renvoie undefined en cas d'erreur (propriété inconnue)
178   *  @method getLabelText
179   *  @param {string} propertyName – nom de propriété
180   *  @return {string} le texte de description courte du champs
181   */
182  getLabelText : function(propertyName){
183    return propertiesPatterns[propertyName].labelText;
184  },
185
186  /**
187   *  Expose le test d'expression régulière des propriétés attendues des
188   *  instances d'adresse.
189   *  Peut être utilisée pour le filtrage des données d'un formulaire.
190   *  @method testRegex
191   *  @param {string} propertyName – nom de propriété
192   *  @param {string} value – valeur pour initialiser la propriété
193   *  @return {boolean|string} true si la chaîne est un code postal valide, un
194   *  message d'erreur sinon.
195   */
196  testRegex : function(propertyName, value){
197    if (propertiesPatterns[propertyName] === undefined){
198      return "La propriété " + propertyName + " n'existe pas";
199    }else{
200      return propertiesPatterns[propertyName].regexTest(value);
201    }
202  }; // fin de l'objet publicInterfaceAdresse
203
204  return publicInterfaceAdresse;
205
206 }() // fin ET APPEL de la fonction qui crée l'objet "publicInterfaceAdresse"
207
208 ); // fin de l'appel "apply" de la méthode myApp.addModule
209 // (ajout de l'objet publicInterfaceAdresse au métier, sous le nom adresse)

```

Nous créons ensuite, via un *pattern* Module, une fabrique d'instances d'adresse. Celle-ci prend comme paramètre un objet contenant des valeurs pour initialiser les propriétés, effectue les tests d'expressions régulières, et crée deux objets privés. Le premier objet, appelé **adresse**, contient les propriétés de l'objet adresse. Le deuxième objet, appelé **dataError**, contient, pour chaque propriété pour laquelle une erreur a été détectée au filtrage, un message d'erreur.

Des méthodes publiques, dans l'interface du module, permettent d'accéder à, ou de modifier

les données de l'instance.

exemples/objet/ex06_fabriqueAdresse.js

```

1  /**
2   * Fabrique qui crée des objets représentant des adresse, suivant le "pattern
3   * Le paramètre spec de notre fonction est un objet contenant les propriétés d'
4   * une adresse à créer.
5   *
6   * @method createInstance
7   * @augments myApp.metier.adresse
8   * @param {Object} inputObj - spécification des propriétés d'une instance d'
9   * adresse
10  * @param {string} inputObj.id - identifiant unique de l'instance
11  * @param {string} inputObj.numeroRue - numero de rue
12  * @param {string} inputObj.rue - nom de rue
13  * @param {string} inputObj.complementAdresse - complément d'adresse (lieu dut,
14  * bâtiment, résidence, etc.)
15  * @param {string} inputObj.codePostal - code postal
16  * @param {string} inputObj.ville - nom de ville
17  * @param {string} inputObj.pays - nom de pays
18  */
19 myApp.addModule.apply(myApp.metier.adresse, ["createInstance", function(inputObj
20 ) {
21
22     //////////////////////////////////////
23     // Propriétés et méthodes "statiques" privées
24
25     /**
26     * Objet privé contenant les propriétés de l'instance, initialement vide
27     * @member
28     * @private
29     */
30     var adresse = {};
31
32     /**
33     * Objet privé contenant les messages d'erreur associés aux propriétés
34     * attendues des instances.
35     * @member
36     * @private
37     */
38     var dataError = false;
39
40     /**
41     * Ajoute une propriété (message d'erreur) dans dataError
42     * @method addError
43     * @private
44     */
45     var addError = function(propertyName, message){
46         // si dataError n'existe pas, on le crée
47         if (dataError === false){
48             dataError = {};
49         }
50         // Ajout d'une propriété
51         dataError[propertyName] = message;
52     }
53

```

```

48  /**
49   * Setter : initialise la valeur pour une propriété attendues d'une instance.
50   * En cas d'erreur un message pour cette propriété est est ajouté dans
      dataError.
51   * En l'absence d'erreur, une éventuelle erreur précédente est effacée.
52   * @method addError
53   * @private
54   */
55  var setPropertyOrError = function(propertyName, value){
56      var resultTestRegex = myApp.metier.adresse.testRegex(propertyName, value);
57      // On initialise la propriété de l'adresse
58      adresse[propertyName] = value;
59      // Si la validation par expression régulière est passée
60      if (resultTestRegex === true){
61          // On efface une vieille erreur éventuelle
62          delete dataError[propertyName];
63      }else{
64          // On initialise la propriété de l'objet des erreurs.
65          // avec le message d'erreur.
66          addError(propertyName, "Propriété " + value +
67              " invalide : " + resultTestRegex);
68      }
69  }
70
71  // Parcours des propriétés de getPropertyList()
72  // qui correspondent aux propriétés de l'adresse à créer
73  for (var i=0 ; i<this.getPropertyList().length ; ++i){
74      var propertyName = this.getPropertyList()[i];
75      setPropertyOrError(propertyName, inputObj[propertyName]);
76  }
77
78  //////////////////////////////////////
79  // Interface publique du module
80
81  /**
82   * Création d'un objet contenant les données et méthodes publiques
83   * (les propriétés publiques sont retournées par la fonction "module").
84   */
85  var publicInterfaceInstance = {
86
87      /**
88       * Retourne le module avec les méthodes "statiques"
89       * (comme l'accès direct à la liste des propriétés ou les tests regex)
90       * @return {Object} le module myApp.metier.adresse
91       */
92      getModule : function(){
93          return myApp.metier.adresse;
94      },
95
96      /**
97       * Accesseur pour tous les membres privés d'instance.
98       * @param {string} propertyName - nom de la propriété attendue d'une
          instance
99       * @return {string} la valeur de la propriété ou undefined en cas de nom de
          propriété inconnu.
100      */

```

```

101     getProperty : function(propertyName){
102         return adresse[propertyName];
103     },
104
105     /**
106     * Setter : initialise la valeur pour une propriété attendues d'une instance
107     * après un test.
108     * En cas d'erreur, un message pour cette propriété est est ajouté dans
109     * dataError.
110     * En l'absence d'erreur, une éventuelle erreur précédente est effacée.
111     * @param {string} propertyName – nom de la propriété attendue d'une
112     * instance
113     * @param {string} value – valeur à prendre pour la propriété attendu d'une
114     * instance
115     * @return {boolean} true s'il y a au moins une erreur, false sinon
116     */
117     setProperty : setPropertyOrError ,
118
119     /**
120     * @return {boolean} true s'il y a (au moins) une erreur, false sinon
121     */
122     hasError : function() {
123         if (dataError === false) {
124             return false;
125         }
126         for (var propertyName in dataError) {
127             if (dataError.hasOwnProperty(propertyName)) {
128                 return true;
129             }
130         }
131         return false;
132     },
133
134     /**
135     * Donne l'accès aux différents messages d'erreur.
136     * @param {string} propertyName – nom de propriété d'une instance d'adresse
137     * @return {string|undefined} le message d'erreur pour une propriété s'il
138     * existe ou undefined en l'absence d'erreur
139     */
140     getErrorMessage : function(propertyName){
141         return dataError[propertyName];
142     },
143
144     /**
145     * Récupère la liste des champs qui ont une erreur
146     * @return {string[]} tableau des noms de propriétés qui comportent une
147     * erreur.
148     */
149     getErrorList : function() {
150         var errorList = [];
151         for (var propertyName in dataError) {
152             if (dataError.hasOwnProperty(propertyName)) {
153                 errorList.push(propertyName);
154             }
155         }
156         return errorList;
157     }
158 }

```

```

151     }
152
153     }; // fin de publicInterfaceInstance
154
155     return publicInterfaceInstance;
156
157     } // fin de la méthode createInstance
158   }); // fin de l'appel "apply" de la méthode myApp.addModule

```

Le fichier *HTML* réalise le test de création d'une instance et d'utilisation de *setters*, et affiche les données et les erreurs obtenues.

<p>Données : Identifiant : 04abf85bc9 Numéro : 2 bis rue/place : Rue de l'a Paix Lieu dit, Bâtiment, BP : undefined Code Postal : 630000 Ville : Clermont-Ferrand Pays : France 2</p>	<p>Erreurs : Identifiant : undefined Numéro : undefined rue/place : undefined Lieu dit, Bâtiment, BP : Propriété undefined invalide : le complément d'adresse ne contient que des lettres, tirets et guillemets ou chiffres. Code Postal : Propriété 630000 invalide : Le code postal doit comporter 5 chiffres décimaux. Ville : undefined Pays: Propriété France 2 invalide : Le pays, obligatoire, ne contient que des lettres, tirets et guillemets.</p>
<p>Données : Identifiant : 04abf85bc9 Numéro : @#*m rue/place : Rue de l'a Paix Lieu dit, Bâtiment, BP : "Bâtiment 3D" Code Postal : 63000 Ville : Clermont-Ferrand Pays : France</p>	<p>Erreurs : Identifiant : undefined Numéro : Propriété @#*m invalide : Le numéro de la rue contient au plus 15 caractères, lettres, tirets et guillemets ou chiffres. rue/place : undefined Lieu dit, Bâtiment, BP : undefined Code Postal : undefined Ville : undefined Pays: undefined</p>

exemples/objet/ex06_moduleMetierAdresse.html

```

1  <!doctype HTML>
2  <html lang="fr">
3  <head>
4    <meta charset="UTF-8" />
5    <title>Création d'Instances d'Adresses</title>
6    <link rel="stylesheet", href="ex06_moduleMetierAdresse.css"/>
7  </head>
8  <body>
9    <!-- Création de l'application vide avec deux méthodes -->
10   <script src="ex04_structureApplication.js"></script>
11   <!-- Création de sous-module regexUtil de myApp.metier -->
12   <script src="./ex05_modulePatternRegex.js"></script>
13   <!-- Création de sous-module adresse de myApp.metier -->
14   <script src="./ex06_moduleMetierAdresse.js"></script>
15   <!-- Création d'une méthode fabrique d'adresse de myApp.metier.adresse -->
16   <script src="./ex06_fabriqueAdresse.js"></script>
17
18   <!-- Ajout d'un main et exécution -->
19   <script>
20

```

```

21 var testAfficheAdresse = function(adresse){
22     document.write("<div>");
23     document.write("<span><h2>Données </h2>" +
24         "<strong>" + myApp.metier.adresse.getLabelText('id') +
25         "</strong>" + adresse.getProperty('id') + "<br/>" +
26         "<strong>" + myApp.metier.adresse.getLabelText('numeroRue') +
27         "</strong>" + adresse.getProperty('numeroRue') + "<br/>" +
28         "<strong>" + myApp.metier.adresse.getLabelText('rue') +
29         "</strong>" + adresse.getProperty('rue') + "<br/>" +
30         "<strong>" + myApp.metier.adresse.getLabelText('complementAdresse') +
31         "</strong>" + adresse.getProperty('complementAdresse') + "<br/>" +
32         "<strong>" + myApp.metier.adresse.getLabelText('codePostal') +
33         "</strong>" + adresse.getProperty('codePostal') + "<br/>" +
34         "<strong>" + myApp.metier.adresse.getLabelText('ville') +
35         "</strong>" + adresse.getProperty('ville') + "<br/>" +
36         "<strong>" + myApp.metier.adresse.getLabelText('pays') +
37         "</strong>" + adresse.getProperty('pays') +
38         "</span>");
39
40     // variante en énumérant automatiquement les propriétés
41     var htmlCode = "<span><h2>Erreurs </h2>";
42     for (var i=0 ; i < myApp.metier.adresse.getPropertyList().length ; ++i){
43         var propertyName = myApp.metier.adresse.getPropertyList()[i];
44         htmlCode += "<strong>" + myApp.metier.adresse.getLabelText(propertyName)
45             + " </strong>" +
46             adresse.getErrorMessage(propertyName) + "<br/>";
47     }
48     htmlCode += "</span>";
49     document.write(htmlCode);
50     document.write("</div>");
51 };
52
53 // Ajout d'une méthode mainFunction
54 myApp.addModule("mainFunction", function(){
55     // création d'une instance
56     var adresse = myApp.metier.adresse.createInstance({
57         id: "04abf85bc9",
58         numeroRue: "2 bis",
59         rue: "Rue de l'a Paix",
60         // oublié du champs complementAdresse
61         codePostal: "63000",
62         ville: "Clermont-Ferrand",
63         pays: "France 2"
64     });
65
66     testAfficheAdresse(adresse);
67
68     adresse.setProperty("complementAdresse", "\Bâtiment 3D\");
69     adresse.setProperty("codePostal", "63000");
70     adresse.setProperty("pays", "France");
71     adresse.setProperty("numeroRue", "@#*m");
72
73     testAfficheAdresse(adresse);
74 });
75

```

```

76      // Exécution de la méthode mainFunction
77      myApp.mainFunction();
78    </script>
79  </body>
80 </html>

```

2.7 Création d'un Module `myApp.view.adresse`

Nous ajoutons, dans un module `myApp.view.adresse`, des méthodes pour générer le code *HTML* d'une adresse, au format compact (sur une ligne) ou au format développé (avec le détail des labels des attribut). Nous ajoutons enfin une méthode pour générer le code d'un formulaire d'adresse, avec affichage des erreurs sur la forme des champs saisis.

exemples/objet/ex07_adresseView.js

```

1  // Création d'un module myApp.view et d'un sous-module myApp.view.adresse
2  myApp.addModule("view", {adresse: {}});
3
4  /**
5   * Méthode de génération de code HTML pour une instance d'adresse.
6   * Pour chaque propriété attendue d'une adresse, la description de la propriété
7   *   et sa valeur sont affichées.
8   *
9   * @method getHtmlDevelopped
10  * @augments myApp.view.adresse
11  * @param {Object} adresse - spécification des propriétés d'une instance d'
12  *   adresse
13  * @param {string} adresse.id - identifiant unique de l'instance
14  * @param {string} adresse.numeroRue - numero de rue
15  * @param {string} adresse.rue - nom de rue
16  * @param {string} adresse.complementAdresse - complément d'adresse (lieu dut, b
17  *   âtiment, résidence, etc.)
18  * @param {string} adresse.codePostal - code postal
19  * @param {string} adresse.ville - nom de ville
20  * @param {string} adresse.pays - nom de pays
21  */
22  myApp.addModule.apply(myApp.view.adresse, ["getHtmlDevelopped", function(adresse
23  ){
24    var htmlCode = "";
25
26    var moduleAdresse = myApp.metier.adresse;
27
28    if (adresse.getProperty('numeroRue')){
29      htmlCode += "<span class=\"adresseItem\">" + moduleAdresse.getLabelText('
30        numeroRue') + "Énbsp;" + moduleAdresse.getLabelText('
31        numeroRue') + "</span>" +
32      moduleAdresse.getLabelText('numeroRue') + "<br/>";
33    }
34
35    htmlCode += "<span class=\"adresseItem\">" + moduleAdresse.getLabelText('rue')
36      + "Énbsp;" + moduleAdresse.getLabelText('rue') + "</span>" +
37      moduleAdresse.getLabelText('rue') + "<br/>";
38
39    if (typeof adresse.getProperty('complementAdresse') === "string" &&
40      adresse.getProperty('complementAdresse') !== ""){

```

```

34     htmlCode += "<span class=\"adresseItem\">" + moduleAdresse.getLabelText( '
        complementAdresse ') + "Énbsp; </span> " +
35     adresse.getProperty( 'complementAdresse ') + "<br />";
36 }
37
38 htmlCode += "<span class=\"adresseItem\">" + moduleAdresse.getLabelText( '
        codePostal ') + "Énbsp; </span> " +
39     adresse.getProperty( 'codePostal ') + "<br />" +
40     "<span class=\"adresseItem\">" + moduleAdresse.getLabelText( 'ville ') + "É
        nbsp; </span> " +
41     adresse.getProperty( 'ville ') + "<br />" +
42     "<span class=\"adresseItem\">" + moduleAdresse.getLabelText( 'pays ') + "É
        nbsp; </span> " +
43     adresse.getProperty( 'pays ') + "<br />";
44
45 if (adresse.hasError()){
46     var errorList = adresse.getErrorList();
47     htmlCode += "<strong>Certains champs ont une erreur </strong><br />";
48     for (var i=0 ; i<errorList.length ; i++){
49         if (i > 0){
50             htmlCode += ", ";
51         }
52         htmlCode += errorList[i];
53     }
54 }
55
56 return htmlCode;
57 }]);
58
59 /**
60  * Méthode de génération de code HTML pour une instance d'adresse.
61  * L'adresse est affichée sur une ligne, sans mention des erreurs.
62  *
63  * @method getHtmlDevelopped
64  * @augments myApp.view.adresse
65  * @param {Object} adresse - spécification des propriétés d'une instance d'
        adresse
66  * @param {string} adresse.id - identifiant unique de l'instance
67  * @param {string} adresse.numeroRue - numero de rue
68  * @param {string} adresse.rue - nom de rue
69  * @param {string} adresse.complementAdresse - complément d'adresse (lieu dut, b
        âtiment, résidence, etc.)
70  * @param {string} adresse.codePostal - code postal
71  * @param {string} adresse.ville - nom de ville
72  * @param {string} adresse.pays - nom de pays
73  */
74 myApp.addModule.apply(myApp.view.adresse, ["getHtmlCompact", function(adresse){
75     var htmlCode = "";
76
77     if (adresse.getProperty( 'numeroRue ')){
78         htmlCode += adresse.getProperty( 'numeroRue ') + ", ";
79     }
80
81     htmlCode += adresse.getProperty( 'rue ') + ", ";
82     if (adresse.getProperty( 'complementAdresse ')){
83         htmlCode += adresse.getProperty( 'complementAdresse ') + ", ";

```

```

84 }
85 htmlCode += adresse.getProperty( 'codePostal' ) + " " +
86     adresse.getProperty( 'ville' ) + ", " +
87     adresse.getProperty( 'pays' );
88 return htmlCode;
89 }});

```

Affichage d'Adresses

Numéro: 2 bis
 rue/place: Rue de l'a Paix
 Lieu dit, Bâtiment, BP:
 Code Postal: 630000
 Ville: Clermont-Ferrand
 Pays: France 2

Numéro : @#*m,
 rue/place : Rue de l'a Paix,
 Code Postal : 630000
 Ville : Clermont-Ferrand
 Pays : France 2

Certains champs ont une erreur :
 complementAdresse,
 codePostal, pays

Numéro : @#*m,
 rue/place : Rue de l'a Paix,
 Lieu dit, Bâtiment, BP :
 "Bâtiment 3D",
 Code Postal : 630000
 Ville : Clermont-Ferrand
 Pays : France

Certains champs ont une erreur :
 numeroRue

Valider

exemples/objet/ex07_adresseView.html

```

1  <!doctype HTML>
2  <html lang="fr">
3  <head>
4      <meta charset="UTF-8" />
5      <title>Affichage d'Adresses</title>
6      <link rel="stylesheet" href="ex07_adresseView.css"/>
7  </head>
8  <body>
9      <!-- Création de l'application vide avec deux méthodes -->
10     <script src="ex04_structureApplication.js"></script>
11     <!-- Création de sous-module regexUtil de myApp.metier -->
12     <script src="./ex05_modulePatternRegex.js"></script>
13     <!-- Création de sous-module adresse de myApp.metier -->
14     <script src="./ex06_moduleMetierAdresse.js"></script>
15     <!-- Création d'une méthode fabrique d'adresse de myApp.metier.adresse -->
16     <script src="./ex06_fabriqueAdresse.js"></script>
17     <!-- Création de fonctions d'affichage dans myApp.metier.adresse -->
18     <script src="./ex07_adresseView.js"></script>
19
20     <!-- Ajout d'un main et exécution -->
21     <script>
22
23         var testAfficheAdresse = function(adresse){
24             document.write( "<span style=\|\"width:260px; display: inline-block;
25                             vertical-align: top;\|\">" +
26                             myApp.view.adresse.getHtmlDevelopped(adresse)+ "<br/>" +
27                             myApp.view.adresse.getHtmlCompact(adresse) +
28                             "</span>" );
29
30         };
31
32         // Ajout d'une méthode mainFunction
33         myApp.addModule( "mainFunction", function() {
34             // création d'une instance

```



```

34     var adresse = myApp.metier.adresse.createInstance({
35         id : "04abf85bc9",
36         numeroRue : "2 bis",
37         rue : "Rue de l'a Paix",
38         // oubli du champs complementAdresse
39         codePostal : "63000",
40         ville : "Clermont-Ferrand",
41         pays : "France 2"
42     });
43
44     testAfficheAdresse(adresse);
45
46     adresse.setProperty("complementAdresse", "\Bâtiment 3D\");
47     adresse.setProperty("codePostal", "63000");
48     adresse.setProperty("pays", "France");
49     adresse.setProperty("numeroRue", "@#*m");
50
51     testAfficheAdresse(adresse);
52 });
53
54 // Exécution de la méthode mainFunction
55 myApp.mainFunction();
56 </script>
57 <script src="jquery.js"></script>
58 </body>
59 </html>

```

Chapitre 3

Constructeurs, Prototype et *Patterns* Associés

3.1 Constructeurs

Un *classe* en *Javascript* se crée à partir d'un constructeur, qui est une fonction dont le nom est le nom de la classe à créer. À l'intérieur du constructeur, les propriétés de la classe sont créées et initialisées à l'aide de l'identificateur **this**. Le constructeur retourne un unique objet dont les propriétés correspondent à celles qui ont été initialisées à l'aide de l'identificateur **this**. En d'autres termes, le constructeur retourne une instance de la classe. Par convention, **les noms de constructeurs commencent par une majuscule**.

exemples/objetPrototype/ex01_classeTelephone.js

```
1  /**
2   * Constructeur de téléphone. Notez la majuscule sur le nom.
3   * @constructor
4   * @param {string} tel1 - le numéro de téléphone.
5   * @param {string} [tel2] - un second numéro de téléphone.
6   */
7  var Telephone = function(tel1, /* argument optionnel */ tel2){
8
9      var checkPhone = function(tel){
10         // Test d'expression régulière après suppression des espaces et tabulations
11         if (typeof tel.libelle !== "string" || typeof tel.numero !== "string" ||
12             /^(\\+33|0)[0-9]{9}$/.test(tel.numero.replace(/\\s/g, '')) !== true){
13             throw {
14                 name: "IllegalArgumentException",
15                 message: "Numéro de téléphone \"\" + tel.libelle + \" : \" + tel.numero + \"
16                 \\\" invalide \"
17             }
18         };
19
20         checkPhone(tel1);
21         // Création d'un attribut de la classe
22         this.tel1=tel1;
23
24         if (tel2 !== undefined){
25             checkPhone(tel2);
26             // Création d'un attribut de la classe
```

```

27     this.tel2=tel2;
28 }
29
30 /**
31  * @method getHtml
32  * @return {string} le code HTML pour afficher une instance.
33  */
34 this.getHtml = function(){
35     var htmlCode = this.tel1.libelle + " : " + this.tel1.numero + "<br/>";
36     if (this.tel2 !== undefined){
37         htmlCode += this.tel2.libelle + " : " + this.tel2.numero + "<br/>";
38     }
39     return htmlCode;
40 };
41 }

```

exemples/objetPrototype/ex01_classeTelephone.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4     <meta charset="UTF-8" />
5     <title>Classes</title>
6     <script src="./ex01_classeTelephone.js"></script>
7 </head>
8 <body>
9     <script>
10         try{
11             // Appel du constructeur avec le mot cl  "new" :
12             var tel = new Telephone({ libelle : "Maison", numero : "+33 1 23 45 67 89"},
13                                     { libelle : "Mobile", numero : "09 87 65 43 21"});
14             // Utilisation de la m thode getHtml()
15             document.write("<p>" + tel.getHtml() + "</p>");
16         }catch (err){
17             alert(err.message);
18         }
19     </script>
20 </body>
21 </html>

```



Un constructeur doit syst matiquement  tre employ  avec le mot cl  **new**. En effet, l'emploi d'un constructeur sans le mot cl  **new**, qui ne g n re, en soi, aucune exception ni *warning* conduit   un comportement impr visible, g n ralement catastrophique. D'o  l'importance de **respecter la convention que les noms de constructeurs commencent par une majuscule**, contrairement   toutes les autres fonctions ou variables.

3.2 Prototypes

3.2.1 Notion de prototype

Les m thodes de classes telles que vues jusqu'  pr sent ont l'inconv nient que ces m thodes sont des propri t s des objets, qui existent en autant d'exemplaires qu'il y a d'instance des

objets, alors qu'elles sont constantes.

Pour éviter cela, on peut mettre les méthodes non pas directement dans l'objet, mais dans son *prototype*. Le prototype est lui-même une propriété de l'objet, mais qui est partagée entre tous les objets de la classe (il s'agit d'une variable de classe). Toutes les variables de classes doivent être créées au niveau du prototype.

exemples/objetPrototype/ex02_prototype.js

```

1  /**
2  *
3  * Augmente la classe Telephone en ajoutant une méthode au prototype de
4  *   Telephone
5  * @method getHtmlByLibelle
6  * @return {string} le code HTML pour afficher une instance.
7  */
8  Telephone.prototype.getNumero = function(libelle){
9      if (this.tel1.libelle.toLowerCase() === libelle.toLowerCase()){
10         return this.tel1.numero;
11     }
12     if (this.tel2 !== undefined &&
13         this.tel2.libelle.toLowerCase() === libelle.toLowerCase()){
14         return this.tel2.numero;
15     }
16     return "Numéro inexistant";
17 };

```

exemples/objetPrototype/ex02_prototype.html

```

1  <!doctype HTML>
2  <html lang="fr">
3  <head>
4      <meta charset="UTF-8" />
5      <title>Prototypes</title>
6      <!-- Inclusion de la définition de la classe Telephone -->
7      <script src="./ex01_classeTelephone.js"></script>
8      <script src="./ex02_prototype.js"></script>
9  </head>
10 <body>
11     <script>
12         try{
13             // Appel du constructeur avec le mot clé "new" :
14             var tel = new Telephone({ libelle : "Maison", numero : "+33 1 23 45 67 89"},
15                                     { libelle : "Mobile", numero : "09 87 65 43 21"});
16             // Utilisation de la méthode getNumero() du prototype
17             document.write("<p>" + tel.getNumero("maison") + "</p>");
18             document.write("<p>" + tel.getNumero("mobile") + "</p>");
19             document.write("<p>" + tel.getNumero("travail") + "</p>");
20         }catch (err){
21             alert(err.message);
22         }
23     </script>
24 </body>
25 </html>

```

La méthode `Object.prototype.hasOwnProperty()` permet de tester si une propriété d'un objet existe au niveau de l'objet lui-même, ou au niveau de son prototype, ou encore du

prototype de son prototype.

Le fonctionnement de la notion de prototype est le suivant : Lors d'une tentative d'accès à une propriété de l'objet, la propriété est tout d'abord recherchée au niveau des propriétés propres. Seulement si la propriété n'existe pas dans les propriétés propres, elle est ensuite recherchée dans le prototype de l'objet. Si elle n'existe pas non plus à ce niveau, la propriété est recherchée dans le prototype du prototype, et ainsi de suite...

Ce processus s'appelle la *délégation* et il permet de spécialiser les objets, en les faisant hériter des propriétés d'un prototype, tout en leur permettant de surcharger (redéfinir) les données ou méthodes. Ceci constitue un mécanisme très souple d'héritage, entièrement dynamique.

3.2.2 Surcharge des méthodes du prototype : l'exemple de toString

La méthode `toString`, qui permet de convertir un objet en chaîne de caractères (par exemple pour l'afficher) a une implémentation par défaut définie dans le prototype de la classe `Object`. On peut la surcharger dans le prototype de notre classe `Telephone` pour changer le comportement par défaut de la méthode `toString` et mettre en forme à notre guise les numéros de téléphone.

exemples/objetPrototype/ex03_toString.js

```

1  /**
2   * @override toString
3   * @return {string} une chaîne de caractère représentant l'instance de
4   *   Telephone
5   */
6  Telephone.prototype.toString = function() {
7      var texte = this.tel1.libelle + " : " + this.tel1.numero;
8      if (this.tel2 !== undefined) {
9          texte += " et " + this.tel2.libelle + " : " + this.tel2.numero;
10     }
11     return texte;
12 }
```

exemples/objetPrototype/ex03_toString.html

```

1  <!doctype HTML>
2  <html lang="fr">
3  <head>
4      <meta charset="UTF-8" />
5      <title>Surcharge des propriétés du prototype</title>
6      <!-- Inclusion de la définition de la classe Telephone -->
7      <script src="/ex01_classeTelephone.js"></script>
8      <script src="/ex03_toString.js"></script>
9  </head>
10 <body>
11     <script>
12         try {
13             // Appel du constructeur avec le mot clé "new" :
14             var tel = new Telephone({ libelle : "Maison", numero : "+33 1 23 45 67 89"},
15                                     { libelle : "Mobile", numero : "09 87 65 43 21"});
16             // Utilisation implicite de la méthode toString() (conversion)
17             document.write("<p>" + tel + "</p>");
18         } catch (err) {
```

```

19     alert (err . message );
20   }
21 </script>
22 </body>
23 </html>

```

3.3 Exemple : assurer l'implémentation d'interfaces

Voici une classe `Interface`, qui possède comme attribut un `array` de noms de méthodes, et qui permet de vérifier qu'un objet possède bien des méthodes avec les noms correspondants. Notons que nous ne vérifions pas que les méthodes correspondent bien à un prototype déterminé, mais seulement que les noms de méthodes sont présents.

exemples/objetPrototype/ex04_interfaceImplementation.js

```

1  /**
2   * Définit une "interface", avec un nom et un ensemble de méthodes.
3   * Ceci nous permettra de vérifier qu'un certain nombre d'opérations
4   * sont présentes dans un objet JavaScript.
5   *
6   * @constructor Interface
7   * @param {string} name - nom de l'interface
8   * @param {string[]} methods - tableau contenant les noms des méthodes de l'
9   *   interface.
10  */
11  var Interface = function (methods) {
12    if (methods.length === undefined) {
13      throw {
14        name: "IllegalArgument",
15        message: "Une interface nécessite un array (ou array-like) de noms de méthodes."
16      };
17    }
18
19    // Création d'une propriété pour stocker le nom de l'interface
20    // Création d'un tableau pour stocker les noms de méthodes
21    this.methods = [];
22    // pour chaque nom de méthode
23    for (var i = 0 ; i < methods.length; ++i) {
24      // Vérification de type
25      if (typeof methods[i] !== 'string') {
26        throw {
27          name: "IllegalArgument",
28          message: "Les noms de méthodes d'une interface doivent être de type string."
29        };
30      }
31      // Ajout du nom de méthode
32      this.methods.push(methods[i]);
33    }
34  };
35
36  /**

```

```

37  * Vérifie qu'un objet "implémente une interface", en ce sens qu'il comporte
38  * un certain nombre de méthodes (propriétés de type fonction) qui ont les
39  * mêmes noms que les méthodes de l'interface.
40  *
41  * @method isImplementedBy
42  * @param {Object} objet - objet qui doit implémenter l'interface.
43  * @return {boolean|string} true si l'objet comporte toutes les méthodes de l'
    interface,
44  * un message d'erreur indiquant une méthode qui n'est pas présente dans l'objet
    sinon.
45  */
46  Interface.prototype.isImplementedBy = function(objet) {
47    // Pour chaque nom de méthode
48    for (var i = 0 ; i < this.methods.length ; ++i) {
49      var methodName = this.methods[i];
50      // Si l'objet n'a pas de propriété de ce nom qui soit de type fonction
51      if (!objet[methodName] || typeof objet[methodName] !== 'function') {
52        return "L'objet n'implémente pas la méthode " + methodName;
53      }
54    }
55    return true;
56  };

```

Voici un exemple dans lequel nous définissons deux interfaces attendues de nos modules métier :

1. L'interface attendue d'un module permet de tester la présence d'un certain nombre de méthodes statiques ;
2. L'interface attendue des instances permet de s'assurer de la présence d'un certain nombre de méthodes sur les instances.

Remarque. Dans nos exemples, une méthode `getModule` permet d'obtenir le module correspondant à une instance (par exemple `myApp.metier.adresse`) à partir d'une instance d'adresse obtenue par la fabrique `createInstance` de ce même module). Nous verrons dans la partie 3.4 comment simplifier notre interface des modules métier en supprimant le besoin de cette méthode `getModule`.

exemples/objetPrototype/ex04_interfaceImplementation.html

```

1  <!doctype HTML>
2  <html lang="fr">
3  <head>
4    <meta charset="UTF-8" />
5    <title>Implémentation d'interfaces</title>
6    <!-- Chargement de l'ensemble des modules métier -->
7    <script src="./modulesMetier.js"></script>
8    <!-- Classe de vérification de l'implémentation d'interfaces -->
9    <script src="./ex04_interfaceImplementation.js"></script>
10 </head>
11 <body>
12   <script>
13
14     // Ajout d'une méthode mainFunction

```

```

15 myApp.addModule( "mainFunction", function() {
16
17     // Définition de l'interface commune aux modules métier (adresse, personne
18     // , etc.)
19     var metierCommonMethods = new Interface(
20     [ "getPropertyList", "getLabelText", "testRegex", "createInstance" ] );
21     // Définition de l'interface commune aux instances des modules métier (
22     // adresse, personne, etc.)
23     var metierCommonInstanceMethods = new Interface(
24     [ "getModule", "getProperty", "setProperty", "hasError", "getErrorMessage
25     ", "getErrorList" ] );
26
27     // création d'une instance
28     var monObjet = myApp.metier.adresse.createInstance({
29     id : "04abf85bc9",
30     numeroRue : "2 bis",
31     rue : "Rue de l'a Paix",
32     complementAdresse : "Bâtiment 3D",
33     codePostal : "63000",
34     ville : "Clermont-Ferrand",
35     pays : "France"
36     });
37
38     var testInstanceInterface = metierCommonInstanceMethods.isImplementedBy(
39     monObjet);
40     if (testInstanceInterface !== true){
41         document.write("<p>" + testInstanceInterface + "</p>");
42     }else{
43         var module = monObjet.getModule();
44         var testModuleInterface = metierCommonMethods.isImplementedBy(module);
45         if (testModuleInterface !== true){
46             document.write("<p>" + testModuleInterface + "</p>");
47         }else{
48             document.write("<p>L'objet semble bien implémenter les méthodes
49             requises.</p>");
50         }
51     }
52 }
53 });
54
55 // Exécution de la méthode mainFunction
56 myApp.mainFunction();
57 </script>
58 </body>
59 </html>

```

3.4 Fabrique d'Objets Métier avec prototype

Le but de cette partie est de redéfinir la babrique d'instances d'adresses (méthode `myApp.metier.adresse.createInstance` de la partie 2.6) pour que les méthodes du module (méthodes `getPropertyList`, `getLabelText`, `testRegex`, `createInstance`) soient accessibles au niveau des instances (comme méthode dans le prototype des instances). Les avantages de cette implémentation sont les suivants :

- Interface plus simple dans laquelle la méthode `getModule()` des instances, qui retournait

l'objet `myApp.metier.adresse` a été supprimée, puisque l'objet `myApp.metier.adresse` constituera désormais le prototype des instances créées par notre fabrique. Les méthodes du module seront ainsi accessibles de manière transparentes via l'instance.

- Le code source des méthodes d'instance (`getProperty`, `setProperty`, `hasError`, `getErrorMessage`, `getErrorList`) n'est compilé qu'une seule fois, car il se trouve comme variable locale privée d'un module, est ensuite exposé via de simples alias dans l'interface des instances.

Nous avons aussi ajouté la possibilité, en passant un argument `inputObj` égal à `null`, de réer une instance par défaut (`id` aléatoire et autres attributs vides) Ceci permet par exemple d'initialiser un formulaire vide pour créer une nouvelle instance.

exemples/objetPrototype/ex05_fabriqueAdressePrototype.js

```

1  /**
2   * Fabrique qui crée des objets représentant des adresse, suivant le "pattern
   * module".
3   * Le paramètre spec de notre fonction est un objet contenant les propriétés d'
   * une adresse à créer.
4   *
5   * @method createInstance
6   * @augments myApp.metier.adresse
7   * @param {Object|null} inputObj - spécification des propriétés d'une instance d
   * 'adresse. Si inputObj est null, on crée une adresse par défaut (id alé
   * atoire, autres propriétés vide).
8   * @param {string} inputObj.id - identifiant unique de l'instance
9   * @param {string} inputObj.numeroRue - numero de rue
10  * @param {string} inputObj.rue - nom de rue
11  * @param {string} inputObj.complementAdresse - complément d'adresse (lieu dut,
   * bâtiment, résidence, etc.)
12  * @param {string} inputObj.codePostal - code postal
13  * @param {string} inputObj.ville - nom de ville
14  * @param {string} inputObj.pays - nom de pays
15  */
16  myApp.addModule.apply(myApp.metier.adresse, ["createInstance", function() {
17
18      //////////////////////////////////////
19      // Interface publique du module
20
21      /**
22       * Création d'un constructeur privé, créant une classe dont une instance
23       * privée contiendra les données de l'instance, rendue publique via l'
   * interface
24       * du module.
25       *
26       * L'utilisation d'un constructeur est de définir les méthode privées au
   * niveau du prototype.
27       */
28      var PrivateInstanceConstructor = function() {
29
30          /**
31           * Objet privé contenant les propriétés (id, rue, ...) de l'instance,
   * initialement vide
32           * @member
33           * @private
34           */

```

```

35     this.adresse = {};
36     /**
37      * Objet privé contenant les messages d'erreur associés aux propriétés
        attendues des instances.
38      * @member
39      * @private
40      */
41     this.dataError = false;
42
43 };
44
45 /**
46  * Ajoute une propriété (message d'erreur) dans dataError
47  * @method addError
48  * @private
49  */
50 PrivateInstanceConstructor.prototype.addError = function(propertyName, message
    ){
51     // si dataError n'existe pas, on le crée
52     if (this.dataError === false){
53         this.dataError = {};
54     }
55     // Ajout d'une propriété
56     this.dataError[propertyName] = message;
57 };
58
59 /**
60  * Setter : initialise la valeur pour une propriété attendues d'une instance.
61  * En cas d'erreur un message pour cette propriété est ajouté dans
        dataError.
62  * En l'absence d'erreur, une éventuelle erreur précédente est effacée.
63  * @method addError
64  * @private
65  */
66 PrivateInstanceConstructor.prototype.setPropertyOrError = function(
    propertyName, value){
67     var resultTestRegex = myApp.metier.adresse.testRegex(propertyName, value);
68     // On initialise la propriété de l'adresse
69     this.adresse[propertyName] = value;
70     // Si la validation par expression régulière est passée
71     if (resultTestRegex === true){
72         // On efface une vieille erreur éventuelle
73         delete this.dataError[propertyName];
74     }else{
75         // On initialise la propriété de l'objet des erreurs.
76         // avec le message d'erreur.
77         PrivateInstanceConstructor.prototype.addError.call(this, propertyName, "
            Propriété " + value +
78             " invalide : " + resultTestRegex);
79     }
80 };
81
82
83 /**
84  * @return {boolean} true s'il y a (au moins) une erreur, false sinon
85  */

```

```

86     PrivateInstanceConstructor.prototype.hasError = function(){
87         if (this.dataError === false){
88             return false;
89         }
90         for (var propertyName in this.dataError){
91             if (this.dataError.hasOwnProperty(propertyName)){
92                 return true;
93             }
94         }
95         return false;
96     };
97
98
99     /**
100     * Récupère la liste des champs qui ont une erreur
101     * @return {string[]} tableau des noms de propriétés qui comportent une
102     *       erreur.
103     */
104     PrivateInstanceConstructor.prototype.getErrorList = function(){
105         var errorList = [];
106         for (var propertyName in this.dataError){
107             if (this.dataError.hasOwnProperty(propertyName)){
108                 errorList.push(propertyName);
109             }
110         }
111         return errorList;
112     };
113
114     // Génération d'un ID aléatoire en cas de création d'une nouvelle adresse
115     // (cas où les specifications inputObj sont null)
116     var generateRandomId = function(){
117         var idLength = 10;
118         var resultat = "";
119         var hexaDigits = Array("0","1","2","3","4","5","6","7","8","9","a","b","c",
120                                "d","e","f");
121         var i;
122         for (i=0 ; i<10 ; ++i){
123             resultat += hexaDigits[Math.floor(Math.random()*16)];
124         }
125         return resultat;
126     };
127
128     /**
129     * Fabrication d'une instance à partir d'une inputObj de spécifications
130     * (voir documentation du module) et d'une instance privée vide.
131     */
132     var fabriqueInstance = function(inputObj, privateInstance){
133
134         /**
135         * Constructeur de l'instance qui sera effectivement retournée (instance
136         * publique),
137         * qui crée les données dans l'objet privateInstance.
138         * Ce constructeur contiendra aussi les méthodes d'instance publiques,
139         * et les méthodes (statiques) du module dans son prototype.
140         */
141         var PublicInstanceConstructor = function(){

```

```

139
140     var adresseMethods = myApp.metier.adresse; // raccourci
141
142     // Si l'objet en argument est null, on construit une instance par défaut
143     // (vide)
144     if (inputObj === null){
145         privateInstance.adresse = {
146             id : generateRandomId(),
147             numeroRue : "",
148             rue : "",
149             complementAdresse : "",
150             codePostal : "",
151             ville : "",
152             pays : ""
153         };
154     }else{
155         // Parcours des propriétés de getPropertyList()
156         // qui correspondent aux propriétés de l'adresse à créer
157         for (var i = 0 ; i < adresseMethods.getPropertyList().length ; ++i){
158             var propertyName = adresseMethods.getPropertyList()[i];
159             privateInstance.setPropertyOrError(propertyName, inputObj[
160                 propertyName]);
161         }
162     }
163
164     /**
165     * Accesseur pour tous les membres privés d'instance.
166     * @param {string} propertyName - nom de la propriété attendue d'une
167     * instance
168     * @return {string} la valeur de la propriété ou undefined en cas de nom
169     * de propriété inconnu.
170     */
171     this.getProperty = function(propertyName){
172         return privateInstance.adresse[propertyName];
173     };
174
175     /**
176     * Setter : initialise la valeur pour une propriété attendues d'une
177     * instance après un test.
178     * En cas d'erreur, un message pour cette propriété est ajouté dans
179     * dataError.
180     * En l'absence d'erreur, une éventuelle erreur précédente est effacée.
181     * @param {string} propertyName - nom de la propriété attendue d'une
182     * instance
183     * @param {string} value - valeur à prendre pour la propriété attendu d'
184     * une instance
185     * @return {boolean} true s'il y a au moins une erreur, false sinon
186     */
187     this.setProperty = function(propertyName, value){
188         return privateInstance.setPropertyOrError(propertyName, value);
189     };
190
191     /**
192     * @return {boolean} true s'il y a (au moins) une erreur, false sinon
193     */
194     this.hasError = function(){

```

```

187         return privateInstance.hasError();
188     };
189
190     /**
191      * Donne l'accès aux différents messages d'erreur.
192      * @param {string} propertyName – nom de propriété d'une instance d'
193      *   adresse
194      * @return {string|undefined} le message d'erreur pour une propriété s'
195      *   il existe ou undefined en l'absence d'erreur
196      */
197     this.getErrorMessage = function(propertyName){
198         return privateInstance.dataError[propertyName];
199     };
200
201     /**
202      * Récupère la liste des champs qui ont une erreur
203      * @return {string[]} tableau des noms de propriétés qui comportent une
204      *   erreur.
205      */
206     this.getErrorList = function(){
207         return privateInstance.getErrorList();
208     };
209     // fin du PublicInstanceConstructor
210
211     // MISE À DISPOSITION DES MÉTHODES DU MODULE VIA LE PROTOTYPE
212     PublicInstanceConstructor.prototype = myApp.metier.adresse;
213
214     return new PublicInstanceConstructor(inputObj);
215
216 }; // fin de la fonction fabriqueInstance
217
218 // Construction d'une instance avec
219 return function(inputObj){
220     // Un objet PrivateInstanceConstructor est construit pour l'occasion
221     return fabriqueInstance(inputObj, new PrivateInstanceConstructor());
222 };
223 }() // fin de la méthode createInstance
224 ]; // fin de l'appel "apply" de la méthode myApp.addModule

```

exemples/objetPrototype/ex05_fabriqueAdressePrototype.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4   <meta charset="UTF-8" />
5   <title>Création d'Instances d'Adresses</title>
6   <link rel="stylesheet", href="basicStyle.css"/>
7 </head>
8 <body>
9   <!-- Création de l'application vide avec deux méthodes -->
10  <script src="../objet/ex04_structureApplication.js"></script>
11  <!-- Création de sous-module regexUtil de myApp.metier -->
12  <script src="../objet/ex05_modulePatternRegex.js"></script>
13  <!-- Création de sous-module adresse de myApp.metier -->
14  <script src="../objet/ex06_moduleMetierAdresse.js"></script>
15  <!-- Création d'une méthode fabrique d'adresse de myApp.metier.adresse -->

```

```

16 <script src="./ex05_fabriqueAdressePrototype.js"></script>
17
18 <!-- Ajout d'un main et exécution -->
19 <script>
20
21     var testAfficheAdresse = function(adresse){
22         document.write("<div>");
23         document.write("<span<h2>Données </h2>" +
24             "<strong>" + adresse.getLabelText('id') +
25             "</strong>" + adresse.getProperty('id') + "<br/>" +
26             "<strong>" + adresse.getLabelText('numeroRue') +
27             "</strong>" + adresse.getProperty('numeroRue') + "<br/>" +
28             "<strong>" + adresse.getLabelText('rue') +
29             "</strong>" + adresse.getProperty('rue') + "<br/>" +
30             "<strong>" + adresse.getLabelText('complementAdresse') +
31             "</strong>" + adresse.getProperty('complementAdresse') + "<br/>" +
32             "<strong>" + adresse.getLabelText('codePostal') +
33             "</strong>" + adresse.getProperty('codePostal') + "<br/>" +
34             "<strong>" + adresse.getLabelText('ville') +
35             "</strong>" + adresse.getProperty('ville') + "<br/>" +
36             "<strong>" + adresse.getLabelText('pays') +
37             "</strong>" + adresse.getProperty('pays') +
38             "</span>");
39
40         // variante en énumérant automatiquement les propriétés
41         var htmlCode = "<span<h2>Erreurs </h2>";
42         for (var i=0 ; i < adresse.getPropertyList().length ; ++i){
43             var propertyName = adresse.getPropertyList()[i];
44             htmlCode += "<strong>" + adresse.getLabelText(propertyName) + " </s
45                 trong>" +
46                 adresse.getErrorMessage(propertyName) + "<br/>";
47         }
48
49         htmlCode += "</span>";
50         document.write(htmlCode);
51         document.write("</div>");
52     };
53
54     // Ajout d'une méthode mainFunction
55     myApp.addModule("mainFunction", function(){
56         // création d'une instance
57         var adresse = myApp.metier.adresse.createInstance({
58             id : "04abf85bc9",
59             numeroRue : "2 bis",
60             rue : "Rue de l'a Paix",
61             // oubli du champs complementAdresse
62             codePostal : "63000",
63             ville : "Clermont-Ferrand",
64             pays : "France 2"
65         });
66
67         testAfficheAdresse(adresse);
68
69         adresse.setProperty("complementAdresse", "\"Bâtiment 3D\"");
70         adresse.setProperty("codePostal", "63000");
71         adresse.setProperty("pays", "France");

```

```

71     adresse.setProperty("numeroRue", "@#*m");
72
73     testAfficheAdresse(adresse);
74 });
75
76     // Exécution de la méthode mainFunction
77     myApp.mainFunction();
78 </script>
79 </body>
80 </html>

```

En utilisant cette fabrique, le code la méthode `myApp.view.adresse.getHtmlDevelopped` est un peu différent car il faut supprimer les appels à la méthode `getModule`, en accédant directement au module via le prototype des instances.

En outre, l'interface des objets métiers se trouve simplifiée, car il n'y a plus qu'une seule interface, au lieu de deux dans l'exemple de la partie 3.3.

exemples/objetPrototype/ex07_interfaceImplementationPrototype.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4     <meta charset="UTF-8" />
5     <title>Implémentation d'interfaces</title>
6     <!-- Création de l'application vide avec deux méthodes -->
7     <script src="../objet/ex04_structureApplication.js"></script>
8     <!-- Création de sous-module regexUtil de myApp.metier -->
9     <script src="../objet/ex05_modulePatternRegex.js"></script>
10    <!-- Création de sous-module adresse de myApp.metier -->
11    <script src="../objet/ex06_moduleMetierAdresse.js"></script>
12    <!-- Création d'une méthode fabrique d'adresse de myApp.metier.adresse -->
13    <script src="../ex05_fabriqueAdressePrototype.js"></script>
14    <!-- Classe de vérification de l'implémentation d'interfaces -->
15    <script src="../ex04_interfaceImplementation.js"></script>
16 </head>
17 <body>
18     <script>
19
20         // Ajout d'une méthode mainFunction
21         myApp.addModule("mainFunction", function() {
22
23             // Définition de l'interface commune aux instances des modules métier (
24             // adresse, personne, etc.)
25             var metierCommonInstanceMethods = new Interface(
26                 ["getPropertyList", "getLabelText", "testRegex", "getProperty", "
27                 setProperty", "hasError", "getErrorMessage", "getErrorList"]);
28
29             // création d'une instance
30             var monObjet = myApp.metier.adresse.createInstance({
31                 id: "04abf85bc9",
32                 numeroRue: "2 bis",
33                 rue: "Rue de l'a Paix",
34                 complementAdresse: "Bâtiment 3D",
35                 codePostal: "63000",
36                 ville: "Clermont-Ferrand",
37                 pays: "France"
38             });

```

```

37
38     var testInstanceInterface = metierCommonInstanceMethods.isImplementedBy(
        monObjet);
39     if (testInstanceInterface !== true){
40         document.write("<p>" + testInstanceInterface + "</p>");
41     }else{
42         document.write("<p>L'objet semble bien implémenter les méthodes
            requises.</p>");
43     }
44 });
45
46 // Exécution de la méthode mainFunction
47 myApp.mainFunction();
48 </script>
49 </body>
50 </html>

```

3.5 Patterns *pseudo-classique* (à éviter)

Dans l'exemple suivant, nous créons une classe **Personne** qui hérite des propriétés de la classe **Adresse**. Pour cela :

1. le constructeur d'**Adresse** est appelé explicitement dans le constructeur de **Personne** ;
2. la classe **Adresse** est déclarée comme **superclass** de la classe **Personne** ;
3. Les méthodes qui existent au niveau du prototype de la classe **Adresse** et qui doivent être spécifiées pour des personnes sont surchargées au niveau du prototype de la classe **Personne**.

Dans l'exemple suivant, nous surchargeons l'accesseur de la propriété **ville** et la méthode **toString**.

exemples/vieux/objet.vieux/ex14_extension_de_classe.js

```

1  function Adresse(numeroRue, rue, complement, codePostal, ville) {
2      if (numeroRue.match(/^([0-9]*)([0-9]+)(\ ?)((bis)|(ter))?\$/)) {
3          this.numeroRue = numeroRue.replace(/\s+/g, ' ');
4      } else {
5          throw new Error("Numéro de la rue invalide.");
6      }
7
8      if (rue.match(/^(([a-zA-ZêéèöàöÊÊËÀÖË| | - |.|0-9| ])/(\ "))/(\ ')){1,300}\$/))
9          {
10         this.rue = rue.replace(/\s+/g, ' ');
11     };
12 } else {
13     throw new Error("Nom de la rue/place invalide.");
14 }
15
16 if (complement.match(/^(([a-zA-ZêéèöàöÊÊËÀÖË| | - |.|0-9| ])/(\ "))/(\ '))
17     {0,300}\$/)) {
18     this.complement = complement.replace(/\s+/g, ' ');
19 } else {

```



```

18     throw new Error("Complement d'adresse invalide.");
19 }
20
21 if (codePostal.match(/^([0-9]{5})$/)) {
22     this.codePostal = codePostal;
23 } else {
24     throw new Error("Code postal invalide.");
25 }
26
27 if (ville.match(/^(([a-zA-ZêéèöàöÉÊËÀÖË| |-,0-9| ])/(\ | ')){0,300}$/)
28 ) {
29     this.ville = ville.replace(/\s+/g, ' ');
30 } else {
31     throw new Error("Nom de ville invalide.");
32 }
33
34 Adresse.prototype.getVille = function() {
35     return this.ville;
36 }
37
38 Adresse.prototype.toString = function() {
39     var resultat = this.numeroRue;
40     if (this.numeroRue != "")
41         resultat += ", ";
42     resultat += this.rue + ", ";
43     resultat += this.complement;
44     if (this.complement != "")
45         resultat += ", ";
46     resultat += this.ville + "<br/>";
47     return resultat;
48 }
49
50 function Personne(nom, prenom, numeroRue, rue, complement, codePostal, ville){
51     Adresse.call(this, numeroRue, rue, complement, codePostal, ville);
52     this.nom = nom;
53     this.prenom = prenom;
54 }
55
56 Personne.superclass = Adresse;
57 Personne.prototype.getVille = function() {
58     return Adresse.prototype.getVille.call(this);
59 }
60
61 Personne.prototype.toString = function() {
62     return this.nom + ", " + this.prenom + ", " + Adresse.prototype.toString.call(this);
63 }

```

exemples/vieux/objet.vieux/ex14_extention_de_classe.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Chainage de constructeurs</title>
6 <script src="/ex14_extention_de_classe.js"></script>

```

```
7 </head>
8 <body>
9 <p>
10 <script>
11
12 try{
13     var pers = new Personne("Dujardin", "Jean", "10 ter", "rue de l'avenir", "Le
        Rastou", "86098", "Les Flots Bleus");
14     document.write(pers);
15     document.write("<br/>L'adresse se trouve dans la ville de \""+pers.getVille()+
        "\" \".");
16 }catch (err){
17     alert(err);
18 }
19 </script>
20 <p>
21 </body>
22 </html>
```

Chapitre 4

Interfaces Hommes Machines (*IHM*)

4.1 Filtrage Basique des Inputs d'un Formulaire

L'exemple suivant montre comment filtrer les attributs d'un formulaires côté client en affichant immédiatement un message d'erreur lors de la saisie d'une valeur incorrecte. On associe à chaque événement `onchange` de chaque attribut une fonction *JavaScript* qui réalisera le filtrage.

exemples/gui/ex01_basicForm.js

```
1 // alias vers le module d'expressions régulières
2 var regexUtil = myApp.metier.regexUtil;
3 /**
4  * Gestionnaire d'événement onchange de l'input d'ID "mainForm_titre".
5  * Cette méthode effectue le filtrage par exexpression régulière.
6  * @method filterTitre
7  */
8 var filterTitre = function(){
9     var titreValue = $("#mainForm_titre").val();
10    // Expressions du langage courant et chiffres
11    var resultRegexTest = regexUtil.testRegexLatin1WithDigits({
12        chaine : titreValue ,
13        minLength : 1
14    });
15    // Modification du contenu du span d'ID "error_mainForm_titre"
16    if (resultRegexTest === true){
17        $("#error_mainForm_titre").empty();
18    }else{
19        $("#error_mainForm_titre").html(
20            "Erreur : le titre ne doit contenir que les lettres et chiffres<br />");
21    }
22 };
23
24 /**
25  * Gestionnaire d'événement onchange de l'input d'ID "mainForm_resume".
26  * Cette méthode effectue le filtrage par exexpression régulière.
27  * @function filterTitre
28  */
29 var filterResume = function(){
30     var titreValue = $("#mainForm_resume").val();
31    // Expressions du langage courant et chiffres et ponctuation
32    var resultRegexTest = regexUtil.testRegexLatin1WithDigitsPunctuation({
33        chaine : titreValue ,
```

```

34     minLength : 1
35   });
36   // Modification du contenu du span d'ID "error_mainForm_resume"
37   if (resultRegexTest == true){
38     $("#error_mainForm_resume").empty();
39   }else{
40     $("#error_mainForm_resume").html(
41       "Erreur : le résumé ne doit contenir que les lettres et chiffres" +
42       " ou des caractères de ponctuation<br />");
43   }
44 };

```



exemples/gui//ex01_basicForm.html

```

1  <!doctype HTML>
2  <html lang="fr">
3  <head>
4  <meta charset="UTF-8" />
5  <title>Filtrage d'inputs</title>
6  <link rel="stylesheet" href="basicStyle.css"/>
7  </head>
8  <body>
9    <h1>Saisie d'un film</h1>
10   <form id="mainForm" action="post">
11
12     <!-- input avec gestionnaire de l'événement onchange -->
13     <span id="error_mainForm_titre" class="errorMsg"></span>
14     <label for="mainForm_titre">Titre :</label>
15     <input type="text" id="mainForm_titre" size="15"
16     placeholder="Titre du film" onchange="filterTitre()" /><br />
17
18     <!-- textarea avec gestionnaire de l'événement onchange -->

```

```

19     <span id="error_mainForm_resume" class="errorMsg"></span>
20     <label for="mainForm_resume">Résumé </label>
21     <textarea id="mainForm_resume" rows="10" cols="50"
22         placeholder="Saisissez votre résumé" onchange="filterResume()"></textarea>
23
24 </form>
25 <!-- Inclusion de la structure d'application et du module regexUtil -->
26 <script src="modulesMetier.js"></script>
27 <!-- Inclusion de jQuery pour les événements et manipulation du DOM -->
28 <script src="jquery.js"></script>
29 <script src="ex02_mediatorInputFilter.js"></script>
30 <script src="ex01_basicForm.js"></script>
31 </body>
32 </html>

```

4.2 Pattern Mediator pour le filtrage d'attributs

L'inconvénient du filtrage présenté dans la partie 4.1 est que, dans le code *HTML* d'un champs du formulaire lui-même, on doit déclarer une méthode de filtrage spécifique pour ce champs (attribut `onchange` de l'élément *HTML* `input` ou `textarea`).

Dans l'architecture d'application que nous proposons par la suite, la méthode de filtrage ne sera pas codée en dût dans le module chargé de générer le formulaire, mais plutôt dans les tests d'expressions régulières effectués dans les modules métier. En particulier, la méthode précise dépendra de l'instance et de la propriété considérée, ce qui entraînera un fort *couplage* (interdépendance) des méthodes chargées de l'*IHM* et des classes métier (ou du *modèle*).

Nous savons par expérience que ce type de couplage va provoquer des difficultés pour la maintenance et l'évolution de notre application (comme par exemple la migration de nos objet métier côté serveur avec *NodeJS*). Nous allons maintenant introduire un *pattern* qui a pour vocation de découpler le déclenchement des événements (via, en l'occurrence, des événements utilisateurs `onchange`) de l'implémentation des opérations correspondantes sur les données métier, ou les données du modèle. Ce pattern est une généralisation du pattern *Observer*.

Dans notre exemple, un module *Mediator* va enregistrer les méthodes *callbacks* (qui ne sont que des fonctions *JavaScript*) associées à des événements. L'exécution des ces callbacks (en l'occurrence la réaction à un événement `onchange`) sera déclenchée par la publication de l'événement en question par l'intermédiaire du *Mediator*.

exemples/gui/ex02_mediatorInputFilter.js

```

1
2 /**
3  * Ajout d'un module ctrl (contrôleurs) à l'application.
4  * @module ctrl
5  * @augments myApp
6  */
7 myApp.addModule.apply(myApp, [ "ctrl", {} ] );
8
9 /**
10  * Implémentation du pattern "Médiateur" pour gérer le filtrage des inputs de
11    formulaires.
12  * @module mediatorInputFilter
13  * @augments myApp.ctrl
14  */

```

```

14 myApp.addModule.apply(myApp.ctrl, ["mediatorInputFilter", function(){
15
16     //////////////////////////////////////
17     // Propriétés et méthodes "statiques" privées
18
19     /**
20      * Collection, indexée par ID de formulaire de callbacks d'événements liés à
21      * différents formulaires (typiquement : événement onchange d'un input).
22      * @private
23      */
24     var m_subscriptionLists;
25
26     /**
27      * Initialise (ou réinitialise) l'ensemble des listes d'événements à la
28      * collection vide.
29      * @private
30      */
31     var init = function(){
32         m_subscriptionLists = {};
33     };
34
35     // Initialiser une fois l'ensemble des listes d'événements à la collection
36     // vide.
37     init();
38
39     //////////////////////////////////////
40     // Interface publique du module
41
42     /**
43      * Création d'un objet contenant les données et méthodes publiques
44      * (les propriétés publiques sont retournées par la fonction "module").
45      */
46     var publicInterfaceMediator = {
47
48         /**
49          * Ajoute un formulaire et la liste (initialement vide) de ses callbacks
50          * associés.
51          * Si le formulaire est déjà géré, la liste de ses callbacks associés est
52          * supprimée et réinitialisée à la liste vide.
53          * @param {string} formId - l'Id du formulaire (en tant qu'élément HTML)
54          */
55         addForm : function(formId){
56             m_subscriptionLists[formId] = {};
57         },
58
59         /**
60          * Supprime un formulaire et ses callbacks associés
61          * @param {string} formId - l'Id du formulaire (en tant qu'élément HTML)
62          */
63         removeForm : function(formId){
64             if (!m_subscriptionLists.hasOwnProperty(formId)){
65                 return false;
66             }
67             delete m_subscriptionLists[formId];
68             return true;
69         },

```

```

65
66  /**
67   * Ajout d'un événement associé à un attribut de formulaire et de sa
        fonction callback.
68   * Si l'événement existait déjà pour cet input, il est écrasé.
69   * @param {string} formId - l'Id du formulaire (en tant qu'élément HTML)
70   * @param {string} inputName - le nom de l'input (ou de la propriété de l'
        objet métier associé).
71   * @param {function} callbackFunction - la fonction (callback) à appeler en
        cas de publication de l'événement.
72  */
73  subscribe : function(formId, inputName, callbackFunction){
74      if (m_subscriptionLists.hasOwnProperty(formId)){
75          m_subscriptionLists[formId][inputName] = {callback : callbackFunction};
76      }else{
77          throw {name : "IllegalArgumentException",
78                message : "Catégorie d'événements " + eventCateg + " inconnue du mé
                    diateur"}
79      };
80  }
81  },
82
83  /**
84   * Publication d'un événement associé à un attribut de formulaire provoquant
        l'exécution de la fonction callback associée
85   * @param {string} formId - l'Id du formulaire (en tant qu'élément HTML)
86   * @param {string} inputName - le nom de l'input (ou de la propriété de l'
        objet métier associé).
87  */
88  publish : function(formId, inputName){
89
90      if (m_subscriptionLists.hasOwnProperty(formId)){
91          if (m_subscriptionLists[formId].hasOwnProperty(inputName)){
92              // On appelle le callback avec son
93              m_subscriptionLists[formId][inputName].callback();
94          }
95      }else{
96          throw {name : "IllegalArgumentException",
97                message : "Formulaire d' ID " + formId + " inconnu du médiateur"}
98      };
99  }
100 },
101
102 /**
103  * Réinitialise la collection des formulaires gérés à une collection vide.
104  */
105 empty : function(){
106     init();
107 }
108 };
109
110 return publicInterfaceMediator;
111
112 }());

```

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Filtrage d'inputs</title>
6 <link rel="stylesheet" href="basicStyle.css"/>
7 </head>
8 <body>
9   <h1>Saisie d'un film</h1>
10  <form id="mainForm" action="post">
11
12    <!-- input avec gestionnaire de l'événement onchange -->
13    <span id="error_mainForm_titre" class="errorMsg"></span>
14    <label for="mainForm_titre">Titre </label>
15    <input type="text" id="mainForm_titre" size="15"
16    placeholder="Titre du film" onchange="filter Data( 'mainForm', 'titre' )"/>
17
18    <!-- textarea avec gestionnaire de l'événement onchange -->
19    <span id="error_mainForm_resume" class="errorMsg"></span>
20    <label for="mainForm_resume">Résumé </label>
21    <textarea id="mainForm_resume" rows="10" cols="50"
22    placeholder="Saisissez votre résumé" onchange="filter Data( 'mainForm', '
    resume' )"></textarea>
23
24  </form>
25  <!-- Inclusion de la structure d'application et du module regexUtil -->
26  <script src="modulesMetier.js"></script>
27  <!-- Inclusion de jQuery pour les événements et manipulation du DOM -->
28  <script src="jquery.js"></script>
29  <script src="ex02_mediatorInputFilter.js"></script>
30  <script src="ex01_basicForm.js"></script>
31  <script>
32    // Ajout du formulaire "mainForm" au médiateur qui gèrera ses événements.
33    myApp.ctrl.mediatorInputFilter.addForm( 'mainForm' );
34    // Enregistrement du callback associé à l'événement onchange du titre
35    myApp.ctrl.mediatorInputFilter.subscribe( 'mainForm', 'titre', filterTitre );
36    // Enregistrement du callback associé à l'événement onchange du résumé
37    myApp.ctrl.mediatorInputFilter.subscribe( 'mainForm', 'resume', filterResume )
38    ;
39    /**
40     * Publie l'événement onchange d'un input auprès du médiateur, provoquant l'
41     * exécution du callback enregistré pour cet événement.
42     * @function filterData
43     */
44    var filterData = function( formId, inputName ){
45      myApp.ctrl.mediatorInputFilter.publish( formId, inputName );
46    };
47  </script>
48 </body>
49 </html>

```


4.3 Exemple : génération automatique de formulaire d'adresse

4.3.1 Avec l'interface d'objets métier sans prototype

Dans l'exemple suivant, des méthodes d'un module `myApp.gui` permettent de générer automatiquement les inputs d'un formulaire permettant de saisir les propriétés (ici supposées de type texte) d'un objet qui implémente des interface qui apparaissent dans l'exemple de la partie 3.3.

Nous appliquons cette méthode pour afficher et filtrer automatiquement un formulaire de saisie d'une adresse.

exemples/gui/ex03_formsGui.js

```

1 // Ajout d'un module "gui" à notre application myApp
2 myApp.addModule.apply(myApp, ["gui", {}]);
3
4 /**
5  * Fonction de génération de l'ID d'un élément HTML de type input préfixé par l'
6  *   ID du formulaire
7  * @method myApp.gui.getInputId
8  * @param {Object} inputSpec contient les spécifications de l'input
9  * @param {string} inputSpec.formId id du formulaire dans lequel l'input sera
10  *   inséré
11  * @param {string} inputSpec.propertyName nom de la propriété de inputSpec.
12  *   objetMetier à saisir dans l'input
13 */
14 myApp.addModule.apply(myApp.gui, ["getInputId", function(inputSpec){
15   return inputSpec.formId + "_" + inputSpec.propertyName;
16 }]);
17
18 /**
19  * Publie auprès du Mediator un événement onchange d'un Input
20  * @param {string} formId id du formulaire dans lequel l'input sera inséré
21  * @param {string} propertyName nom de la propriété de inputSpec.objetMetier à
22  *   saisir dans l'input
23 */
24 myApp.addModule.apply(myApp.gui, ["publishInputChange", function(formId,
25   propertyName){
26   myApp.ctrl.mediatorInputFilter.publish(formId, propertyName);
27 }]);
28
29 /**
30  * Génération du code HTML d'un input.
31  * @method myApp.gui.getTextInputCode
32  * @param {Object} inputSpec contient les spécifications de l'input
33  * @param {Object} inputSpec.objetMetier instance d'un module métier (par exemple
34  *   instance d'adresse, de personne...).
35  *   cet objet doit implémenter des interfaces précises (
36  *     getProperty(), getModule(), etc.)
37  * @param {string} inputSpec.formId id du formulaire dans lequel l'input sera
38  *   inséré
39  * @param {string} inputSpec.propertyName nom de la propriété de inputSpec.
40  *   objetMetier à saisir dans l'input
41  * @param {string} [inputSpec.type=text] type de l'input
42  * @param {number} [inputSpec.inputSize=10] taille de l'input (nombre de
43  *   caractères)
44 */

```

```

35 myApp.addModule.apply(myApp.gui, ["getInputCode", function(inputSpec){
36     // Calcul de l'ID de l'input :
37     var inputId = myApp.gui.getInputId(inputSpec);
38
39     // Valeur de la propriété de l'objet pour l'attribut value de l'input
40     var propertyValue = inputSpec.objetMetier.getProperty(inputSpec.propertyName
41     ) || "";
42     // Création d'un éventuel message si l'objet comportait déjà une erreur
43     var errorMessage = inputSpec.objetMetier.getErrorMessage(inputSpec.
44     propertyName) !== undefined
45         ? inputSpec.objetMetier.getErrorMessage(inputSpec.propertyName)
46         + "<br/>" : "";
47
48     var moduleMetier = inputSpec.objetMetier.getModule(); // raccourci
49
50     // Callback de gestion du filtrage de l'input :
51     myApp.ctrl.mediatorInputFilter.subscribe(inputSpec.formId, inputSpec.
52     propertyName, function(){
53
54         // Si aucun test d'expression régulière n'est prévu
55         if (moduleMetier === undefined ||
56             moduleMetier.testRegex === undefined){
57             return true; // accepter la valeur
58         }
59
60         var resultatTestRegex = moduleMetier.testRegex(inputSpec.propertyName,
61             document.getElementById(inputId).value);
62         if (resultatTestRegex !== true){
63             document.getElementById("error_"+inputId).innerHTML =
64             resultatTestRegex + "<br/>";
65         }else{
66             document.getElementById("error_"+inputId).innerHTML = "";
67         }
68     }); // fin du callback //////////////////////////////////////
69
70     var inputType = inputSpec.inputType === undefined ? "text" : inputSpec.
71     inputType;
72     var inputSize = inputSpec.inputSize === undefined ? "15" : inputSpec.
73     inputSize;
74     var labelText = moduleMetier.getLabelText(inputSpec.propertyName);
75     // retour du code HTML de l'input
76     return "<span class=\"errorMsg\" id=\"error_"+inputId+"\">" + errorMessage +
77     "</span>" +
78     "<label for=\"" + inputSpec.propertyName + "\">" + labelText + "</label>"
79     +
80     "<input type=\"" + inputType + "\" name=\"" + inputSpec.
81     propertyName +
82     "\" id=\"" + inputId + "\" value=\"" + propertyValue + "\" " +
83     "size=\"" + inputSize + "\" " +
84     "onchange=\"myApp.gui.publishInputChange('"+inputSpec.formId+"',
85     '"+inputSpec.propertyName+"')\">";
86
87     });
88
89 /**
90  * Génération du code HTML de l'ensemble des inputs d'un formulaire.

```

```

80  * @method myApp.gui.getHtmlFormInputs
81  * @param {Object} objetMetier instance d'un module métier (par exemple instance
    d'adresse, de personne...).
82  *      cet objet doit implémenter des interfaces précises (
    getProperty(), getModule(), etc.)
83  * @param {string} formId id du formulaire dans lequel l'input sera inséré
84  * @return {string} le code HTML des tous les inputs correspondant aux proprié-
    és de l'objet métier.
85  */
86  myApp.addModule.apply(myApp.gui, [ "getHtmlFormInputs", function(objetMetier,
    formId){
87
88      // Définition de l'interface commune aux modules métier (adresse, personne,
    etc.)
89      var metierCommonMethods = new Interface(
90          [ "getPropertyList", "getLabelText", "testRegex", "createInstance" ]);
91      // Définition de l'interface commune aux instances des modules métier (adresse
    , personne, etc.)
92      var metierCommonInstanceMethods = new Interface(
93          [ "getModule", "getProperty", "setProperty", "hasError", "getErrorMessage", "
    getErrorList" ]);
94
95      var testInterface = metierCommonInstanceMethods.isImplementedBy(objetMetier);
96      var message;
97      if (testInterface !== true){
98          message = testInterface;
99      }else{
100          message = metierCommonMethods.isImplementedBy(objetMetier.getModule());
101          if (message !== true){
102              throw new Error(message);
103          }
104      }
105
106      // Ajout du formulaire "mainForm" au médiateur qui gèrera ses événements.
107      myApp.ctrl.mediatorInputFilter.addForm(formId);
108
109      var htmlCode = "";
110
111      var propertyList = objetMetier.getModule().getPropertyList();
112
113      // Tous les inputs sont de type texte, donc on peut
114      // faire une boucle automatique sur les propriétés.
115      for (var i=0 ; i < propertyList.length ; i++){
116          var propertyName = propertyList[i];
117          // l'utilisateur ne peut pas modifier l'ID :
118          if (propertyName !== "id"){
119              // Concaténation du code HTML de l'input
120              htmlCode += myApp.gui.getInputCode(
121                  {objetMetier : objetMetier,
122                   propertyName : propertyName,
123                   formId : formId}) + "<br/>";
124          }
125      }
126
127      return htmlCode;
128  }]);

```

exemples/gui//ex03_formsGui.html

```

1  /<!doctype HHTML>
2  <html lang="fr">
3  <head>
4  <meta charset="UTF-8" />
5  <title>Filtrage d'inputs</title>
6  <link rel="stylesheet" href="basicStyle.css"/>
7  </head>
8  <body>
9  <h1>Saisie d'une adresse</h1>
10 <!-- Inclusion de la structure d'application et du module regexUtil -->
11 <script src="modulesMetier.js"></script>
12 <!-- Inclusion de jQuery pour les événements et manipulation du DOM -->
13 <script src="jquery.js"></script>
14 <script src="ex02_mediatorInputFilter.js"></script>
15 <script src="ex03_formsGui.js"></script>
16 <script src="./ex11_interfaceImplementation.js"></script>
17 <script>
18
19 // Ajout d'une méthode mainFunction
20 myApp.addModule("mainFunction", function(){
21
22 // création d'une instance
23 var adresse = myApp.metier.adresse.createInstance({
24   id: "04abf85bc9",
25   numeroRue: "2 bis@",
26   rue: "Rue de l'a Paix",
27   complementAdresse: "Bâtiment 3D",
28   codePostal: "63000",
29   ville: "Clermont-Ferrand",
30   pays: "France"
31 });
32
33 // Génération du formulaire avec les callbacks
34 document.write("<form id=\"mainForm\" method=\"post\">" +
35   myApp.gui.getHtmlFormInputs(adresse, "mainForm") +
36   "<label></label><input type=\"submit\" value=\"valider\">" +

```

```

37         "</form>");
38     });
39
40     // Exécution de la méthode mainFunction
41     myApp.mainFunction();
42 </script>
43 </body>
44 </html>

```

4.3.2 Avec l'interface d'objets métier utilisant le prototype

En utilisant le fabrique d'instances d'objets métier de la partie 3.4, le code de génération du formulaire est un peu plus simple (une seule interface à tester et suppression des appels de la méthode `getModule` :

exemples/gui/ex04_formsGuiPrototype.js

```

1 // Ajout d'un module "gui" à notre application myApp
2 myApp.addModule.apply(myApp, ["gui", {}]);
3
4 /**
5  * Fonction de génération de l'ID d'un élément HTML de type input préfixé par l'
6  * ID du formulaire
7  * @method myApp.gui.getInputId
8  * @param {Object} inputSpec contient les spécifications de l'input
9  * @param {string} inputSpec.formId id du formulaire dans lequel l'input sera
10  * inséré
11  * @param {string} inputSpec.propertyName nom de la propriété de inputSpec.
12  * objetMetier à saisir dans l'input
13 */
14 myApp.addModule.apply(myApp.gui, ["getInputId", function(inputSpec){
15     return inputSpec.formId + "_" + inputSpec.propertyName;
16 }]);
17
18 /**
19  * Publie auprès du Mediator un événement onchange d'un Input
20  * @param {string} formId id du formulaire dans lequel l'input sera inséré
21  * @param {string} propertyName nom de la propriété de inputSpec.objetMetier à
22  * saisir dans l'input
23 */
24 myApp.addModule.apply(myApp.gui, ["publishInputChange", function(formId,
25     propertyName){
26     myApp.ctrl.mediatorInputFilter.publish(formId, propertyName);
27 }]);
28
29 /**
30  * Génération du code HTML d'un input.
31  * @method myApp.gui.getTextInputCode
32  * @param {Object} inputSpec contient les spécifications de l'input
33  * @param {Object} inputSpec.objetMetier instance d'un module métier (par exemple
34  * instance d'adresse, de personne...).
35  * cet objet doit implémenter des interfaces précises (
36  * getProperty(), getModule(), etc.)
37  * @param {string} inputSpec.formId id du formulaire dans lequel l'input sera
38  * inséré

```

```

31  * @param {string} inputSpec.propertyName nom de la propriété de inputSpec.
    objetMetier à saisir dans l'input
32  * @param {string} [inputSpec.type=text] type de l'input
33  * @param {number} [inputSpec.inputSize=10] taille de l'input (nombre de
    caractères)
34  */
35  myApp.addModule.apply(myApp.gui, [ "getInputCode", function(inputSpec){
36      // Calcul de l'ID de l'input :
37      var inputId = myApp.gui.getInputId(inputSpec);
38
39      // Valeur de la propriété de l'objet pour l'attribut value de l'input
40      var propertyValue = inputSpec.objetMetier.getProperty(inputSpec.propertyName
    ) || "";
41      // Création d'un éventuel message si l'objet comportait déjà une erreur
42      var errorMessage = inputSpec.objetMetier.getErrorMessage(inputSpec.
    propertyName) !== undefined
43          ? inputSpec.objetMetier.getErrorMessage(inputSpec.propertyName)
    + "<br/>" : "";
44
45      //////////////////////////////////////
46      // Callback de gestion du filtrage de l'input :
47      myApp.ctrl.mediatorInputFilter.subscribe(inputSpec.formId, inputSpec.
    propertyName, function(){
48
49          var resultatTestRegex = inputSpec.objetMetier.testRegex(inputSpec.
    propertyName,
50              document.getElementById(inputId).value);
51          if (resultatTestRegex !== true){
52              document.getElementById("error_"+inputId).innerHTML = resultatTestRegex
    + "<br/>";
53          }else{
54              document.getElementById("error_"+inputId).innerHTML = "";
55          }
56      }); // fin du callback //////////////////////////////////////
57
58      var inputType = inputSpec.inputType === undefined ? "text" : inputSpec.
    inputType;
59      var inputSize = inputSpec.inputSize === undefined ? "10" : inputSpec.inputSize
    ;
60      var labelText = inputSpec.objetMetier.getLabelText(inputSpec.propertyName);
61
62      // retour du code HTML de l'input
63      return "<span class=\"errorMsg\" id=\"error_"+inputId+"\">" + errorMessage + "
    </span>" +
64          "<label for=\"" + inputSpec.propertyName + "\">" + labelText + "</label>"
    +
65          "<input type=\"" + inputType + "\" name=\"" + inputSpec.propertyName
    +
66          "\" id=\"" + inputId + "\" value=\"" + propertyValue + "\" " +
67          "size=\"" + inputSize + "\" " +
68          "onchange=\"myApp.gui.publishInputChange(' " + inputSpec.formId + "', '
    " + inputSpec.propertyName + "')\" ">";
69  });
70
71  /**
72  * Génération du code HTML de l'ensemble des inputs d'un formulaire.

```

```

73  * @method myApp.gui.getHtmlFormInputs
74  * @param {Object} objetMetier instance d'un module métier (par exemple instance
    d'adresse, de personne...).
75  *      cet objet doit implémenter des interfaces précises (
    getProperty(), getModule(), etc.)
76  * @param {string} formId id du formulaire dans lequel l'input sera inséré
77  * @return {string} le code HTML des tous les inputs correspondant aux proprié-
    tés de l'objet métier.
78  */
79  myApp.addModule.apply(myApp.gui, [ "getHtmlFormInputs", function(objetMetier,
    formId){
80
81      // Définition de l'interface commune aux instances des modules métier (adresse
    , personne, etc.)
82      var metierCommonInstanceMethods = new Interface(
83          [ "getPropertyList", "getLabelText", "testRegex", "getProperty",
84            "setProperty", "hasError", "getErrorMessage", "getErrorList" ]);
85
86      var testInterface = metierCommonInstanceMethods.isImplementedBy(objetMetier);
87      var message;
88      if (testInterface !== true){
89          throw new Error(testInterface);
90      }
91
92      // Ajout du formulaire "mainForm" au médiateur qui gèrera ses événements.
93      myApp.ctrl.mediatorInputFilter.addForm(formId);
94
95      var htmlCode = "";
96
97      var propertyList = objetMetier.getPropertyList();
98
99      // Tous les inputs sont de type texte, donc on peut
100     // faire une boucle automatique sur les propriétés.
101     for (var i=0 ; i < propertyList.length ; i++){
102         var propertyName = propertyList[i];
103         // l'utilisateur ne peut pas modifier l'ID :
104         if (propertyName !== "id"){
105             // Concaténation du code HTML de l'input
106             htmlCode += myApp.gui.getInputCode({
107                 objetMetier : objetMetier,
108                 propertyName : propertyName,
109                 labelText : objetMetier.getLabelText(propertyName),
110                 formId : formId
111             }) + "<br/>";
112         }
113     }
114
115     // champs caché représentant l'ID de l'instance
116     htmlCode += "<input type='hidden' id='" + formId + "_id' value='" +
117         objetMetier.getProperty("id") + "'/>";
118
119     return htmlCode;
120 }});

```

Chapitre 5

Exemple d'Application Interactive

5.1 Principe de l'application et analyse fonctionnelle

Notre application, qui possède un *modèle* constitué d'une collection de personnes, permet (voir les *storyboards* sur la figure 5.1) :

- D'afficher la liste des noms de personnes (*items*) ;
- De sélectionner une personne en cliquant sur l'*item* correspondant (l'*item* est alors surligné et les détails concernant cette personne sont affichés) ;
- De modifier les données de la personnes (en l'occurrence le nom) en cliquant sur un bouton "Modifier".
- D'ajouter une personne ;
- De supprimer la personne sélectionnée.
- d'ajouter, de supprimer ou de modifier une adresse pour la personne sélectionnée.

Comme on peut le voir, nous avons une *agrégation* entre les personnes et les adresses, une personne pouvant avoir plusieurs adresses.

En recensant les événements (*clics* de boutons d'*items*, liens) possibles sur les *storyboards* de la figure 5.1, on dresse le diagramme de cas d'utilisation représenté sur la figure 5.2.

5.2 Modèle de donnée

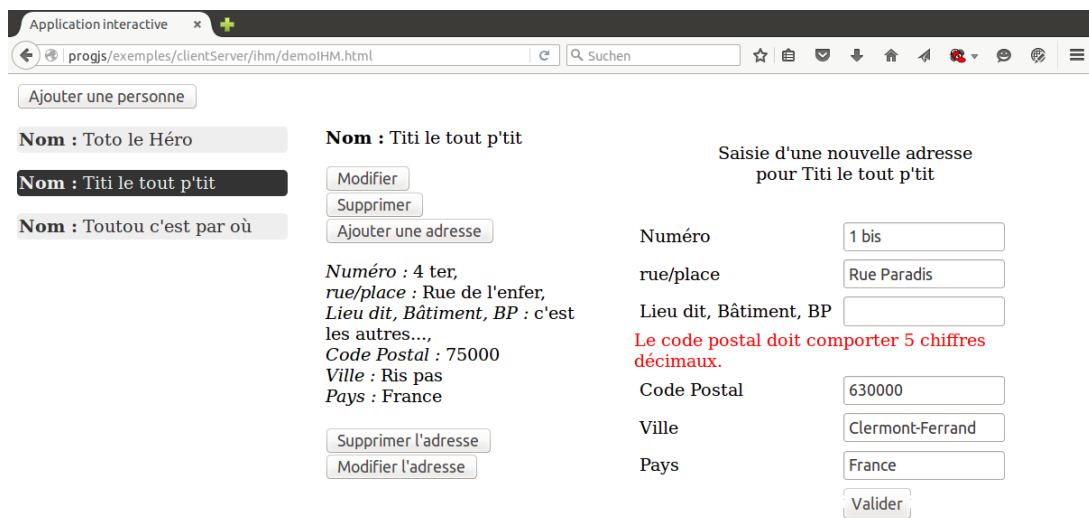
Dans notre modèle de données, une classe *personne* comporte un nom et une composition avec des instances d'adresse. Nous créons, pour le moment, quelques instances "en dur", dans un tableau **personnes**, avec chacune une adresse. Une autre propriété **selectedPersonne** contient une référence vers l'instance de personne sélectionnée (*item* surligné et détails affichés).

exemples/ihm/ex00_modelModule.js

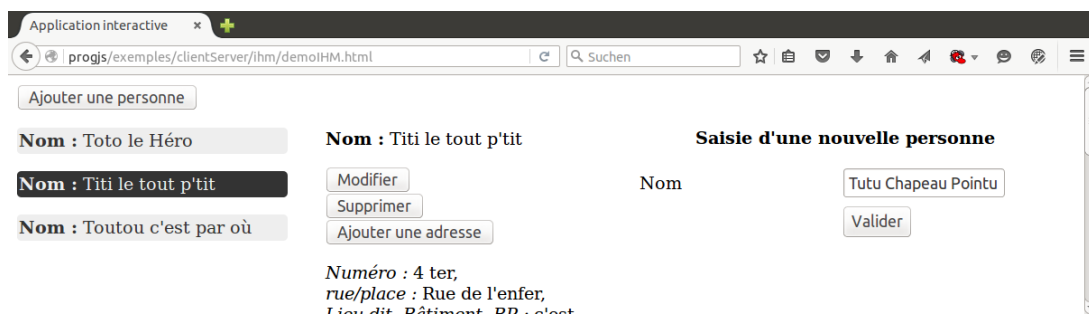
```
1 myApp.addModule.apply(myApp, [ "modele", {  
2   selectedPersonne : null ,  
3   personnes : [] ,  
4 ]
```




(a) Sélection d'une personne (*item* surligné à gauche)



(b) Ajout d'une adresse pour la personne sélectionnée



(c) Ajout d'une personne



(d) Après ajout d'une personne

FIGURE 5.1 : Captures d'écran de notre application

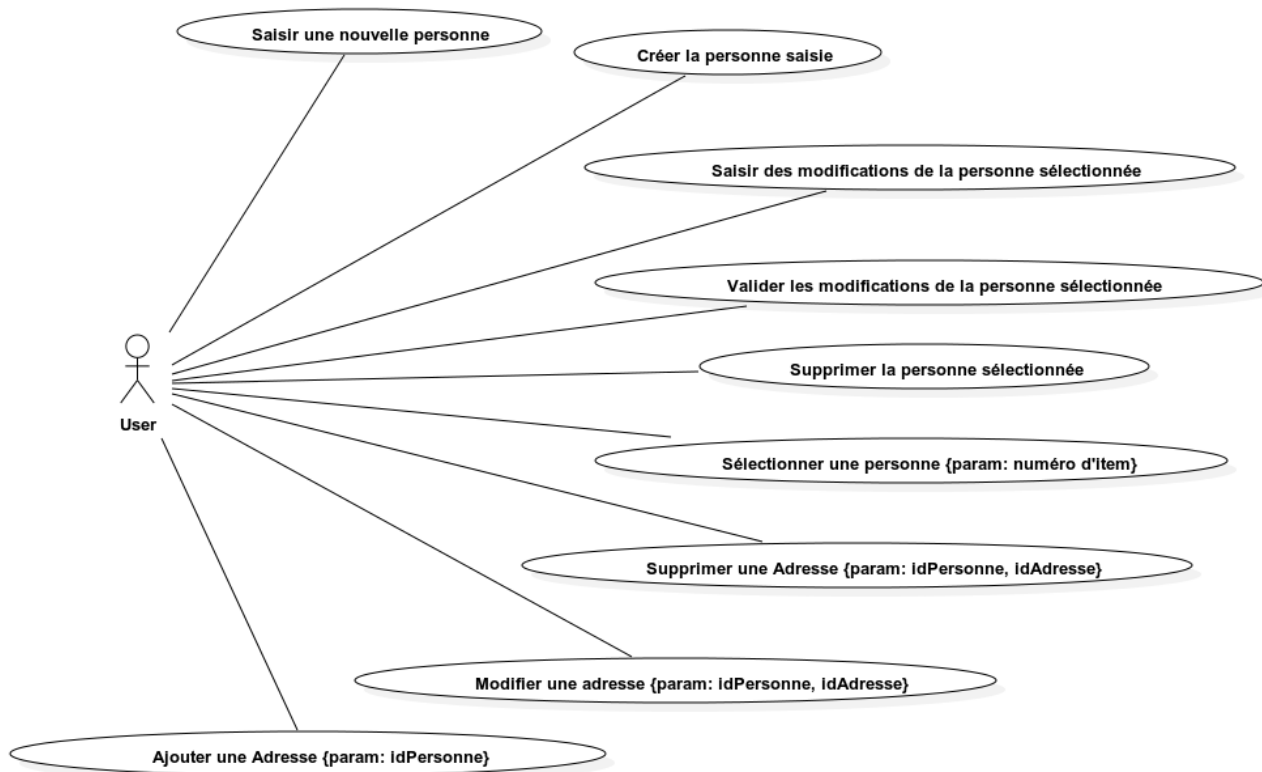


FIGURE 5.2 : Diagramme de cas d'utilisation de notre application avec son *IHM*

```

5   });
6
7   myApp.modele.personnes.push(myApp.metier.personne.createInstance({
8     id: "0123abcdef",
9     nom: "Toto le Héro",
10    adresse: myApp.metier.adresse.createInstance({
11      id: "04abf85bc9",
12      numeroRue: "2 bis",
13      rue: "Rue de l'a Paix",
14      complementAdresse: "Dalle",
15      codePostal: "63000",
16      ville: "Clermont-Ferrand",
17      pays: "France 2"
18    })
19  }));
20
21  myApp.modele.personnes.push(myApp.metier.personne.createInstance({
22    id: "0123abcd12",
23    nom: "Titi le tout p'tit",
24    adresse: myApp.metier.adresse.createInstance({
25      id: "04abf85bb5",
26      numeroRue: "4 ter",
27      rue: "Rue de l'enfer",
28      complementAdresse: "c'est les autres...",
29      codePostal: "75000",
30      ville: "Ris pas",
31      pays: "France"
32    })
  })

```

```

33     }));
34
35 myApp.modele.personnes.push(myApp.metier.personne.createInstance({
36     id: "0123abcd01",
37     nom: "Toutou c'est par où",
38     adresse: myApp.metier.adresse.createInstance({
39         id: "04abf85ba4",
40         numeroRue: "1",
41         rue: "Place de l'Alternative",
42         complementAdresse: "Pourquoi pas",
43         codePostal: "63123",
44         ville: "Les Paumiers",
45         pays: "France"
46     })
47 }));

```

5.3 *Pattern Mediator* : centraliser les événements

Notre module `mediator` va nous permettre :

- De découpler l'implémentation de la réaction aux événements utilisateurs (modification du modèle, mise à jour des vues) de la gestion de ces événements utilisateurs via la technologie *jQuery*, qui, de ce fait, se trouve circonscrite à une seule classe (*Wrapper*).
- D'éliminer les dépendances cycliques entre les modules de notre application ;
- De recenser les événements utilisateurs de manière lisible dans un module centralisé ;
- De provoquer des mises à jour de panneaux de la vue qui observent des propriétés du modèle.

Contrairement au médiateur spécialisé dans le filtrage des attributs de formulaires décrit dans la partie 4.2, le module `mediator` va nous permettre d'exécuter plusieurs *callback* en réaction à un même événement (par exemple pour mettre à jour différentes parties de la vue après une modification du modèle).

exemples/ihm/ex01_mediator.js

```

1  /**
2   * Implémentation du pattern "Médiateur" pour la gestion des événements
3   *   utilisateurs ,
4   *   et la mise à jour des vues (ou des sous-arbres du DOM)
5   */
6  myApp.addModule.apply(myApp.gui, [ "mediator", function() {
7
8      /**
9       * Liste des événements pour lesquels une liste de callbacks peut être
10       *   enregistrée
11       * @private
12       */
13     var m_subscriptionLists;

```

```

14  * Initialise la liste des événements, avec pour chacun, une liste de
15      callbacks vide.
16  * @method init
17  */
18  var init = function() {
19      m_subscriptionLists = {
20          // Opérations CRUD sur les personnes
21          "personne/read": [], // Lire toutes les personnes pour (re)onstruire le
                modèle
22
23          "personne/update": [], // validation du formulaire de mise à jour de la
                personne sélectionnée.
24          "personne/create": [], // validation du formulaire d'ajout d'une personne.
25          "personne/delete" : [], // Suppression d'une personne
26
27          // Opérations CRUD sur les adresses
28          "adresse/create": [], // validation du formulaire d'ajout d'une adresse.
29          "adresse/update" : [], // mise à jour d'une adresse
30          "adresse/delete" : [], // Suppression d'une adresse
31
32          // Actions Utilisateur donnant lieu à un changement de le vue
33          "personne/selectDetails" : [], // Sélection d'une personne pour voir les d
                étails
34          "personne/edit": [], // click sur la modification de la personne sé
                lectionnée
35          "personne/saisie": [], // click sur la modification de la personne sé
                lectionnée
36
37          "adresse/edit" : [], // Suppression d'une adresse
38          "adresse/saisie": [], // click sur la modification de la personne sé
                lectionnée
39
40          // Notifications de modification du modèle pour requête AJAX et/ou mise à
                jour de la vue
41          "personne/changed" : [], // mise à jour d'une personne
42          "personne/created" : [], // mise à jour d'une personne
43          "personne/detailsChanged": [], // Mise à jour requise du panneau des dé
                tails
44
45          // Notifications de modification du modèle pour requête AJAX et/ou mise à
                jour de la vue
46          "adresse/changed" : [], // mise à jour d'une adresse
47          "adresse/created" : [], // mise à jour d'une adresse
48
49
50          // Demande de ré-enregistrement d'événements utilisateurs suite à
                reconstruction d'éléments HTML
51          "personne/htmlListItemRebuilt": [], // Réenregistrement des événements de
                click sur les items
52
                // suite à reconstruction complète du
                code HTML des items.
53          "personne/detailsRebuilt": [], // Réenregistrement des événements de click
                sur les boutons "Supprimer", "Modifier"
54
                // suite à reconstruction du code HTML
                des détails.

```

```

55     };
56   };
57 };
58
59 // Appel de la méthode d'initialisation
60 init();
61
62 /**
63  * Interface publique du module mediator
64  */
65 var publicInterfaceMediator = {
66
67   /**
68    * Enregistrement d'un callback sur un événement.
69    * Il peut y avoir plusieurs callbacks sur un même événement
70    * (par exemple : mise à jour de deux parties distinctes de la vue)
71    * @param {string} eventCateg événement, qui doit être un nom de propriété
72    *   de m_subscriptionLists
73    * @param {function} callbackFunction la fonction qui sera appelée en ré
74    *   action à l'événement.
75    */
76   subscribe : function(eventCateg, callbackFunction){
77     if (m_subscriptionLists.hasOwnProperty(eventCateg)){
78       m_subscriptionLists[eventCateg].push({callback : callbackFunction});
79     }else{
80       throw new Error("Catégorie d'événements " + eventCateg + " inconnue du m
81         édiateur");
82     }
83   },
84
85   /**
86    * Publication d'un événement survenu et exécution de tous les callbacks
87    * correspondants.
88    * @param {string} eventCateg événement, qui doit être un nom de propriété
89    *   de m_subscriptionLists
90    * @param {Object} contextArg argument optionnel à transmettre au callback (
91    *   exemple : item cliqué...)
92    */
93   publish : function(eventCateg, contextArg){
94     var i;
95     if (m_subscriptionLists.hasOwnProperty(eventCateg)){
96       for (i=0 ; i< m_subscriptionLists[eventCateg].length ; ++i){
97         // On appelle le callback avec son
98         m_subscriptionLists[eventCateg][i].callback(contextArg);
99       }
100     }else{
101       throw new Error("Catégorie d'événements " + eventCateg + " inconnue du m
102         édiateur");
103     }
104   },
105
106   // Réinitialise les listes de callbacks à vide.
107   empty : function(){
108     init();
109   }
110 };

```

```

104
105     return publicInterfaceMediator;
106 }());

```

5.4 Événements concernant les personnes

5.4.1 Enregistrement des événements utilisateurs via *jQuery*

Tous les événements recensés dans le **diagramme de cas d'utilisation** (voir la figure 5.2) se verront ici attribué un gestionnaire qui, généralement, ne fera que publier l'événement auprès de *mediator* (partie 5.3). Les éléments *HTML* constants de la vue (``, `<button>`, `<div>`, `<p>`, etc.) sur lesquels ces événements seront appliqués sont définis dans le fichier *HTML* principal décrit dans la partie 5.4.9.

Ces événements utilisateurs doivent parfois être réenregistrés suite à la reconstruction des éléments *HTML* concernés. Les événements sont alors détruits (méthodes `jQuery.off()`, ou `jQuery.empty()`, ou encore `jQuery.remove()`), puis, le code *HTML* est régénéré, et enfin, les événements utilisateur sont ré-enregistrés (méthode `jQuery.on()`).

S'il faut prévoir de ré-enregistrer un gestionnaire d'événement utilisateur, nous allons permettre de déclencher ce ré-enregistrement via le *mediator*. Ceci permet d'éviter notamment des problèmes de dépendance cyclique des fonctions *JavaScript* ou modules, par exemples du fait que les événements *jQuery* doivent être initialisés après la génération de la vue.

exemples/ihm/ex02_guijQueryEventsPersonne.js

```

1  /**
2   * Méthode d'initialisation des événements utilisateurs JavaScript.
3   * Enregistrement des gestionnaires de ces événements via jQuery.
4   */
5  myApp.addModule.apply(myApp.gui, [ "initjQueryEventsPersonne", function() {
6
7      //////////////////////////////////////
8      // click sur le bouton "Ajouter une personne" faisant sortir le formulaire
9
10     /**
11      * Gestionnaire click sur le bouton faisant sortir le formulaire
12      */
13     var clickBoutonSaisiePersonne = function(event){
14         // publication auprès du médiateur
15         myApp.gui.mediator.publish("personne/saisie", {
16             personne : myApp.modele.selectedPersonne
17         });
18     };
19
20     // Enregistrement du Handler du click pour modifier les détails de l'item sélectionné via jQuery
21     $("#boutonAjouterPersonne").on("click", clickBoutonSaisiePersonne);
22
23     //////////////////////////////////////
24     // click sur le bouton "Modifier le nom" faisant sortir le formulaire
25
26     /**
27      * Gestionnaire click sur le bouton faisant sortir le formulaire

```

```

28  */
29  var clickBoutonModifierPersonne = function(event){
30
31      // publication auprès du médiateur
32      myApp.gui.mediator.publish("personne/edit", {
33          personne : myApp.modele.selectedPersonne
34      });
35  };
36
37  //////////////////////////////////////
38  // click sur le bouton "Supprimer la personne" faisant sortir le formulaire
39
40  /**
41   * Gestionnaire click sur le bouton faisant sortir le formulaire
42   */
43  var clickBoutonSupprimerPersonne = function(event){
44      // publication auprès du médiateur
45      myApp.gui.mediator.publish("personne/delete", {
46          personne : myApp.modele.selectedPersonne
47      });
48  };
49
50
51  //////////////////////////////////////
52  // Gestionnaire de submit formulaire de modification de personne.
53
54  /**
55   * Gestionnaire de l'événement submit du formulaire.
56   * @param {Event} jQuery event correspondant au handler.
57   */
58  var formHandlerModifPersonne = function(event){
59
60      // Éviter d'appeler l'"action" par défaut () script PHP, etc...)
61      // du formulaire lors du submit
62      event.preventDefault();
63
64      // publication auprès du médiateur
65      myApp.gui.mediator.publish("personne/update", {
66          personne : myApp.modele.selectedPersonne
67      });
68  } // fin du gestionnaire formHandlerModif()
69
70  // Enregistrement du Handler du submit du formulaire via jQuery
71  $("#modifierPersonneForm").on("submit", formHandlerModifPersonne);
72
73
74  //////////////////////////////////////
75  // Gestionnaire de submit formulaire d'ajout de personne.
76
77  /**
78   * Gestionnaire de l'événement submit du formulaire.
79   * @param {Event} jQuery event correspondant au handler.
80   */
81  var formHandlerAjoutPersonne = function(event){
82
83      // Éviter d'appeler l'"action" par défaut () script PHP, etc...)

```

```

84      // du formulaire lors du submit
85      event.preventDefault();
86
87      // publication auprès du médiateur
88      myApp.gui.mediator.publish("personne/create", {
89          personne : myApp.modele.selectedPersonne
90      });
91  } // fin du gestionnaire formHandlerAjout()
92
93  // Enregistrement du Handler du submit du formulaire via jQuery
94  $("#ajouterPersonneForm").on("submit", formHandlerAjoutPersonne);
95
96
97  /**
98   * Enregistre les événements de clicks sur les boutons "Modifier" et "
    Supprimer"
99   * la personne sélectionnée.
100  * Cette fonction doit être invoquée en cas de sélection d'une nouvelle
    personne
101  * (reconstruction du code HTML du panneau des détails.
102  */
103  var registerButtonClickEvents = function(){
104      // Enregistrement du Handler du click pour modifier les détails de l'item sé
    lectionné via jQuery
105      $("#boutonModifierPersonne").on("click", clickBoutonModifierPersonne);
106      // Enregistrement du Handler du click pour supprimer l'item sélectionné via
    jQuery
107      $("#boutonSupprimerPersonne").on("click", clickBoutonSupprimerPersonne);
108  }
109
110  //////////////////////////////////////
111  // Clicks sur les éléments de la liste d'items
112
113  /** Méthode qui permet de créer un gestionnaire d'événement de click
114   * sur chaque nom de personnes (sélection des détails)
115   * Ces gestionnaires publient l'événement "nouvelle personne sélectionnée"
    auprès du médiateur.
116   * @param {int} index indice de l'item pour lequel on enregistre l'événement.
117   */
118  var registerHelperSelectDetails = function(index){
119      return function(){
120          myApp.gui.mediator.publish("personne/selectDetails",
121              {
122                  personne : myApp.modele.personnes[index]
123              });
124      };
125  };
126
127  /**
128   * Enregistre les événements javascript de click sur les éléments de la liste
129   * (noms des personnes).
130   * Cette méthode doit être appelée lors de la régénération du code de la liste
    .
131   * @method registerListePersonnesClicks
132   * @param {Object} contextArgs non utilisé
133   * @return {function} une fonction callback qui gère le click sur l'item

```



```

134     index
135     */
136     var registerListePersonnesClicks = function(contextArgs){
137         for (var i=0 ; i<myApp.modele.personnes.length ; ++i){
138             $("#master_" + myApp.modele.personnes[i].getId()).on(
139                 "click", registerHelperSelectDetails(i));
140         }
141     };
142
143     // Enregistrer les clicks lors de l'initialisation
144     registerButtonClickEvents();
145     registerListePersonnesClicks();
146
147     // Permet à la méthode qui régénère toute la liste des items
148     // de recréer, via le médiateur, les événements "click" sur les items.
149     myApp.gui.mediator.subscribe("personne/htmlListeItemRebuilt",
150         registerListePersonnesClicks);
151
152     // Permet à la méthode qui régénère le panneau des détails de recréer,
153     // via le médiateur, les événements "click" sur les boutons dans le panneau des
154     // détails.
155     myApp.gui.mediator.subscribe("personne/detailsRebuilt",
156         registerButtonClickEvents());
157 }));

```

5.4.2 Mise à jour du panneau des détails

Le panneau des détails de l'*item* sélectionné doit être mis à jour lors de la modification de la personne par validation du formulaire, ou lors du changement de l'*item* sélectionné (*click* sur un autre *item*). dans ce cas, les événements utilisateurs sur les éléments *HTML* qui sont générés dynamiquement sur le panneau des détails doivent aussi être reconstruit (événement *personne/detailsRebuilt* du *mediator*).

exemples/ihm/ex03_guiDetailsChanged.js

```

1  /**
2   * Définition et enregistrement des callbacks de mise à jour des détails de l'
3   * item sélectionnée.
4   */
5  myApp.addModule.apply(myApp.gui, [ "callbacksUpdateDetails", function() {
6
7      /**
8       * Génération du code HTML des détails de la personne sélectionnée.
9       */
10     var getHtmlCodeDetail = function() {
11         var htmlCode = "<span class=\"panel\">" +
12             "<p><strong>Nom </strong>" + myApp.modele.selectedPersonne.getNom()
13             + "</p>" +
14             "<button id=\"boutonModifierPersonne\">Modifier</button><br/>" +
15             "<button id=\"boutonSupprimerPersonne\">Supprimer</button><br/>" +
16             "<button id=\"boutonAjouterAdresse\">Ajouter une adresse</button>";
17         for (var index = 0 ; index < myApp.modele.selectedPersonne.getNbAdresses() ;
18             ++index){
19             htmlCode += "<p>" +

```

```

17         myApp.view.adresse.getHtmlDevelopped(myApp.modele.
18             selectedPersonne.getAdresse(index))
19         + "<br/><button id=\"boutonSupprimerAdresse_\"
20         + myApp.modele.selectedPersonne.getAdresse(index).getProperty(
21             'id')
22         + \">Supprimer l'adresse</button>\"
23         + "<br/><button id=\"boutonModifierAdresse_\"
24         + myApp.modele.selectedPersonne.getAdresse(index).getProperty(
25             'id')
26         + \">Modifier l'adresse</button>\"
27         + "</p>\";
28     }
29     htmlCode += "</span>\";
30     return htmlCode;
31 };
32
33 /**
34  * Redessine les détails d'une personne suite à sa sélection ou sa
35  * modification.
36  * @param {Object} contextArg non utilisé.
37  */
38 var repaintDetail = function(contextArg){
39
40     $("#modifierPersonneForm").empty(); // Vider les inputs et les événements JS
41     // existant
42     $("#ajouterPersonneForm").empty(); // Vider les inputs et les événements JS
43     // existant
44     $("#ajouterAdresseForm").empty(); // Vider les inputs et les événements JS
45     // existant
46     $("#modifierAdresseForm").empty(); // Vider les inputs et les événements JS
47     // existant
48
49     $("#vueDetail").empty(); // Vider les détails de l'item sélectionné
50
51     $("#vueDetail").html(getHtmlCodeDetail()); // Génération du code HTML
52
53     // Recréer les événements de clicks sur les boutons "modifier", "supprimer",
54     // etc.
55     myApp.gui.mediator.publish("personne/detailsRebuilt");
56 };
57
58 // Enregistrement du callback de l'événement dédié (m.a.j. des détails)
59 myApp.gui.mediator.subscribe("personne/detailsChanged", repaintDetail);
60 // Enregistrement du callback de l'événement de mise à jour de la personne
61 myApp.gui.mediator.subscribe("personne/changed", repaintDetail);
62
63 }());

```

5.4.3 Mise à jour du panneau des *items*

Le panneau qui affiche la liste des *items* doit être mis à jour lors de la modification de la personne par validation du formulaire (le nom de la personne peut changer), ou lors du changement de l'*item* sélectionné, celui-ci étant surligné.

En cas de changement de l'*item* sélectionné, la propriété `selectedPersonne` du modèle sera

modifiée, et le rafraîchissement du panneau des détails sera ensuite provoqué.

Lors de la création d'une nouvelle personne, celle-ci sera automatiquement sélectionnée.

exemples/ihm/ex04_guiPersonneChanged.js

```

1  /**
2   * Définition et abonnement des callbacks de mise à jour de la
3   * liste clickable des items, soit lors de la modification
4   * du modèle, soit lors du changement de personne sélectionnée.
5   */
6  myApp.addModule.apply(myApp.gui, [ "callbacksMainListUpdate", function() {
7
8      /**
9       * Active ou désactive le surlignage (style CSS) d'un item de la liste.
10      * @param {personne} personne item de la liste à modifier (via l'ID de l'élé
11      *   ment HTML).
12      * @param {boolean} highlighted true si on doit surligner, false pour
13      *   remettre le style par défaut.
14      */
15      var setHighlighted = function(personne, highlighted){
16          if (highlighted){
17              // Mettre le style surligné sur l'item de la liste
18              $("#master_" + personne.getId()).css("background-color", "#333")
19              .css("color", "#eee")
20              .css("border-radius", "4px")
21              .css("padding", "2px");
22          }else{
23              // Remettre le style normal sur l'item de la liste
24              $("#master_" + personne.getId()).css("background-color", "#eee")
25              .css("color", "#333")
26              .css("border-radius", "4px")
27              .css("padding", "2px");
28          }
29      }
30
31      /**
32      * Génération du code HTML de la liste de personnes
33      */
34      var getHtmlCodeListePersonnes = function(){
35          var htmlCode = "";
36          for (i=0 ; i<myApp.modele.personnes.length ; ++i){
37              htmlCode +=
38                  "<p id=\"master_\"+ myApp.modele.personnes[i].getId() + \">\" +
39                  "<strong>Nom </strong> \" + myApp.modele.personnes[i].getNom() + "</p>";
40          }
41          return htmlCode;
42      };
43
44      /**
45      * Raffraichissement (ou affichage) de toute la vue.
46      * @param contextArg non utilisé.
47      */
48      var repaintVue = function(contextArg){
49          $("#listePersonnes").empty(); // Vider la liste et ses événements
50          $("#listePersonnes").html(getHtmlCodeListePersonnes()); // afficher

```

```

51 // Appliquer le style par défaut sur tous les items
52 for (var i=0 ; i < myApp.modele.personnes.length ; ++i){
53     setHighlighted(myApp.modele.personnes[i], false);
54 }
55 // Surligner l'item sélectionné
56 setHighlighted(myApp.modele.selectedPersonne, true);
57
58 // Recréer les événements de clicks sur les items de la liste
59 myApp.gui.mediator.publish("personne/htmlListeItemRebuilt");
60 };
61
62 /**
63  * Changer l'item sélectionné en réaction à un click.
64  * @param {Object} contextArg argument indiquant la nouvelle personne sé-
65  *   lectionnée.
66  * @param {personne} contextArg.personne nouvelle personne sélectionnée.
67  */
68 var selectPersonne = function(contextArg){
69     // Supprimer le surlignage de l'ancienne personne sélectionnée
70     setHighlighted(myApp.modele.selectedPersonne, false);
71
72     // Changer l'item sélectionné
73     myApp.modele.selectedPersonne = contextArg.personne;
74
75     // Mettre le style surligné sur l'item sélectionné de la liste
76     setHighlighted(myApp.modele.selectedPersonne, true);
77
78     // Provoquer la mise à jour du panneau des détails
79     myApp.gui.mediator.publish("personne/detailsChanged", {
80         personne : myApp.modele.selectedPersonne
81     });
82 };
83
84 /**
85  * Changer l'item sélectionné suite à création d'une personne et mise à jour
86  *   de la vue.
87  * @param {Object} contextArg argument indiquant la nouvelle personne sé-
88  *   lectionnée.
89  * @param {personne} contextArg.personne nouvelle personne sélectionnée.
90  */
91 var selectPersonneAnRepaint = function(contextArg){
92     selectPersonne(contextArg);
93     repaintVue();
94 }
95
96 // Enregistrement du callback de modification de la personne
97 myApp.gui.mediator.subscribe("personne/changed", repaintVue);
98 // Enregistrement du callback de création de la personne
99 myApp.gui.mediator.subscribe("personne/created", selectPersonneAnRepaint);
100 // Enregistrement du callback de sélection d'une nouvelle personne.
101 myApp.gui.mediator.subscribe("personne/selectDetails", selectPersonne);
102 }());

```

5.4.4 Bouton "Supprimer"

Lorsque l'utilisateur clique sur "Supprimer", la personne sélectionnée est supprimée du modèle. Une nouvelle personne est sélectionnée (personne par défaut) et la vue est réinitialisée.

exemples/ihm/ex07_guiBoutonSupprimerPersonne.js

```

1  /**
2   * Définition et enregistrement des callbacks appelés à gérer le clic sur le
      bouton
3   * "modifier" la personne sélectionnée.
4   */
5  myApp.addModule.apply(myApp.gui, [ "callbacksClickSupprimer", function() {
6
7   /**
8   * Callback qui supprime la personne passée dans l'objet passé en argument.
9   * @param {Object} contextArg argument indiquant la personne à supprimer.
10  * @param {personne} contextArg.personne référence de l'instance de personne à
      supprimer dans le modèle.
11  */
12  var deletePersonne = function(contextArg){
13    // Indice dans le tableau de la personne à supprimer.
14    var indexSelectedPersonne = myApp.modele.personnes.indexOf(contextArg.
      personne);
15    // Suppression de la personne dans le modèle
16    myApp.modele.personnes.splice(indexSelectedPersonne, 1);
17    // Personne sélectionnée par défaut
18    myApp.modele.selectedPersonne = myApp.modele.personnes[0];
19
20    // Provoquer la mise à jour de la vue :
21    myApp.gui.mediator.publish("personne/changed", {
22      personne : myApp.modele.selectedPersonne
23    });
24  }
25
26  // Enregistrement du callback
27  myApp.gui.mediator.subscribe("personne/delete", deletePersonne);
28
29  }());

```

5.4.5 Bouton "Modifier" et affichage du formulaire

Lorsque l'utilisateur clique sur "Modifier", le formulaire doit être affiché avec les données de la personnes dans les *inputs*.

exemples/ihm/ex06_guiBoutonModifierPersonne.js

```

1  /**
2   * Définition et enregistrement des callbacks appelés à gérer le clic sur le
      bouton
3   * "modifier" la personne sélectionnée.
4   */
5  myApp.addModule.apply(myApp.gui, [ "callbacksClickModifierPersonne", function() {
6
7   /**
8   * Génération du code HTML du formulaire de modification de la personne sé
      lectionnée.

```

```

9      */
10     var getHtmlFormInputs = function() {
11         return "<span style=\`width :360px; display : inline-block; vertical-align :
            top;\`>" +
12             myApp.gui.getHtmlFormInputs(myApp.modele.selectedPersonne, "
                modifierPersonneForm") +
13             "<label×/label×input type=\`submit\` value=\`Valider\`×/input>" +
14             "</span>";
15     }
16
17     /**
18      * Callback d’Affichage (via le DOM) du formulaire dans l’élément d’ID "
        modifierPersonneForm"
19      * @param {Object} contextArg non utilisé.
20      */
21     var repaintFormInputs = function(contextArg){
22         $("#modifierPersonneForm").empty(); // Vider les inputs et les événements JS
            existant
23         $("#ajouterPersonneForm").empty(); // Vider les inputs et les événements JS
            existant
24         $("#ajouterAdresseForm").empty(); // Vider les inputs et les événements JS
            existant
25         $("#modifierAdresseForm").empty(); // Vider les inputs et les événements JS
            existant
26
27
28         $("#modifierPersonneForm").append(getHtmlFormInputs()); // ajouter les
            nouveaux inputs
29     };
30
31     // Enregistrement du callback
32     myApp.gui.mediator.subscribe("personne/edit", repaintFormInputs);
33
34 }());

```

5.4.6 Bouton "Ajouter une personne"

Lorsque l'utilisateur clique sur "Ajouter une personne", le formulaire doit être affiché avec les valeurs par défaut (typiquement des champs vides) dans les *inputs*.

Pour cela, on utilise la possibilité offerte par la fabrique de nos modules métier (partie 3.4) de créer un objet par défaut en passant `null` en argument de la fabrique. Ceci permet de ne pas générer de messages d'erreur en cas de champs obligatoire initialement vide.

Après validation du formulaire, la personne est ajoutée dans le modèle, elle est automatiquement sélectionnée, et la vue est mise à jour.

exemples/ihm/ex05_guiBoutonAjouterPersonne.js

```

1  /**
2   * Définition et enregistrement des callbacks appelés à gérer le clic sur le
        bouton
3   * "modifier" la personne sélectionnée.
4   */
5  myApp.addModule.apply(myApp.gui, ["callbacksClickAjouter", function() {
6
7      /**

```

```

8      * Génération du code HTML du formulaire de modification de la personne sé
      lectionnée.
9      */
10     var getHtmlFormInputs = function() {
11         return "<span style=\|width :360px; display : inline-block; vertical-align :
            top;\|>" +
12             "<strong style=\|width : 360px; display : inline-block; text-align :
                center; padding : 15px;\|>Saisie d'une nouvelle personne</strong>"
            +
13             myApp.gui.getHtmlFormInputs(myApp.metier.personne.createInstance(
                null), "ajouterPersonneForm") +
14             "<labelX/labelXinput type=\|submit\| value=\|Valider\|X/|input>" +
15             "</span>";
16     }
17
18     /**
19      * Callback d’Affichage (via le DOM) du formulaire dans l’élément d’ID "
        mainForm"
20      * @param {Object} contextArg non utilisé.
21      */
22     var repaintFormInputs = function(contextArg){
23         $("#modifierPersonneForm").empty(); // Vider les inputs et les événements JS
            existant
24         $("#ajouterPersonneForm").empty(); // Vider les inputs et les événements JS
            existant
25         $("#ajouterAdresseForm").empty(); // Vider les inputs et les événements JS
            existant
26         $("#modifierAdresseForm").empty(); // Vider les inputs et les événements JS
            existant
27
28         $("#ajouterPersonneForm").append(getHtmlFormInputs()); // ajouter les
            nouveaux inputs
29     };
30
31     // Enregistrement du callback
32     myApp.gui.mediator.subscribe("personne/saisie", repaintFormInputs);
33
34 }());

```

5.4.7 Validation du formulaire de modification

Lors de la validation (événement *submit*) du formulaire de modification, les données de la personne sélectionnée doivent être mises à jour à partir des valeurs saisies dans le formulaire. Les panneaux potentiellement impactés (liste des *items*, panneau des détails) sont alors mis à jour.

exemples/ihm/ex09_guiModifierPersonneFormValidate.js

```

1  /**
2   * Définition et enregistrement du callback réagissant à la validation (submit)
3   * du formulaire de modification d'une personne.
4   */
5  myApp.addModule.apply(myApp.gui, [ "callbacksValidateModifierForm", function() {
6      // Formulaire de modification d'une personne
7

```

```

8  /**
9   *  Modifie le modèle à partir des données saisies dans le formulaire
10  */
11  var updateModel = function() {
12
13      // 1) Mise à jour des données du modèle
14      // à partir des valeurs des inputs du formulaire
15      var propertyName,
16          inputId;
17      // Pour chaque propriété (chaque input du formulaire)
18      for (var j=0 ; j< myApp.metier.personne.getPropertyList().length ; ++j){
19          propertyName = myApp.metier.personne.getPropertyList()[j];
20          if (propertyName !== "id"){
21              // calcul de l'ID de l'input
22              inputId = myApp.gui.getInputId({
23                  propertyName : propertyName,
24                  formId : "modifierPersonneForm"
25              });
26              // Modification de la propriété de la personne
27              // avec la valeur saisie dans l'input.
28              myApp.modele.selectedPersonne.setProperty(propertyName,
29                  document.getElementById(inputId).value
30              );
31          }
32      }
33      // Provoquer la mise à jour des éléments de la vue observant la personne
34      myApp.gui.mediator.publish("personne/changed", {
35          personne : null
36      });
37  };
38
39  // Enregistrement du callback de l'événement de validation du formulaire
40  myApp.gui.mediator.subscribe("personne/update", updateModel);
41
42
43 }());

```

5.4.8 Validation du formulaire d'ajout d'une personne

Lors de la validation (événement *submit*) du formulaire d'ajout, une personne doit être ajoutée au modèle à partir des valeurs saisies dans le formulaire. Les panneaux potentiellement impactés (liste des *items*, panneau des détails) sont alors mis à jour.

exemples/ihm/ex08_guiAjouterPersonneFormValidate.js

```

1  /**
2   *  Définition et enregistrement du callback réagissant à la validation (submit)
3   *  du formulaire de modification d'une personne.
4   */
5  myApp.addModule.apply(myApp.gui, ["callbacksValidateAjouterForm", function() {
6
7      /**
8       *  Modifie le modèle à partir des données saisies dans le formulaire
9       */
10     var updateModel = function() {

```



```

11 // 1) Mise à jour des données du modèle
12 // à partir des valeurs des inputs du formulaire
13 var propertyName,
14     inputId;
15
16 // Ajout d'une personne vide dans la collection
17 var nouvellePersonne = myApp.metier.personne.createInstance(null);
18 myApp.modele.personnes.push(nouvellePersonne);
19
20 // Pour chaque propriété (chaque input du formulaire)
21 for (var j=0 ; j< myApp.metier.personne.getPropertyList().length ; ++j){
22     propertyName = myApp.metier.personne.getPropertyList()[j];
23     if (propertyName != "id"){
24         // calcul de l'ID de l'input
25         inputId = myApp.gui.getInputId({
26             propertyName : propertyName,
27             formId : "ajouterPersonneForm"
28         });
29         // Modification de la propriété de la personne
30         // avec la valeur saisie dans l'input.
31         nouvellePersonne.setProperty(propertyName,
32             document.getElementById(inputId).value
33         );
34     }
35 }
36 }
37
38 // Provoquer la sélection de la nouvelle personne (et par suite la mise à
39 // jour de la vue)
40 myApp.gui.mediator.publish("personne/created", {
41     personne : nouvellePersonne
42 });
43
44 // Enregistrement du callback de l'événement de validation du formulaire
45 myApp.gui.mediator.subscribe("personne/create", updateModel);
46
47 }());

```

5.4.9 Code *HTML* de la vue et invocation des méthodes

Il faut surtout penser à inclure `jquery.js` le plus tard possible et à invoquer la méthode d'enregistrement des événements utilisateurs après la génération de la vue, qui crée les éléments *HTML* sur lesquels on applique ces événements.

exemples/ihm/ex10_demoIHM.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4     <meta charset="UTF-8" />
5     <title>Application interactive</title>
6     <link rel="stylesheet" href="basicStyle.css"/>
7 </head>
8 <body>

```

```

9  <script src="/modulesMetierPrototype.js"></script>
10 <script src="/mediatorInputFilter.js"></script>
11 <script src="/formsGuiPrototype.js"></script>
12 <script src="/interfaceImplementation.js"></script>
13 <script src="/personneModule.js"></script>
14 <script src="/modelModule.js"></script>
15 <script src="/mediator.js"></script>
16 <script src="/guiBoutonModifierPersonne.js"></script>
17 <script src="/guiBoutonSupprimerPersonne.js"></script>
18 <script src="/guiBoutonAjouterPersonne.js"></script>
19 <script src="/guiBoutonAjouterAdresse.js"></script>
20 <script src="/guiBoutonModifierAdresse.js"></script>
21 <script src="/guiBoutonSupprimerAdresse.js"></script>
22 <script src="/guiModifierPersonneFormValidate.js"></script>
23 <script src="/guiAjouterPersonneFormValidate.js"></script>
24 <script src="/guiAjouterAdresseFormValidate.js"></script>
25 <script src="/guiModifierAdresseFormValidate.js"></script>
26 <script src="/guiDetailsChanged.js"></script>
27 <script src="/guiPersonneChanged.js"></script>
28
29 <!-- Code HTML de la vue — Structure générale de la page HTML -->
30
31 <button id="boutonAjouterPersonne">Ajouter une personne</button><br />
32 <span id="listePersonnes" class="panel"></span>
33 <span class="panel">
34   <span id="vueDetail">
35     </span><br /><br />
36 </span>
37 <span id="spanMainForm" class="panel">
38   <form id="ajouterPersonneForm" method="post"></form>
39   <form id="modifierPersonneForm" method="post"></form>
40   <form id="ajouterAdresseForm" method="post"></form>
41   <form id="modifierAdresseForm" method="post"></form>
42 </span>
43
44 <!-- Inclusion de jQuery le plus trad possible -->
45 <script src="/jquery.js"></script>
46 <script src="/guijQueryEventsPersonne.js"></script>
47 <script src="/guijQueryEventsAdresse.js"></script>
48
49 <!-- Ajout d'un main et exécution -->
50 <script>
51   /**
52    * Série d'instructions effectuées pour initialiser l'application/
53    * @method mainFunction
54    * @augments myApp
55    */
56   myApp.addModule("mainFunction", function() {
57
58     // Personne sélectionnée par défaut
59     myApp.modele.selectedPersonne = myApp.modele.personnes[0];
60
61     // Provoquer le premier affichage de la vue :
62     myApp.gui.mediator.publish("personne/changed", {
63       personne : myApp.modele.selectedPersonne
64     });

```

```

65
66      // Enregistrement des événements utilisateurs gérés par jQuery
67      myApp.gui.initjQueryEventsPersonne();
68      myApp.gui.initjQueryEventsAdresse();
69
70  });
71
72  //////////////////////////////////////
73  // Exécution du Main avec un test d'exception
74  // try{
75      // Exécution de la méthode mainFunction
76      myApp.mainFunction();
77  // } catch (e){
78      // alert(e.message);
79  // }
80  </script>
81 </body>
82 </html>

```

5.5 Événements concernant les Adresses

5.5.1 Enregistrement des événements utilisateurs via jQuery

De même que pour les personnes, l'utilisation de jQuery est limitée à un module *Wrapper*, qui va définir tous les *handler*.

Comme il peut y avoir plusieurs adresses, dont les éléments *HTML* sont générés dynamiquement, sur le panneau des détails, les événements concernant les adresses doivent pouvoir être reconstruits dans le cas d'une reconstruction du panneau des détails de la vue (événement *personne/detailsRebuilt* du *mediator*. De plus, nous devons créer un *handler* pour chacune des adresses de la personne sélectionnée. Ces *handler* seront créés grâce à des *helpers*.

exemples/ihm/ex11_guijQueryEventsAdresse.js

```

1  /**
2   * Méthode d'initialisation des événements utilisateurs JavaScript.
3   * Enregistrement des gestionnaires de ces événements via jQuery.
4   */
5  myApp.addModule.apply(myApp.gui, [ "initjQueryEventsAdresse", function() {
6
7      /**
8       * Gestionnaire click sur le bouton faisant sortir le formulaire
9       */
10     var clickBoutonSaisieAdresse = function(event) {
11         // publication auprès du médiateur
12         myApp.gui.mediator.publish("adresse/saisie", {
13             personne : myApp.modele.selectedPersonne
14         });
15     };
16
17     /** Méthode qui permet de créer un gestionnaire d'événement de click
18      * du bouton de suppression sur chaque adresse de la personne sélectionnée.
19      * Ces gestionnaires publient l'événement "nouvelle personne sélectionnée"
20      * auprès du médiateur.

```

```

20      * @param {int} index indice de l'adresse pour lequel on enregistre l'évé
      nement.
21      */
22      var registerHelperSupprimerAdresse = function(index){
23          return function(){
24              myApp.gui.mediator.publish("adresse/delete",
25                  {
26                      personne : myApp.modele.selectedPersonne ,
27                      adresse : myApp.modele.selectedPersonne.getAdresse(index)
28                  });
29          };
30      };
31
32      /** Méthode qui permet de créer un gestionnaire d'événement de click
33      * du bouton de suppression sur chaque adresse de la personne sélectionnée.
34      * Ces gestionnaires publient l'événement "nouvelle personne sélectionnée"
35      * auprès du médiateur.
36      * @param {int} index indice de l'adresse pour lequel on enregistre l'événement
37      */
38      var registerHelperModifierAdresse = function(index){
39          return function(){
40              myApp.gui.mediator.publish("adresse/edit",
41                  {
42                      personne : myApp.modele.selectedPersonne ,
43                      adresse : myApp.modele.selectedPersonne.getAdresse(index)
44                  });
45          };
46      };
47
48      /**
49      * Enregistre les événements de clicks sur les boutons "Ajouter une adresse"
50      * et
51      * les boutons "Supprimer" ou modifier de toutes les adresses de la personne s
52      * électionnée.
53      * Cette fonction doit être invoquée en cas de sélection d'une nouvelle
54      * personne
55      * (reconstruction deu code HTML du panneau des détails).
56      */
57      var registerButtonClickEvents = function(){
58          var idBoutonSupprimerAdresse ,
59              idBoutonModifierAdresse;
60          // Enregistrement du Handler du click pour ajouter une adresse
61          $("#boutonAjouterAdresse").on("click", clickBoutonSaisieAdresse);
62
63          for (var i=0 ; i < myApp.modele.selectedPersonne.getNbAdresses() ; ++i){
64              idBoutonSupprimerAdresse = "boutonSupprimerAdresse_" +
65                  myApp.modele.selectedPersonne.getAdresse(i).getProperty('id');
66              $("##" + idBoutonSupprimerAdresse).on("click",
67                  registerHelperSupprimerAdresse(i));
68              idBoutonModifierAdresse = "boutonModifierAdresse_" +
69                  myApp.modele.selectedPersonne.getAdresse(i).getProperty('id');
70              $("##" + idBoutonModifierAdresse).on("click", registerHelperModifierAdresse
71                  (i));
72          }
73      }

```

```

68
69 ///////////////////////////////////////////////////
70 // Gestionnaire de submit formulaire d'ajout de adresse.
71
72 /**
73  * Gestionnaire de l'événement submit du formulaire.
74  * @param {Event} jQuery event correspondant au handler.
75  */
76 var formHandlerAjoutAdresse = function(event){
77
78     // Éviter d'appeler l'"action" par défaut () script PHP, etc...)
79     // du formulaire lors du submit
80     event.preventDefault();
81
82     // publication auprès du médiateur
83     myApp.gui.mediator.publish("adresse/create", {
84         personne : myApp.modele.selectedPersonne
85     });
86 } // fin du gestionnaire formHandlerAjout()
87
88 // Enregistrement du Handler du submit du formulaire via jQuery
89 $("#ajouterAdresseForm").on("submit", formHandlerAjoutAdresse);
90
91 ///////////////////////////////////////////////////
92 // Gestionnaire de submit formulaire d'ajout de adresse.
93
94 /**
95  * Gestionnaire de l'événement submit du formulaire.
96  * @param {Event} jQuery event correspondant au handler.
97  */
98 var formHandlerModifierAdresse = function(event){
99
100     // Éviter d'appeler l'"action" par défaut () script PHP, etc...)
101     // du formulaire lors du submit
102     event.preventDefault();
103
104     // publication auprès du médiateur
105     myApp.gui.mediator.publish("adresse/update", {
106         personne : myApp.modele.selectedPersonne
107     });
108 } // fin du gestionnaire formHandlerAjout()
109
110 // Enregistrement du Handler du submit du formulaire via jQuery
111 $("#modifierAdresseForm").on("submit", formHandlerModifierAdresse);
112
113 // Enregistrer les clicks lors de l'initialisation
114 registerButtonClickEvents();
115
116 // Permet à la méthode qui regénère le panneau des détails de recréer,
117 // via le médiateur, les événements "click" sur les boutons dans le panneau des
118 // détails.
119 myApp.gui.mediator.subscribe("personne/detailsRebuilt",
120     registerButtonClickEvents);
121 }]);

```

5.5.2 Boutons d'ajout, de suppression, et de modification

Le bouton d'ajout d'une adresse, qui existe un seul exemplaire car il dépend uniquement de la personne, est le plus simple. Il faut créer un formulaire vierge pour la saisie d'une adresse.

Comme pour une personne, on utilise la possibilité de passer `null` comme argument de la fabrique d'adresse, qui crée alors une adresse par défaut, sans créer d'erreurs pour les champs vides (même pour les champs obligatoires).

exemples/ihm/ex12_guiBoutonAjouterAdresse.js

```

1  /**
2   * Définition et enregistrement des callbacks appelés à gérer le clic sur le
3   * bouton
4   * "modifier" la personne sélectionnée.
5   */
6  myApp.addModule.apply(myApp.gui, [ "callbacksClickAjouter", function() {
7
8      /**
9       * Génération du code HTML du formulaire de modification de la personne sé-
10      * lectionnée.
11      */
12      var getHtmlFormInputs = function() {
13          return "<span style=\"width : 360px; display : inline-block; vertical-align :
14              top;\">\" +
15              "<p style=\"width : 360px; display : inline-block; text-align : center;
16                  padding : 15px;\">\" +
17              "<strong>Saisie d'une nouvelle adresse</strong>\" +
18              "<br />pour \" + myApp.modele.selectedPersonne.getProperty( \"nom\" ) + \"<
19              /p>\" +
20              myApp.gui.getHtmlFormInputs(myApp.metier.adresse.createInstance( null
21              ), \"ajouterAdresseForm\" ) +
22              "<label>X</label><input type=\"submit\" value=\"Valider\">X</input>\" +
23              "</span>\";
24      }
25
26      /**
27       * Callback d’Affichage (via le DOM) du formulaire dans l’élément d’ID \"
28       * mainForm\"
29       * @param {Object} contextArg non utilisé.
30       */
31      var repaintFormInputs = function(contextArg) {
32          $(\"#modifierPersonneForm\").empty(); // Vider les inputs et les événements JS
33          // existant
34          $(\"#ajouterPersonneForm\").empty(); // Vider les inputs et les événements JS
35          // existant
36          $(\"#ajouterAdresseForm\").empty(); // Vider les inputs et les événements JS
37          // existant
38          $(\"#modifierAdresseForm\").empty(); // Vider les inputs et les événements JS
39          // existant
40          $(\"#ajouterAdresseForm\").append(getHtmlFormInputs()); // ajouter les
41          // nouveaux inputs
42      };
43
44      // Enregistrement du callback
45      myApp.gui.mediator.subscribe( \"adresse/saisie\", repaintFormInputs );
46  }
47  ];

```

```
36 }()]);
```

Les boutons de modification et de suppression des adresse doivent exister en autant d'exemple qu'il y a d'adresse. On crée donc un *helper* chargé de créer le *callback* correspondant à chaque adresse.

exemples/ihm/ex13_guiBoutonModifierAdresse.js

```
1  /**
2   * Définition et enregistrement des callbacks appelés à gérer le clic sur le
      bouton
3   * "modifier" la personne sélectionnée.
4   */
5  myApp.addModule.apply(myApp.gui, [ "callbacksClickModifierPersonne", function() {
6
7      /**
8       * Génération du code HTML du formulaire de modification de la personne sé-
          lectionnée.
9       */
10     var getHtmlFormInputs = function(adresse){
11         return "<span style=\"width :360px; display : inline-block; vertical-align :
            top;\">\" +
12             myApp.gui.getHtmlFormInputs(adresse, "modifierAdresseForm") +
13             "<label></label><input type=\"submit\" value=\"Valider\"></input>\" +
14             "</span>";
15     }
16
17     /**
18      * Callback d’Affichage (via le DOM) du formulaire dans l’élément d’ID "
          modifierPersonneForm"
19      * @param {Object} contextArg non utilisé.
20      */
21     var repaintFormInputs = function(contextArg){
22         $("#modifierPersonneForm").empty(); // Vider les inputs et les événements JS
            existant
23         $("#ajouterPersonneForm").empty(); // Vider les inputs et les événements JS
            existant
24         $("#ajouterAdresseForm").empty(); // Vider les inputs et les événements JS
            existant
25         $("#modifierAdresseForm").empty(); // Vider les inputs et les événements JS
            existant
26
27         $("#modifierAdresseForm").append(getHtmlFormInputs(contextArg.adresse)); //
            ajouter les nouveaux inputs
28     };
29
30     // Enregistrement du callback
31     myApp.gui.mediator.subscribe("adresse/edit", repaintFormInputs);
32
33 }()]);
34
```

exemples/ihm/ex14_guiBoutonSupprimerAdresse.js

```
1  /**
2   * Définition et enregistrement des callbacks appelés à gérer le clic sur le
      bouton
```

```

3  * "modifier" la adresse sélectionnée.
4  */
5  myApp.addModule.apply(myApp.gui, [ "callbacksClickSupprimerAdresse", function() {
6
7      /**
8       * Callback qui supprime la adresse passée dans l'objet passé en argument.
9       * @param {Object} contextArg argument indiquant la adresse à supprimer.
10      * @param {adresse} contextArg.adresse référence de l'instance de adresse à
        supprimer dans le modèle.
11      */
12      var deleteAdresse = function(contextArg){
13
14          // Suppression de l'adresse dans la personne
15          contextArg.personne.deleteAdresse(contextArg.adresse);
16
17          // Provoquer la mise à jour de la vue :
18          myApp.gui.mediator.publish("personne/detailsChanged", {
19              adresse : myApp.modele.selectedPersonne
20          });
21      }
22
23      // Enregistrement du callback
24      myApp.gui.mediator.subscribe("adresse/delete", deleteAdresse);
25
26  }())];

```

5.5.3 Création d'une nouvelle adresse

L'adresse est automatiquement ajoutée à la personne sélectionnée, et son *ID* est généré automatiquement. Comme dans le cas d'une personne, les propriétés de l'adresse (autre que l'*ID*) sont récupérées à partir des valeurs des *inputs* du formulaire.

exemples/ihm/ex15_guiAjouterAdresseFormValidate.js

```

1  /**
2   * Définition et enregistrement du callback réagissant à la validation (submit)
3   * du formulaire de modification d'une adresse.
4   */
5  myApp.addModule.apply(myApp.gui, [ "callbacksValidateAjouterAdresseForm",
        function() {
6
7
8      /**
9       * Modifie le modèle à partir des données saisies dans le formulaire
10      */
11      var updateModel = function() {
12          // 1) Mise à jour des données du modèle
13          // à partir des valeurs des inputs du formulaire
14          var propertyName,
15              inputId;
16
17          // Ajout d'une adresse vide dans la collection
18          var nouvelleAdresse = myApp.metier.adresse.createInstance(null);
19          myApp.modele.selectedPersonne.addAdresse(nouvelleAdresse);
20

```



```

21 // Pour chaque propriété (chaque input du formulaire)
22 for (var j=0 ; j< myApp.metier.adresse.getPropertyList().length ; ++j){
23     propertyName = myApp.metier.adresse.getPropertyList()[j];
24     if (propertyName != "id"){
25         // calcul de l'ID de l'input
26         inputId = myApp.gui.getInputId({
27             propertyName : propertyName,
28             formId : "ajouterAdresseForm"
29         });
30         // Modification de la propriété de la adresse
31         // avec la valeur saisie dans l'input.
32         nouvelleAdresse.setProperty(propertyName,
33             document.getElementById(inputId).value
34         );
35     }
36 }
37
38 // Provoquer la mise à jour de la vue (panneau des détails)
39 myApp.gui.mediator.publish("personne/detailsChanged", {
40     personne : myApp.modele.selectedPersonne
41 });
42 // Provoquer la requête AJAX pour l'implémentation de la persistance
43 myApp.gui.mediator.publish("adresse/created", {
44     personne : myApp.modele.selectedPersonne,
45     adresse : nouvelleAdresse
46 });
47
48 };
49
50 // Enregistrement du callback de l'événement de validation du formulaire
51 myApp.gui.mediator.subscribe("adresse/create", updateModel);
52
53 }());

```

5.5.4 Modification d'une adresse

La modification d'une adresse après modification présente la difficulté suivante : il faut retrouver l'instance d'adresse à modifier, parmi les adresses de la personne sélectionnée. Nous avons choisi de mettre un champs caché avec l'*ID* dans le formulaire (voir la partie 4.3.2). Il nous faut alors rechercher l'*ID* de l'adresse dans les instances d'adresse de la personne sélectionnée. Nous aurions aussi pu ajouter une référence vers l'adresse éditée dans le modèle.

exemples/ihm/ex16_guiModifierAdresseFormValidate.js

```

1 /**
2  * Définition et enregistrement du callback réagissant à la validation (submit)
3  * du formulaire de modification d'une personne.
4  */
5 myApp.addModule.apply(myApp.gui, ["callbacksValidateModifierAdresseForm",
6     function(){
7         // Formulaire de modification d'une personne
8
9         /**
10          * Modifie le modèle à partir des données saisies dans le formulaire
11          */

```

```

11  var updateModel = function(){
12
13
14      // 1) Mise à jour des données du modèle
15      // à partir des valeurs des inputs du formulaire
16      var propertyName,
17          inputId;
18
19      // Recherche de l'adresse qui a été modifiée à partir de son ID unique
20      // L'ID se trouve en champs caché du formulaire.
21      var inputId_id = myApp.gui.getInputId({
22          propertyName : "id",
23          formId : "modifierAdresseForm"
24      });
25
26      // ID unique de l'adresse
27      var idAdresse = document.getElementById(inputId_id).value;
28      var adresseEnQuestion;
29      for (var i = 0 ; i < myApp.modele.selectedPersonne.getNbAdresses() ; ++i){
30          if (idAdresse == myApp.modele.selectedPersonne.getAdresse(i).getProperty(
31              'id')){
32              adresseEnQuestion = myApp.modele.selectedPersonne.getAdresse(i);
33          }
34      }
35      if (adresseEnQuestion == undefined){
36          throw new Error("Adresse introuvable (ID inexistant)");
37      }
38
39      // Pour chaque propriété (chaque input du formulaire)
40      for (var j=0 ; j< myApp.metier.adresse.getPropertyList().length ; ++j){
41          propertyName = myApp.metier.adresse.getPropertyList()[j];
42          if (propertyName != "id"){
43              // calcul de l'ID de l'input
44              inputId = myApp.gui.getInputId({
45                  propertyName : propertyName,
46                  formId : "modifierAdresseForm"
47              });
48              // Modification de la propriété de la personne
49              // avec la valeur saisie dans l'input.
50              adresseEnQuestion.setProperty(propertyName,
51                  document.getElementById(inputId).value
52              );
53          }
54      }
55      // Provoquer la mise à jour des éléments de la vue observant la personne
56      myApp.gui.mediator.publish("personne/detailsChanged", {
57          personne : null
58      });
59
60      // Provoquer la mise à jour des éléments de la vue observant la personne
61      myApp.gui.mediator.publish("adresse/changed", {
62          personne : myApp.modele.selectedPersonne,
63          adresse : adresseEnQuestion
64      });
65  };

```

```
66 |  
67 | // Enregistrement du callback de l'événement de validation du formulaire  
68 | myApp.gui.mediator.subscribe("adresse/update", updateModel);  
69 |  
70 | }() ] );
```

Chapitre 6

Requêtes Asynchrones et *API Restful*

6.1 Qu'est-ce qu'une requête asynchrone ?

Les requêtes asynchrones *XMLHttpRequest* permettent d'exécuter (suite à un événement côté client) une requête *HTTP* (exécution d'un *CGI*, par exemple en *PHP*) sur le serveur. On parle de requête *asynchrone* car le client n'est pas bloqué en attendant la réponse du serveur : le déroulement du programme côté client peut se poursuivre, et la réponse du serveur est gérée par des *callbacks*.

Malgré le nom *XMLHttpRequest*, les requêtes asynchrones permettent d'échanger avec le serveur d'autres types de données que du *XML*. Nous utiliserons dans ce cours des données *JSON*.

Le codage *JSON* permet de coder sous forme de chaîne de caractères des collections d'objets. Ainsi, on pourra, par exemple, coder en *JSON* une collection d'objets en *PHP* (tableau associatif), puis transmettre la chaîne *JSON* via une requête asynchrone, et enfin reconstituer une collection d'objets en *JavaScript* pour générer, par exemple, une mise en forme *HTML* dans le document.

Voici un exemple de code *JSON* d'un tableau associatif (qui contient lui-même un tableau de descriptions de formats) :

```
{
  "id": 654,
  "denomination": "Tutoriel JavaScript",
  "prix unitaire": 0.50,
  "formats": ["PDF", "Postscript", "HTML", "ePub"]
}
```

On peut, par exemple, générer un tel tableau en *PHP* par le code suivant :

```
1 <?php
2   $myArray = array( "id" => 654,
3     "denomination" => "Tutoriel JavaScript",
4     "prix unitaire" => 0.50,
5     "formats" => array( "PDF", "Postscript", "HTML", "ePub" ) );
6
7   echo json_encode( $myArray );
8 ?>
```

6.2 Requête Ajax

Le méthode `ajax` de *jQuery* permet d'effectuer une requête *XMLHttpRequest* qui transmet des paramètres (un objet *JavaScript*) à un *CGI* (ici en *PHP*), via une *URL*. Dans notre exemple, le serveur reçoit lui-même un objet (propriété `data`) codé en *JSON*, et génère lui-même du code *JSON*. Le programme client récupère du code *JSON* générée sur la sortie standard du *CGI*, et reconstitue un objet *JavaScript*.

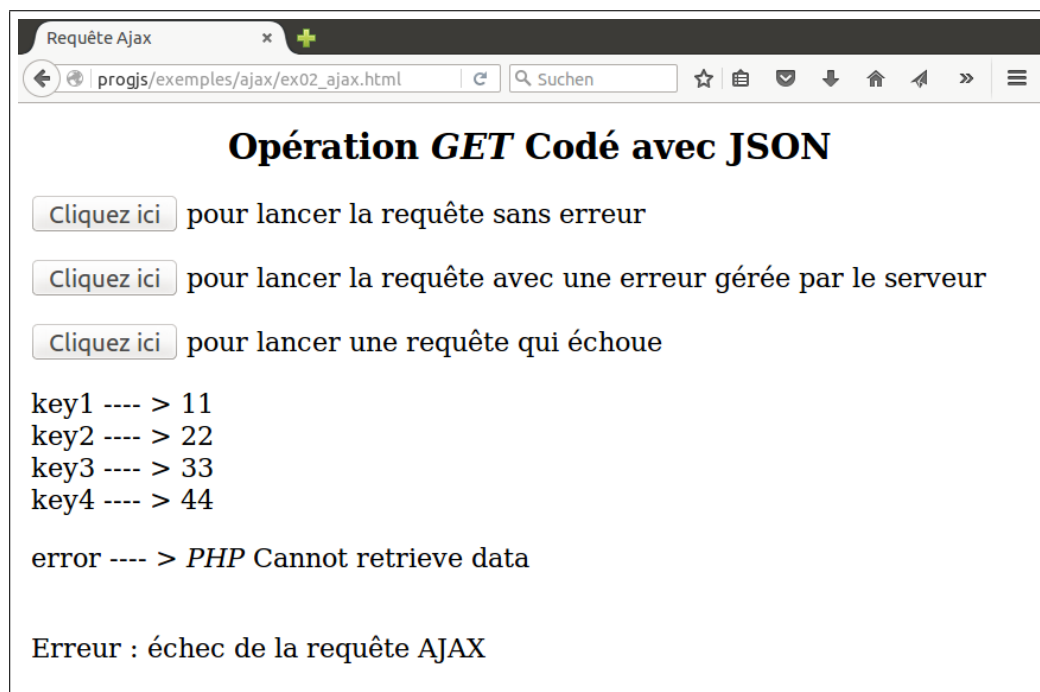
Voici notre exemple où le code *JavaScript* côté client récupère une collection d'objet créée par le *CGI* et la met en forme en *HTML*. Trois boutons permettent de tester :

- Un cas sans erreur ;
- Un cas où la gestion d'erreur est implémentée en *PHP* côté serveur ;
- Un cas où la requête *AJAX* elle-même échoue.

Les trois *callbacks* suivants sont utilisés pour gérer la requête :

- **success** en cas de succès de la requête ;
- **error** en cas d'échec de la requête
- **complete**, ici utilisé pour mettre à jour la vue, que ce soit en cas de succès ou en cas d'échec de la requête.

Le programme en *JavaScript* côté client est le suivant :



exemples/ajax/ex02_ajax.html

```
1 </!doctype html>
2 <html lang="fr">
3 <head>
```

```

4   <meta charset="utf-8">
5   <title>Requête Ajax</title>
6   <script src="/jquery.js"></script>
7   <link rel="stylesheet" href="basicStyle.css"/>
8 </head>
9
10 <body>
11 <h1>Opération <i>GET</i> Codé avec JSON</h1>
12
13 <p><button onclick="lancerRequete(1)">Cliquez ici</button> pour lancer la requête sans erreur</p>
14 <p><button onclick="lancerRequete(0)">Cliquez ici</button> pour lancer la requête avec une erreur gérée par le serveur</p>
15 <p><button onclick="lancerRequete(-1)">Cliquez ici</button> pour lancer une requête qui échoue</p>
16
17 <p id="outputParagraph"></p>
18 <script>
19
20     var model = {
21         paragraphText : "",
22         error : null,
23         getErrorMessage : function() {
24             return this.error !== null ? "<br/>" + this.error : "";
25         }
26     };
27
28     /**
29      * fonction callback exécutée en cas de succès de la requête AJAX.
30      * La méthode parcourt les données retournées par le serveur au format JSON,
31      * et concatène le texte dans le modèle.
32      * @param {Object} retrievedData : collection des données décodées à partir du JSON.
33      *
34      * La donnée peut être un message d'erreur.
35      */
36     var ajaxCallbackSuccess = function(retrievedData){
37         model.error=null;
38         model.paragraphText = "";
39         // Parcours et affichage des données de l'objet
40         for (var key in retrievedData){
41             model.paragraphText += key + " ——— > " + retrievedData[key] + '<br/>';
42         }
43     };
44
45     /**
46      * fonction callback exécutée en cas d'échec de la requête AJAX.
47      * Une erreur est ajoutée dans le modèle et le texte du paragraphe est mis à vide.
48      */
49     var ajaxCallbackError = function() {
50         model.paragraphText = "";
51         model.error = "Erreur : échec de la requête AJAX";
52     };
53
54     /**
55      * fonction callback exécutée lorsque la requête AJAX se termine.

```

```

55  * Ce callback est appelé en cas d'échec ET en cas de succès de la requête
56  * Ici, la méthode met à jour la vue en affichant le texte et une éventuelle
57  */
58  var ajaxCallbackComplete = function() {
59      $( "#outputParagraph" ).append(
60          "<p>" +
61              model.paragraphText +
62              model.getErrorMessage() +
63          "</p>");
64  }
65
66  /**
67   * Gestionnaire de click sur les boutons, qui déclenche une requête AJAX.
68   * @param {int} simpleTestValue donnée transmise au serveur via la propriété
69   *   simpleTest
70   *   si simpleTestValue est négatif, une URL du serveur inexistante
71   *   est utilisée,
72   *   provoquant l'échec de la requête (c'est juste pour l'exemple
73   *   ...).
74   */
75  var lancerRequete = function(simpleTestValue){
76
77      var urlServeur = "http://progjs/exemples/ajax/ex01_encode_json.php";
78
79      // Pour provoquer une requête qui échoue complètement
80      if (simpleTestValue < 0){ // URL qui n'existe pas
81          urlServeur = "http://progjs/exemples/ajax/bidon.php";
82      }
83
84      // Lancement d'une requête AJAX avec données (POST) codée en JSON
85      var jqxhr = $.ajax({
86          // Envoyer les données de la personne avec le format JSON
87          dataType: "json",
88          url: urlServeur, // URL du serveur
89          method: 'post', // Envoyer les données dans le tableau $_POST
90          contentType: 'application/x-www-form-urlencoded',
91          // données à transmettre au serveur
92          data : {
93              simpleTest : simpleTestValue
94          },
95          // Méthode callback qui reconstruit le modèle en cas de succès
96          success : ajaxCallbackSuccess,
97          // Méthode callback qui gère une éventuelle erreur dans la requête
98          error : ajaxCallbackError,
99          // Méthode callback qui met à jours la vue la vue en cas de succès ou
100          // d'erreur
101          complete : ajaxCallbackComplete
102      });
103  }
104
105  </script>
106 </body>

```

Le programme en PHP côté serveur est le suivant :

workspace_progWeb_2a/ajax/ex01_encode_json.php

```

1 <?php
2 if (isset($_REQUEST['simpleTest']) && $_REQUEST['simpleTest'] == 1){
3     $myArray = array('key1' => 11, 'key2' => 22, 'key3' => 33, 'key4' => 44);
4 }else{
5     $myArray = array('error' => "<i>PHP</i> Cannot retrieve data");
6 }
7
8 // Header HTTP
9 header('content-type : application/json; charset=utf-8');
10 echo json_encode($myArray);
11 ?>

```

6.3 Qu'est-ce qu'une *API REST* (ou systèmes *Restful*) ?

L'architecture *REST* (*representational state transfer*) est, dans notre cadre, une architecture d'application client-serveur, qui permet le lien entre une application côté client en *Javascript* et un serveur web sur lequel s'exécutent des *CGI*.

Côté serveur, notre application permettra :

- **Opération *GET***. De lire toutes les ressources (ici d'une table de base de données) ;
- **Opération *GET***. De lire une ressource identifiée (ici une ligne d'une table de base de données) de manière unique (par un identifiant unique) ;
- **Opération *POST***. De créer une ressource (ici une ligne d'une table de base de données) avec son identifiant unique ;
- **Opération *PUT***. De modifier une ressource identifiée (ici une ligne d'une table de base de données) de manière unique (par un identifiant unique) ;
- **Opération *DELETE***. De détruire une ressource identifiée (ici une ligne d'une table de base de données) de manière unique (par un identifiant unique) ;

En utilisant cette interface (*service web*), l'application côté client pourra accéder à la couche persistance du serveur.

Des problèmes de sécurité peuvent se poser. Aussi, les opérations ne sont généralement pas toutes accessibles à un même utilisateur. En général, l'opération *GET* ne permet pas de modifier les données et est moins sensible en matière de sécurité. On aura éventuellement recours aux mêmes techniques d'authentification des utilisateurs ou des programmes clients que dans la sécurisation des sites webs côté serveur à base de *CGI* uniquement (exemple : gestion avancée du numéro de session). **L'exemple qui suit dans ce chapitre ne gère pas les problèmes d'authentification** et ne pourra donc pas être utilisé directement en production sans réflexion sur les besoins de sécurité.

6.4 Exemple d'*API Restful*

Nous développons ici l'organisation d'une *API Restful* qui nous permettra d'implémenter la persistance pour notre application *Web* en *JavaScript* présentée au chapitre 5.

L'implémentation de l'API s'appuie sur le cours de programmation Web côté serveur sur :
<http://malgouyres.org/progweb>

6.4.1 Le *Front Controller*

Le *Front Controller* nous permet d'identifier l'action, de déterminer si l'utilisateur a des droits suffisants pour exécuter l'action, et d'appeler, en tenant compte du rôle de l'utilisateur et de l'action, le contrôleur dédié qui va implémenter l'action. La gestion des erreurs (comme l'action non définie) sera vue dans la partie 6.4.3.

exemples/clientServer/ex01_controleurFront.php

```

1 <?php
2 /**
3  * @brief La classe Controleur identifie l'action et appelle la méthode
4  * pour construire le modèle correspondant à l'action.
5  * Le controleur appelle aussi la vue correspondante.
6  * Il gère aussi les exceptions et appelle le cas échéant une vue d'erreur.
7  */
8 class ControleurFront {
9
10  /**
11   * @brief C'est dans le constructeur que le contrôleur fait son travail.
12   */
13  function __construct() {
14      try{
15          // Récupération de l'action
16          $action = isset($_REQUEST['action']) ? $_REQUEST['action'] : "";
17
18          // L'utilisateur est-il identifié ? Si oui, quel est son rôle ?
19          $modele = Authentication::restoreSession();
20          // $role = ($modele->getError() == false) ? $modele->getRole() : "";
21          // La gestion des rôles au prochain épisode...
22          $role = "admin";
23
24          // On distingue des cas d'utilisation, suivant l'action
25          switch($action){
26
27              // 1) Actions concernant l'authentification :
28              case "auth": // Vue de saisie du login/password
29              case "validateAuth": // Validation du login/password
30                  $authCtrl = new ControleurAuth($action);
31                  break;
32
33              // 2) Actions accessibles uniquement aux administrateurs :
34
35              case "adresse-update": // Met à jour une Adresse dans la BD
36              case "adresse-create": // Cratation d'une nouvelle Adresse dans la BD
37              case "adresse-delete": // Supression d'une Adresse à partir de son ID
38                  if ($role == "admin"){
39                      $adminCtrl = new ControleurAdminAdresse($action);
40                  } else {
41                      $modele = new Model(array('auth' => "Permission non accordée"));
42                      require (Config::getJsonOutput()["errorHandled"]);
43                  }
44              break;

```

```

45
46     case "personne-update": // Met à jour une Adresse dans la BD
47     case "personne-create": // Cratation d'une nouvelle Adresse dans la BD
48     case "personne-delete": // Supression d'une Adresse à partir de son ID
49         if ($role == "admin"){
50             $adminCtrl = new ControleurAdminPersonne($action);
51         }else{
52             $modele = new Model(array( 'authentication' => "Permission Denied"));
53             require (Config::getJsonOutput()[ "errorHandled"]);
54         }
55         break;
56
57     // 3) Actions accessibles aux visiteurs et aux administrateurs :
58
59     case "adresse-get": // Affichage d'une Adresse à partir de son ID
60     case "adresse-get-all": // Affichage de toutes les Adresse's
61         // Ici, l'implémentation (donc le contrôleur) dépend pas du rôle
62         $publicCtrl = new ControleurVisitorAdresse($action);
63         break;
64     case "personne-get-all": // Affichage de toutes les Personne's
65         $publicCtrl = new ControleurVisitorPersonne($action);
66         break;
67     default :
68         $modele = new Model(array( 'action' => "Action non définie (ressource(s)
69             ) introuvables"));
70         require (Config::getJsonOutput()[ "errorHandled"]);
71     }
72 }catch (Exception $e){ // Page d'erreur par défaut
73     $modele = new Model(array( 'exception' => $e->getMessage()));
74     require (Config::getJsonOutput()[ "errorHandled"]);
75 }
76 }
77 ?>

```

6.4.2 Récupération des données venant du client

Les données client sont issues d'une requête *AJAX* présentée dans les parties 6.5.2 et 6.5.2. Dans le cas d'une personnes, on a envoyé une objet dans une propriété **personne**.

exemples/clientServer/ex02_validationPersonne.php

```

1 <?php
2 // Les propriétés de la personne sont dans un tableau associatif $_REQUEST['
   // (voir la propriété data des propriétés de la requête AJAX
3
4
5 $id="";
6 if (isset($_REQUEST['personne']['id'])){
7     $id = filter_var($_REQUEST['personne']['id'], FILTER_SANITIZE_STRING);
8 }
9
10 $nom="";
11 if (isset($_REQUEST['personne']['nom'])){
12     $nom = filter_var($_REQUEST['personne']['nom'], FILTER_SANITIZE_STRING);
13 }

```

14 | ➤

Dans le cas d'une personnes, on a envoyé une objet dans une propriété `personne`, qui est de type tableau associatif.

exemples/clientServer/ex02_zValidationAdresse.php

```

1 <?php
2 // Les propriétés de la personne sont dans un tableau associatif $_REQUEST['
   adresse']
3 // (voir la propriété data des propriétés de la requête AJAX
4
5 $id="";
6 if (isset($_REQUEST['adresse'][$id])){
7     $id = filter_var($_REQUEST['adresse'][$id], FILTER_SANITIZE_STRING);
8 }
9 $idPersonne="";
10 if (isset($_REQUEST['adresse'][$idPersonne])){
11     $idPersonne = filter_var($_REQUEST['adresse'][$idPersonne],
12                             FILTER_SANITIZE_STRING);
13 }
14 $numeroRue="";
15 if (isset($_REQUEST['adresse'][$numeroRue])){
16     $numeroRue = filter_var($_REQUEST['adresse'][$numeroRue],
17                             FILTER_SANITIZE_STRING);
18 }
19 $rue="";
20 if (isset($_REQUEST['adresse'][$rue])){
21     $rue = filter_var($_REQUEST['adresse'][$rue], FILTER_SANITIZE_STRING);
22 }
23 $complementAdresse="";
24 if (isset($_REQUEST['adresse'][$complementAdresse])){
25     $complementAdresse = filter_var($_REQUEST['adresse'][$complementAdresse],
26                                     FILTER_SANITIZE_STRING);
27 }
28 $codePostal="";
29 if (isset($_REQUEST['adresse'][$codePostal])){
30     $codePostal = filter_var($_REQUEST['adresse'][$codePostal],
31                             FILTER_SANITIZE_STRING);
32 }
33 $ville="";
34 if (isset($_REQUEST['adresse'][$ville])){
35     $ville = filter_var($_REQUEST['adresse'][$ville], FILTER_SANITIZE_STRING);
36 }
37 $pays="France";
38 if (isset($_REQUEST['adresse'][$pays]) && $_REQUEST['adresse'][$pays] != "")
39 {
40     $pays = filter_var($_REQUEST['adresse'][$pays], FILTER_SANITIZE_STRING);
41 }
42 }
43 ➤

```

6.4.3 Génération des données *JSON* représentant le modèle

La classe `Config` définit les *URL* des fichiers de génération de *JSON*, pour éviter les *URL* en dûr, comme pour les vues d'un *CGI*.

exemples/clientServer/ex03_config.php

```

1 <?php
2 /**
3  * @brief Classe de configuration
4  * Donne accès aux paramètres spécifiques concernant l'application
5  * telles que les chemins vers les vues, les vues d'erreur,
6  * les hash pour les ID de sessions, etc.
7  */
8 class Config
9 {
10     /**
11      * @brief retourne le tableau des (chemins vers les) fichiers de génération
12      * de JSON
13      */
14     public static function getJsonOutput(){
15         // Racine du site
16         global $rootDirectory;
17         // Répertoire contenant les fichiers de génération de JSON
18         $jsonDirectory = $rootDirectory."json/";
19         return array(
20             "collectionPersonne" => $jsonDirectory."jsonCollectionPersonne
21             .php",
22             "instancePersonne" => $jsonDirectory."jsonInstancePersonne.php",
23             "success" => $jsonDirectory."jsonSuccess.php",
24             "errorHandled" => $jsonDirectory."jsonErrorHandled.php",
25         );
26     }
27 }
28 ?>

```

Pour chaque classe métier, un utilitaire permet de convertir les instances, ou les collections d'instances, en tableaux associatifs.

exemples/clientServer/ex05_jsonAdresseJsonUtils.php

```

1 <?php
2 /**
3  * @brief Implémente la conversion d'instances d'Adresse et de collections d'
4  * adresses
5  * vers des données sous la forme de tableaux associatifs dans le but de géné
6  * rer un
7  * codage JSON de ces données (par exemple avec la fonction json_encode())
8  */
9 class AdresseJsonUtils {
10     /**
11      * @brief retourne une représentation des attributs d'une instance sous forme
12      * de tableau associatif.
13      * @param adresse un instance d'Adresse à convertir
14      * @return la représentation des données sous forme d'array.
15      */
16     public static function instanceToArray($adresse){
17         $arrayData = array(
18             "id" => $adresse->getId(),
19             "numeroRue" => $adresse->getNumeroRue(),

```

```

18     "rue" => $adresse->getRue(),
19     "complementAdresse" => $adresse->getComplementAdresse(),
20     "codePostal" => $adresse->getCodePostal(),
21     "ville" => $adresse->getVille(),
22     "pays" => $adresse->getPays()
23 );
24
25     return $arrayData;
26 }
27
28 /**
29  * @brief retourne une représentation des attributs des instances sous forme
    de tableau associatif.
30  * @param collectionAdresse un collection d'Adresse(s) à convertir
31  * @return la représentation des données sous forme d'array.
32  */
33 public static function collectionToArray($collectionAdresses){
34     $arrayData = array();
35     foreach ($collectionAdresses as $adresse){
36         // Ajout d'un élément au tableau
37         $arrayData[] = self::instanceToArray($adresse);
38     }
39
40     return $arrayData;
41 }
42 } // end of class AdresseView
43 ?>

```

exemples/clientServer/ex04_jsonPersonneJsonUtils.php

```

1 <?php
2 /**
3  * @brief Implémente la conversion d'instances de Personne et de collections d'
    adresses
4  * vers des données sous la forme de tableaux associatifs dans le but de géné
    rer un
5  * codage JSON de ces données (par exemple avec la fonction json_encode())
6  */
7 class PersonneJsonUtils {
8
9     /**
10     * @brief retourne une représentation des attributs d'une instance sous forme
        de tableau associatif.
11     * @param adresse un instance de Personne à convertir
12     * @return la représentation des données sous forme d'array.
13     */
14     public static function instanceToArray($personne){
15         $arrayData = array(
16             "id" => $personne->getId(),
17             "nom" => $personne->getNom(),
18             "adresses" => AdresseJsonUtils::collectionToArray($personne->getAdresses())
19         );
20
21         return $arrayData;
22     }

```

```

23
24  /**
25   * @brief retourne une représentation des attributs des instances sous forme
      de tableau associatif.
26   * @param collectionAdresse un collection de Personne(s) à convertir
27   * @return la représentation des données sous forme d'array.
28   */
29  public static function collectionToArray($collectionPersonnes){
30      $arrayData = array();
31      foreach ($collectionPersonnes as $personne){
32          // Ajout d'un élément au tableau
33          $arrayData[] = self::instanceToArray($personne);
34      }
35
36      return $arrayData;
37  }
38  }
39  ?>

```

À la place des vues dans un *CGI*, un fichier génère les données *JSON* correspondant au modèle (ici une collection de personnes).

exemples/clientServer/ex06_jsonCollectionPersonne.php

```

1  <?php
2      $arrayData = array("error" => null, "data" => PersonneJsonUtils::
      collectionToArray($modele->getData()));
3
4      header('content-type : application/json; charset=utf-8');
5      echo json_encode($arrayData);
6  ?>

```

Un fichier spécifique permet de renvoyer vers le client les messages correspondant aux erreurs détectées par le serveur (erreurs d'accès au serveur de base de données, données de forme incorrecte, etc.)

exemples/clientServer/ex06_jsonErrorHandled.php

```

1  <?php
2      // On retourne le tableau associatif des erreurs
3      header('content-type : application/json; charset=utf-8');
4      echo json_encode(array("error" => $modele->getError(), "data" => array()));
5  ?>

```

Un autre fichier permet, dans le cas où aucune donnée n'est attendue du client (comme par exemple la suppression d'une personne) d'indiquer qu'aucune erreur n'a été détectée.

exemples/clientServer/ex06_jsonSuccess.php

```

1  <?php
2      // On retourne une erreur null et un objet dada vide
3      header('content-type : application/json; charset=utf-8');
4      echo json_encode(array("error" => null, "data" => array()));
5  ?>

```

6.4.4 Implémentation des actions des contrôleurs

exemples/clientServer/ex07_ControlleurVisitorPersonne.php

```

1 <?php
2 /**
3  * @brief La classe Controleur identifie l'action et appelle la méthode
4  * pour construire le modèle correspondant à l'action.
5  * Le controleur appelle aussi la vue correspondante.
6  * Il gère aussi les exceptions et appelle le cas échéant une vue d'erreur.
7  */
8 class ControleurVisitorPersonne {
9
10  /**
11   * @brief C'est dans le constructeur que le contrôleur fait son travail.
12   */
13  function __construct($action) {
14      // On distingue des cas d'utilisation, suivant l'action
15      switch($action){
16          case "personne-get": // Affichage d'une Personne à partir de son ID
17              $this->actionGet();
18              break;
19          case "personne-get-all": // Affichage de toutes les Personne's
20              $this->actionGetAll();
21              break;
22          default: // L'action indéfinie (page par défaut, ici accueil)
23              require (Config::getVues()["default"]);
24              break;
25      }
26  }
27
28  /**
29   * @brief Implemente l'action "get-all" (récupère toutes les instances)
30   */
31  private function actionGetAll(){
32      $modele = ModelCollectionPersonne::getModelPersonneAll();
33      if ($modele->getError() == false){
34          require (Config::getJsonOutput()["collectionPersonne"]);
35      }else{
36          require (Config::getJsonOutput()["errorHandled"]);
37      }
38  }
39
40  /**
41   * @brief Implemente l'action "get" (récupère une instance à partir de ID)
42   */
43  private function actionGet(){
44      // ID de l'instance à récupérer
45      $rawId = isset($_REQUEST['id']) ? $_REQUEST['id'] : "";
46      $id = filter_var($rawId, FILTER_SANITIZE_STRING);
47      $modele = ModelPersonne::getModelPersonne($id);
48      if ($modele->getError() == false){
49          require (Config::getJsonOutput()["instancePersonne"]);
50      }else{
51          require (Config::getJsonOutput()["errorHandled"]);
52      }
53  }

```

```

54 }
55 ?>

```

exemples/clientServer/ex08_ControleurAdminPersonne.php

```

1 <?php
2 /**
3  * @brief La classe Controleur identifie l'action et appelle la méthode
4  * pour construire le modèle correspondant à l'action.
5  * Le controleur appelle aussi la vue correspondante.
6  * Il gère aussi les exceptions et appelle le cas échéant une vue d'erreur.
7  */
8 class ControleurAdminPersonne {
9
10  /**
11   * @brief C'est dans le constructeur que le contrôleur fait son travail.
12   */
13  function __construct($action) {
14      // On distingue des cas d'utilisation, suivant l'action
15      switch($action){
16          case "personne-update": // Met à jour une Adresse dans la BD
17              $this->actionUpdate();
18              break;
19          case "personne-create": // Cration d'une nouvelle Adresse dans la BD
20              $this->actionCreate();
21              break;
22          case "personne-delete": // Supression d'une Adresse à partir de son ID
23              $this->actionDelete();
24              break;
25          default : // L'action indéfinie (page par défaut, ici accueil)
26              $modele = new Model(array( 'action' => "Action non définie (ressource(s)
                introuvables)"));
27              require (Config::getJsonOutput()[ "errorHandled"] );
28              break;
29      }
30  }
31
32  /**
33   * @brief Implemente l'action "update" (met à jour une instance dans la BD)
34   */
35  private function actionUpdate(){
36      // valider ou nettoyer les inputs (par exemple : filter_var)
37      require (dirname(__FILE__). "/validationPersonne.php");
38      $modele = ModelPersonne::getModelPersonneUpdate($id, $nom);
39      if ($modele->getError() == false){
40          require (Config::getJsonOutput()[ "success"] );
41      }else{
42          require (Config::getJsonOutput()[ "errorHandled"] );
43      }
44  }
45
46  /**
47   * @brief Implemente l'action "create" (créé une instance dans la BD)
48   */
49  private function actionCreate(){
50      // valider ou nettoyer les inputs (par exemple : filter_var)

```



```

51     require (dirname(__FILE__). "/validationPersonne.php");
52     $modele = ModelPersonne::getModelPersonneCreate($id, $nom);
53     if ($modele->getError() == false){
54         require (Config::getJsonOutput()["success"]);
55     }else{
56         require (Config::getJsonOutput()["errorHandled"]);
57     }
58 }
59
60 /**
61  * @brief Implemente l'action "delete" (supprime une instance à partir de son
62  * ID)
63  */
64 private function actionDelete(){
65     // ID de l'instance à supprimer
66     $id = filter_var($_REQUEST['id'], FILTER_SANITIZE_STRING);
67     $modele = ModelPersonne::deletePersonne($id);
68     if ($modele->getError() == false){
69         require (Config::getJsonOutput()["success"]);
70     }else{
71         require (Config::getJsonOutput()["errorHandled"]);
72     }
73 }
74 }
75 ?>

```

exemples/clientServer/ex09_ControleurAdminAdresse.php

```

1 <?php
2 /**
3  * @brief La classe Controleur identifie l'action et appelle la méthode
4  * pour construire le modèle correspondant à l'action.
5  * Le controleur appelle aussi la vue correspondante.
6  * Il gère aussi les exceptions et appelle le cas échéant une vue d'erreur.
7  */
8 class ControleurAdminAdresse {
9
10     /**
11      * @brief C'est dans le constructeur que le contrôleur fait son travail.
12      */
13     function __construct($action) {
14         // On distingue des cas d'utilisation, suivant l'action
15         switch($action){
16             case "adresse-update": // Met à jour une Adresse dans la BD
17                 $this->actionUpdate();
18                 break;
19             case "adresse-create": // Cratation d'une nouvelle Adresse dans la BD
20                 $this->actionCreate();
21                 break;
22             case "adresse-delete": // Supression d'une Adresse à partir de son ID
23                 $this->actionDelete();
24                 break;
25             default : // L'action indéfinie (page par défaut, ici accueil)
26                 require (Config::getJsonOutput()["errorHandled"]);
27                 break;

```

```

28     }
29 }
30
31 /**
32  * @brief Implemente l'action "update" (met à jour une instance dans la BD)
33  */
34 private function actionUpdate(){
35     // valider ou nettoyer les inputs (par exemple : filter_var)
36     require (dirname(__FILE__)."/validationAdresse.php");
37     $modele = ModelAdresse::getModelAdresseUpdate($id, $idPersonne, $numeroRue,
38         $rue,
39         $complementAdresse, $codePostal, $ville, $pays);
40     if ($modele->getError() == false){
41         require (Config::getJsonOutput()["success"]);
42     }else{
43         require (Config::getJsonOutput()["errorHandled"]);
44     }
45 }
46
47 /**
48  * @brief Implemente l'action "create" (créé une instance dans la BD)
49  */
50 private function actionCreate(){
51     // valider ou nettoyer les inputs (par exemple : filter_var)
52     require (dirname(__FILE__)."/validationAdresse.php");
53     $modele = ModelAdresse::getModelAdresseCreate($id, $idPersonne, $numeroRue,
54         $rue,
55         $complementAdresse, $codePostal, $ville, $pays);
56     if ($modele->getError() == false){
57         require (Config::getJsonOutput()["success"]);
58     }else{
59         require (Config::getJsonOutput()["errorHandled"]);
60     }
61 }
62
63 /**
64  * @brief Implemente l'action "delete" (supprime une instance à partir de son
65     ID)
66  */
67 private function actionDelete(){
68     // ID de l'instance à supprimer
69     $id = filter_var($_REQUEST['id'], FILTER_SANITIZE_STRING);
70     $modele = ModelAdresse::deleteAdresse($id);
71     if ($modele->getError() == false){
72         require (Config::getJsonOutput()["success"]);
73     }else{
74         require (Config::getJsonOutput()["errorHandled"]);
75     }
76 }
77 }
78 }
79 }
80 }
81 }
82 }
83 }
84 }
85 }
86 }
87 }
88 }
89 }
90 }
91 }
92 }
93 }
94 }
95 }
96 }
97 }
98 }
99 }
100 }
101 }
102 }
103 }
104 }
105 }
106 }
107 }
108 }
109 }
110 }
111 }
112 }
113 }
114 }
115 }
116 }
117 }
118 }
119 }
120 }
121 }
122 }
123 }
124 }
125 }
126 }
127 }
128 }
129 }
130 }
131 }
132 }
133 }
134 }
135 }
136 }
137 }
138 }
139 }
140 }
141 }
142 }
143 }
144 }
145 }
146 }
147 }
148 }
149 }
150 }
151 }
152 }
153 }
154 }
155 }
156 }
157 }
158 }
159 }
160 }
161 }
162 }
163 }
164 }
165 }
166 }
167 }
168 }
169 }
170 }
171 }
172 }
173 }
174 }
175 }
176 }
177 }
178 }
179 }
180 }
181 }
182 }
183 }
184 }
185 }
186 }
187 }
188 }
189 }
190 }
191 }
192 }
193 }
194 }
195 }
196 }
197 }
198 }
199 }
200 }
201 }
202 }
203 }
204 }
205 }
206 }
207 }
208 }
209 }
210 }
211 }
212 }
213 }
214 }
215 }
216 }
217 }
218 }
219 }
220 }
221 }
222 }
223 }
224 }
225 }
226 }
227 }
228 }
229 }
230 }
231 }
232 }
233 }
234 }
235 }
236 }
237 }
238 }
239 }
240 }
241 }
242 }
243 }
244 }
245 }
246 }
247 }
248 }
249 }
250 }
251 }
252 }
253 }
254 }
255 }
256 }
257 }
258 }
259 }
260 }
261 }
262 }
263 }
264 }
265 }
266 }
267 }
268 }
269 }
270 }
271 }
272 }
273 }
274 }
275 }
276 }
277 }
278 }
279 }
280 }
281 }
282 }
283 }
284 }
285 }
286 }
287 }
288 }
289 }
290 }
291 }
292 }
293 }
294 }
295 }
296 }
297 }
298 }
299 }
300 }
301 }
302 }
303 }
304 }
305 }
306 }
307 }
308 }
309 }
310 }
311 }
312 }
313 }
314 }
315 }
316 }
317 }
318 }
319 }
320 }
321 }
322 }
323 }
324 }
325 }
326 }
327 }
328 }
329 }
330 }
331 }
332 }
333 }
334 }
335 }
336 }
337 }
338 }
339 }
340 }
341 }
342 }
343 }
344 }
345 }
346 }
347 }
348 }
349 }
350 }
351 }
352 }
353 }
354 }
355 }
356 }
357 }
358 }
359 }
360 }
361 }
362 }
363 }
364 }
365 }
366 }
367 }
368 }
369 }
370 }
371 }
372 }
373 }
374 }
375 }
376 }
377 }
378 }
379 }
380 }
381 }
382 }
383 }
384 }
385 }
386 }
387 }
388 }
389 }
390 }
391 }
392 }
393 }
394 }
395 }
396 }
397 }
398 }
399 }
400 }
401 }
402 }
403 }
404 }
405 }
406 }
407 }
408 }
409 }
410 }
411 }
412 }
413 }
414 }
415 }
416 }
417 }
418 }
419 }
420 }
421 }
422 }
423 }
424 }
425 }
426 }
427 }
428 }
429 }
430 }
431 }
432 }
433 }
434 }
435 }
436 }
437 }
438 }
439 }
440 }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }
449 }
450 }
451 }
452 }
453 }
454 }
455 }
456 }
457 }
458 }
459 }
460 }
461 }
462 }
463 }
464 }
465 }
466 }
467 }
468 }
469 }
470 }
471 }
472 }
473 }
474 }
475 }
476 }
477 }
478 }
479 }
480 }
481 }
482 }
483 }
484 }
485 }
486 }
487 }
488 }
489 }
490 }
491 }
492 }
493 }
494 }
495 }
496 }
497 }
498 }
499 }
500 }
501 }
502 }
503 }
504 }
505 }
506 }
507 }
508 }
509 }
510 }
511 }
512 }
513 }
514 }
515 }
516 }
517 }
518 }
519 }
520 }
521 }
522 }
523 }
524 }
525 }
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }
541 }
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 }
551 }
552 }
553 }
554 }
555 }
556 }
557 }
558 }
559 }
560 }
561 }
562 }
563 }
564 }
565 }
566 }
567 }
568 }
569 }
570 }
571 }
572 }
573 }
574 }
575 }
576 }
577 }
578 }
579 }
580 }
581 }
582 }
583 }
584 }
585 }
586 }
587 }
588 }
589 }
590 }
591 }
592 }
593 }
594 }
595 }
596 }
597 }
598 }
599 }
600 }
601 }
602 }
603 }
604 }
605 }
606 }
607 }
608 }
609 }
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
1000 }

```

6.5 Persistance avec *AJAX*

6.5.1 Construction du modèle à partir de la base de données

exemples/clientServer/ex10_persistenceRead.js

```

1  /**
2   * Définition et enregistrement des callbacks de chargement du modèle
3   * à partir des données sur le serveur par une requête AJAX.
4   * Permet le chargement du modèle à partir de la base de données.
5   */
6  myApp.addModule.apply(myApp.gui, [ "callbacksRebuildModelFromServer", function() {
7
8   /**
9    * Informe l'utilisateur de l'ensemble des erreurs détectées et renvoyées par
10   le serveur.
11   * @param {Object} dataError couples clef/valeur, où la clef est une catégorie
12   d'erreur
13   * (ou un champs de formulaire), et la valeur un message d'
14   erreur.
15   */
16   var alertErrorMessages = function(dataError){
17     var concatErrorMsg="";
18     if (dataError !== null){
19       for (var key in dataError){
20         if (dataError.hasOwnProperty(key)){
21           concatErrorMsg += key + ": " + dataError[key] + "\n";
22         }
23       }
24       alert(concatErrorMsg);
25     }
26   };
27
28   /**
29   * Méthode callback qui est appelée en cas de succès de la requête AJAX.
30   * Cette méthode reconstruit le modèle à partir des données du serveur.
31   * @param {Object} retrievedData données reçues du serveur (après parsing du
32   JSON)
33   * @param {Object/null} retrievedData.error null en l'absence d'erreur détecté
34   e par le serveur,
35   * ou un objet dont les propriétés sont les messages d'erreur renvoyé
36   es par le serveur.
37   * @param {Object} retrievedData.data données renvoyées par le serveur :
38   * collections d'objets permettant de construire des personnes
39   , avec leurs adresses.
40   */
41   var ajaxCallbackSuccess = function(retrievedData){
42     var addressesData, adresseInstance;
43
44     // Si aucune erreur n'a été détectée sur le serveur
45     if (retrievedData["error"] === null && retrievedData['data'] !== undefined){
46       // Parcours des objets dans les données
47       for (var key in retrievedData['data']){
48         if (retrievedData['data'].hasOwnProperty(key)){
49           // Création d'une personne sans adresse
50           var newPersonne = myApp.metier.personne.createInstance({

```

```

44         id : retrievedData [ 'data ' ][ key ][ "id" ],
45         nom : retrievedData [ 'data ' ][ key ][ "nom" ]
46     });
47
48     // Parcours des objets définissant les adresses
49     adressesData = retrievedData [ 'data ' ][ key ][ "adresses" ];
50     for (var keyAdresse in adressesData) {
51         if (adressesData.hasOwnProperty(keyAdresse)) {
52             // Création et ajout d'une adresse
53             adresseInstance = myApp.metier.adresse.createInstance({
54                 id : adressesData[keyAdresse][ "id" ],
55                 numeroRue : adressesData[keyAdresse][ "numeroRue" ],
56                 rue : adressesData[keyAdresse][ "rue" ],
57                 complementAdresse : adressesData[keyAdresse][ "
                    complementAdresse" ],
58                 codePostal : adressesData[keyAdresse][ "codePostal" ],
59                 ville : adressesData[keyAdresse][ "ville" ],
60                 pays : adressesData[keyAdresse][ "pays" ]
61             });
62             newPersonne.addAdresse(adresseInstance);
63         }
64     }
65     myApp.modele.personnes.push(newPersonne); // ajout dans le modèle
66 }
67 }
68 } else {
69     alertErrorMessages(retrievedData [ "error" ]);
70 }
71 };
72
73 /**
74  * Méthode appelée lorsque la requête AJAX se termine,
75  * que ce soit après une erreur ou après un succès.
76  * Cette méthode reconstruit la vue (après reconstruction du modèle).
77  */
78 var ajaxCallbackComplete = function(retrievedData) {
79
80     // Personne sélectionnée par défaut
81     myApp.modele.selectedPersonne = myApp.modele.personnes[0];
82
83     // La vue est réinitialisée : on vide les éléments et événements
84     $("#listePersonnes").empty();
85     $("#vueDetail").empty();
86     $("#ajouterPersonneForm").empty();
87     $("#modifierPersonneForm").empty();
88     $("#ajouterAdresseForm").empty();
89     $("#modifierAdresseForm").empty();
90
91     // Provoquer le premier affichage de la vue :
92     myApp.gui.mediator.publish("personne/changed", {
93         personne : myApp.modele.selectedPersonne
94     });
95
96     // Enregistrement des événements utilisateurs gérés par jQuery
97     myApp.gui.initJQueryEventsPersonne();
98     myApp.gui.initJQueryEventsAdresse();

```

```

99     };
100
101     /**
102      * Méthode appelée en cas d'erreur de la requête AJAX elle même.
103      */
104     var ajaxCallbackError = function(retrievedData){
105         alert("Erreur : échec de la requête ajax");
106     }
107
108     /**
109      * Callback appelé lors de l'événement "personne/read" du médiateur.
110      * Effectue une requête AJAX pour récupérer toutes les personnes
111      * pour reconstruire le modèle de données.
112      */
113     var readAllPersonne = function(){
114         // requête AJAX get codé en JSON
115         var jqxhr = $.ajax({
116             // Envoyer les données de la personne avec le format JSON
117             dataType: "json",
118             url: "http://progjs/exemples/clientServer/api/", // URL du serveur
119             method: 'post', // Envoyer les données dans le tableau $_POST
120             contentType: 'application/x-www-form-urlencoded',
121             // données à transmettre au serveur
122             data: {
123                 action: "personne-get-all"
124             },
125             // Méthode callback qui reconstruit le modèle en cas de succès
126             success: ajaxCallbackSuccess,
127             // Méthode callback qui gère une éventuelle erreur dans la requête
128             error: ajaxCallbackError,
129             // Méthode callback qui met à jours la vue la vue en cas de succès ou d'
130             // erreur
131             complete: ajaxCallbackComplete
132         });
133     };
134
135     // Enregistrement du callback de l'événement de reconstruction du modèle
136     myApp.gui.mediator.subscribe("personne/read", readAllPersonne);
137
138     }());

```

exemples/clientServer/ex10_index.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4     <meta charset="UTF-8" />
5     <title>Application interactive</title>
6     <link rel="stylesheet" href="basicStyle.css"/>
7 </head>
8 <body>
9     <script src="./modulesMetierPrototype.js"></script>
10    <script src="./modulesIHM.js"></script>
11    <script src="./persistanceRead.js"></script>
12    <script src="./persistanceCreatePersonne.js"></script>

```

```

13 <script src="./persistenceDeletePersonne.js">/script>
14 <script src="./persistenceUpdatePersonne.js">/script>
15 <script src="./persistenceCreateAdresse.js">/script>
16 <script src="./persistenceUpdateAdresse.js">/script>
17 <script src="./persistenceDeleteAdresse.js">/script>
18 <!-- Code HTML de la vue — Structure générale de la page HTML —>
19
20 <button id="boutonAjouterPersonne">Ajouter une personne</button><br/>
21 <span id="listePersonnes" class="panel">/span>
22 <span class="panel">
23   <span id="vueDetail">
24     </span><br/><br/>
25 </span>
26 <span id="spanMainForm" class="panel">
27   <form id="ajouterPersonneForm" method="post">/form>
28   <form id="modifierPersonneForm" method="post">/form>
29   <form id="ajouterAdresseForm" method="post">/form>
30   <form id="modifierAdresseForm" method="post">/form>
31 </span>
32
33 <!-- Inclusion de jQuery le plus trad possible —>
34 <script src="./jquery.js">/script>
35 <script src="./guijQueryEventsPersonne.js">/script>
36 <script src="./guijQueryEventsAdresse.js">/script>
37
38 <!-- Ajout d'un main et exécution —>
39 <script>
40   /**
41    * Série d'instructions effectuées pour initialiser l'application/
42    * @method mainFunction
43    * @augments myApp
44    */
45   myApp.addModule( "mainFunction", function() {
46
47     myApp.addModule.apply(myApp, [ "modele", {
48       selectedPersonne : null,
49       personnes : [],
50     }]);
51
52
53     // Charger le modèle :
54     myApp.gui.mediator.publish( "personne/read", {
55       personne : myApp.modele.selectedPersonne
56     });
57   });
58
59   //////////////////////////////////////////
60   // Exécution du Main avec un test d'exception
61   // try{
62     // Exécution de la méthode mainFunction
63     myApp.mainFunction();
64   // } catch (e){
65     // alert(e.message);
66   // }
67 </script>
68 </body>

```

69 </html>

6.5.2 Création, Mise à jour, et suppression des personnes

exemples/clientServer/ex11_persistenceCreatePersonne.js

```

1  /**
2   * Définition et enregistrement des callbacks de création d'une personne
3   * sur le serveur par requête AJAX.
4   */
5  myApp.addModule.apply(myApp.gui, [ "callbacksCreatePersonneQueryServer", function
6    () {
7
8    /**
9     * Méthode callback qui est appelée en cas de succès de la requête AJAX.
10    * Cette méthode informe simplement l'utilisateur des éventuelles erreurs.
11    * En effet, la requête n'est pas supposée retourner des données.
12    */
13    var ajaxCallbackSuccess = function(retrievedData){
14      var concatErrorMsg="";
15      if (retrievedData["error"] !== null){
16        for (var key in retrievedData['error']){
17          if (retrievedData['error'].hasOwnProperty(key)){
18            concatErrorMsg += key + ": " + retrievedData['error'][key] + "\n";
19          }
20        }
21        alert(concatErrorMsg);
22      }
23    };
24
25    /**
26     * Méthode appelée en cas d'erreur de la requête AJAX elle même.
27     */
28    var ajaxCallbackError = function(retrievedData){
29      alert("Erreur : échec de la requête ajax");
30    };
31
32    /**
33     * Callback appelé lors de l'événement "personne/read" du médiateur.
34     * Effectue une requête AJAX pour récupérer toutes les personnes
35     * pour reconstruire le modèle de données.
36     */
37    var createPersonne = function(contextArg){
38      // requête AJAX get codé en JSON
39      var jqxhr = $.ajax({
40        dataType: "json", // On envoie les données la personne codée en JSON
41        url: "http://progjs/exemples/clientServer/api/", // URL du serveur
42        method: 'post', // Envoyer les données dans le tableau $_POST
43        contentType: 'application/x-www-form-urlencoded',
44        // données à transmettre au serveur
45        data : {
46          action: "personne-create",
47          personne: { // Propriétés de la personne
48            id: contextArg.personne.getProperty("id"),

```

```

49         nom : contextArg.personne.getProperty("nom")
50     },
51 },
52 // Méthode callback qui reconstruit le modèle en cas de succès
53 success : ajaxCallbackSuccess,
54 // Méthode callback qui gère une éventuelle erreur dans la requête
55 error : ajaxCallbackError
56 });
57
58 };
59 // Enregistrement du callback de l'événement de mise à jour de la personne
60 myApp.gui.mediator.subscribe("personne/created", createPersonne);
61
62 }());

```

exemples/clientServer/ex12_persistenceUpdatePersonne.js

```

1  /**
2   * Définition et enregistrement des callbacks de modification d'une personne
3   * sur le serveur par requête AJAX.
4   */
5  myApp.addModule.apply(myApp.gui, ["callbacksUpdatePersonneQueryServer", function
6      () {
7      /**
8       * Méthode callback qui est appelée en cas de succès de la requête AJAX.
9       * Cette méthode informe simplement l'utilisateur des éventuelles erreurs.
10      * En effet, la requête n'est pas supposée retourner des données.
11      */
12      var ajaxCallbackSuccess = function(retrievedData){
13          var concatErrorMsg="";
14          if (retrievedData["error"] !== null){
15              for (var key in retrievedData['error']){
16                  if (retrievedData['error'].hasOwnProperty(key)){
17                      concatErrorMsg += key + ": " + retrievedData['error'][key] + "\n";
18                  }
19              }
20              alert(concatErrorMsg);
21          }
22      };
23
24      /**
25       * Méthode appelée en cas d'erreur de la requête AJAX elle même.
26       */
27      var ajaxCallbackError = function(retrievedData){
28          alert("Erreur : échec de la requête ajax");
29      };
30
31      /**
32       * Callback appelé lors de l'événement "personne/read" du médiateur.
33       * Effectue une requête AJAX pour récupérer toutes les personnes
34       * pour reconstruire le modèle de données.
35       */
36      var updatePersonne = function(contextArg){
37          // requête AJAX get codé en JSON

```



```

39     var jqxhr = $.ajax({
40         dataType: "json", // On envoie les données la personne codée en JSON
41         url: "http://progjs/exemples/clientServer/api/", // URL du serveur
42         method: 'post', // Envoyer les données dans le tableau $_POST
43         contentType: 'application/x-www-form-urlencoded',
44         // données à transmettre au serveur
45         data : {
46             action: "personne-update",
47             personne: { // Propriétés de la personne
48                 id: contextArg.personne.getProperty("id"),
49                 nom: contextArg.personne.getProperty("nom")
50             }
51         },
52         // Méthode callback qui reconstruit le modèle en cas de succès
53         success: ajaxCallbackSuccess,
54         // Méthode callback qui gère une éventuelle erreur dans la requête
55         error: ajaxCallbackError
56     });
57
58 };
59 // Enregistrement du callback de l'événement de mise à jour de la personne
60 myApp.gui.mediator.subscribe("personne/update", updatePersonne);
61
62 }());

```

exemples/clientServer/ex13_persistenceDeletePersonne.js

```

1  /**
2   * Définition et enregistrement des callbacks de suppression d'une personne
3   * sur le serveur par requête AJAX.
4   */
5  myApp.addModule.apply(myApp.gui, ["callbacksDeletePersonneQueryServer", function
6      () {
7
8      /**
9       * Méthode appelée en cas d'erreur de la requête AJAX elle même.
10      */
11      var ajaxCallbackError = function(retrievedData){
12          alert("Erreur : échec de la requête ajax");
13      };
14
15      /**
16       * Callback appelé lors de l'événement "personne/read" du médiateur.
17       * Effectue une requête AJAX pour récupérer toutes les personnes
18       * pour reconstruire le modèle de données.
19       */
19      var deletePersonne = function(contextArg){
20
21          // requête AJAX get codé en JSON
22          var jqxhr = $.ajax({
23              dataType: "json", // On envoie les données la personne codée en JSON
24              url: "http://progjs/exemples/clientServer/api/", // URL du serveur
25              method: 'post', // Envoyer les données dans le tableau $_POST
26              contentType: 'application/x-www-form-urlencoded',
27              // données à transmettre au serveur
28              data : {

```

```

29         action : "personne-delete",
30         id : contextArg.personne.getProperty("id"),
31     },
32     // Méthode callback qui gère une éventuelle erreur dans la requête
33     error : ajaxCallbackError
34 });
35
36 };
37 // Enregistrement du callback de l'événement de mise à jour de la personne
38 myApp.gui.mediator.subscribe("personne/delete", deletePersonne);
39
40 }());

```

6.5.3 Création, Mise à jour, et suppression des adresses

exemples/clientServer/ex14_persistenceCreateAdresse.js

```

1  /**
2   * Définition et enregistrement des callbacks de création d'une personne
3   * sur le serveur par requête AJAX.
4   */
5  myApp.addModule.apply(myApp.gui, ["callbacksCreateAdresseQueryServer", function
6      () {
7      /**
8       * Méthode callback qui est appelée en cas de succès de la requête AJAX.
9       * Cette méthode informe simplement l'utilisateur des éventuelles erreurs.
10      * En effet, la requête n'est pas supposée retourner des données.
11      */
12      var ajaxCallbackSuccess = function(retrievedData){
13          var concatErrorMsg="";
14          if (retrievedData["error"] !== null){
15              for (var key in retrievedData['error']){
16                  if (retrievedData['error'].hasOwnProperty(key)){
17                      concatErrorMsg += key + ": " + retrievedData['error'][key] + "\n";
18                  }
19              }
20              alert(concatErrorMsg);
21          }
22      };
23
24      /**
25       * Méthode appelée en cas d'erreur de la requête AJAX elle même.
26       */
27      var ajaxCallbackError = function(retrievedData){
28          alert("Erreur : échec de la requête ajax");
29      };
30
31      /**
32       * Callback appelé lors de l'événement "personne/read" du médiateur.
33       * Effectue une requête AJAX pour récupérer toutes les personnes
34       * pour reconstruire le modèle de données.
35       */
36      var createAdresse = function(contextArg){
37

```

```

38 // requête AJAX get codé en JSON
39 var jqxhr = $.ajax({
40     dataType: "json", // On envoie les données la personne codée en JSON
41     url: "http://progjs/exemples/clientServer/api/", // URL du serveur
42     method: 'post', // Envoyer les données dans le tableau $_POST
43     contentType: 'application/x-www-form-urlencoded',
44     // données à transmettre au serveur
45     data : {
46         action: "adresse-create",
47         adresse: { // Propriétés de l'adresse
48             id: contextArg.adresse.getProperty("id"),
49             idPersonne : contextArg.personne.getProperty("id"),
50             numeroRue: contextArg.adresse.getProperty("numeroRue"),
51             rue: contextArg.adresse.getProperty("rue"),
52             codePostal: contextArg.adresse.getProperty("codePostal"),
53             ville: contextArg.adresse.getProperty("ville"),
54             pays: contextArg.adresse.getProperty("pays")
55         }
56     },
57     // Méthode callback qui reconstruit le modèle en cas de succès
58     success: ajaxCallbackSuccess,
59     // Méthode callback qui gère une éventuelle erreur dans la requête
60     error: ajaxCallbackError
61 });
62
63 };
64 // Enregistrement du callback de l'événement de mise à jour de la personne
65 myApp.gui.mediator.subscribe("adresse/created", createAdresse);
66
67 }());

```

exemples/clientServer/ex15_persistenceUpdateAdresse.js

```

1 /**
2  * Définition et enregistrement des callbacks de création d'une personne
3  * sur le serveur par requête AJAX.
4  */
5 myApp.addModule.apply(myApp.gui, [ "callbacksUpdateAdresseQueryServer", function
6     () {
7         /**
8          * Méthode callback qui est appelée en cas de succès de la requête AJAX.
9          * Cette méthode informe simplement l'utilisateur des éventuelles erreurs.
10         * En effet, la requête n'est pas supposée retourner des données.
11         */
12         var ajaxCallbackSuccess = function(retrievedData){
13             var concatErrorMsg="";
14             if (retrievedData["error"] !== null){
15                 for (var key in retrievedData['error']){
16                     if (retrievedData['error'].hasOwnProperty(key)){
17                         concatErrorMsg += key + ": " + retrievedData['error'][key] + "\n";
18                     }
19                 }
20                 alert(concatErrorMsg);
21             }
22         };

```

```

23
24  /**
25   * Méthode appelée en cas d'erreur de la requête AJAX elle même.
26   */
27  var ajaxCallbackError = function(retrievedData){
28      alert("Erreur : échec de la requête ajax");
29  };
30
31  /**
32   * Callback appelé lors de l'événement "personne/read" du médiateur.
33   * Effectue une requête AJAX pour récupérer toutes les personnes
34   * pour reconstruire le modèle de données.
35   */
36  var updateAdresse = function(contextArg){
37
38      // requête AJAX get codé en JSON
39      var jqxhr = $.ajax({
40          dataType: "json", // On envoie les données la personne codée en JSON
41          url: "http://progjs/exemples/clientServer/api/", // URL du serveur
42          method: 'post', // Envoyer les données dans le tableau $_POST
43          contentType: 'application/x-www-form-urlencoded',
44          // données à transmettre au serveur
45          data : {
46              action: "adresse-update",
47              adresse: { // Propriétés de l'adresse
48                  id: contextArg.adresse.getProperty("id"),
49                  idPersonne : contextArg.personne.getProperty("id"),
50                  numeroRue: contextArg.adresse.getProperty("numeroRue"),
51                  rue: contextArg.adresse.getProperty("rue"),
52                  codePostal: contextArg.adresse.getProperty("codePostal"),
53                  ville: contextArg.adresse.getProperty("ville"),
54                  pays: contextArg.adresse.getProperty("pays")
55              }
56          },
57          // Méthode callback qui reconstruit le modèle en cas de succès
58          success: ajaxCallbackSuccess,
59          // Méthode callback qui gère une éventuelle erreur dans la requête
60          error: ajaxCallbackError
61      });
62
63  };
64  // Enregistrement du callback de l'événement de mise à jour de la personne
65  myApp.gui.mediator.subscribe("adresse/changed", updateAdresse);
66
67  }());

```

exemples/clientServer/ex16_persistenceDeleteAdresse.js

```

1  /**
2   * Définition et enregistrement des callbacks de création d'une personne
3   * sur le serveur par requête AJAX.
4   */
5  myApp.addModule.apply(myApp.gui, ["callbacksDeleteAdresseQueryServer", function
6      () {
7          /**

```

```

8      * Méthode callback qui est appelée en cas de succès de la requête AJAX.
9      * Cette méthode informe simplement l'utilisateur des éventuelles erreurs.
10     * En effet, la requête n'est pas supposée retourner des données.
11     */
12     var ajaxCallbackSuccess = function(retrievedData){
13         var concatErrorMsg="";
14         if (retrievedData["error"] !== null){
15             for (var key in retrievedData['error']){
16                 if (retrievedData['error'].hasOwnProperty(key)){
17                     concatErrorMsg += key + ": " + retrievedData['error'][key] + "\n";
18                 }
19             }
20             alert(concatErrorMsg);
21         }
22     };
23
24     /**
25      * Méthode appelée en cas d'erreur de la requête AJAX elle même.
26      */
27     var ajaxCallbackError = function(retrievedData){
28         alert("Erreur : échec de la requête ajax");
29     };
30
31     /**
32      * Callback appelé lors de l'événement "personne/read" du médiateur.
33      * Effectue une requête AJAX pour récupérer toutes les personnes
34      * pour reconstruire le modèle de données.
35      */
36     var deleteAdresse = function(contextArg){
37
38         // requête AJAX get codé en JSON
39         var jqxhr = $.ajax({
40             dataType: "json", // On envoie les données la personne codée en JSON
41             url: "http://progjs/exemples/clientServer/api/", // URL du serveur
42             method: 'post', // Envoyer les données dans le tableau $_POST
43             contentType: 'application/x-www-form-urlencoded',
44             // données à transmettre au serveur
45             data : {
46                 action: "adresse-delete",
47                 id: contextArg.adresse.getProperty("id"),
48             },
49             // Méthode callback qui reconstruit le modèle en cas de succès
50             success: ajaxCallbackSuccess,
51             // Méthode callback qui gère une éventuelle erreur dans la requête
52             error: ajaxCallbackError
53         });
54
55     };
56     // Enregistrement du callback de l'événement de mise à jour de la personne
57     myApp.gui.mediator.subscribe("adresse/delete", deleteAdresse);
58
59 }());

```

Annexe A

Graphisme avec les Canvas *HTML5*

A.1 Notion de *canvas*

Les *canvas HTML5* fournissent une petite *API* graphique 2D en *javascript* qui permet de réaliser des dessins, des graphiques, etc. sans plugin. Les canvas 2D sont dorénavant et déjà disponibles sur tous les grands navigateurs. L'extension *webGL* (qui dépasse le cadre de ce cours) permet de faire des affichages de scènes 3D en accédant aux fonctionnalités d'*OpenGL* via les shaders en *GLSL*. L'extension *webGL* est implémentée dans tous les Grands Navigateurs mais n'est pas implémentée à ce jour dans *internet explorer* car l'éditeur de ce navigateur préfère privilégier une solution propriétaire.

Voici un exemple avec un canvas qui dessine un triangle.

exemples/canvas/ex01_triangle.html

```
1  <!doctype HTML>
2  <html lang="fr">
3    <head>
4      <meta charset="UTF-8" />
5      <title>Mon premier canvas HTML5</title>
6    </head>
7    <body>
8      <!-- Déclaration d'un canvas vide avec son id -->
9      <canvas id="monCanvas" width="2000" height="1000" style="position : absolute;"
10        ></canvas>
11      <script>
12        // On récupère le canvas pour dessiner
13        var myCanvas = document.getElementById( "monCanvas" );
14        // On récupère un contexte du canvas pour utiliser les méthodes de
15        // dessin
16        var context = myCanvas.getContext( "2d" );
17        // couleur de remplissage rouge
18        context.fillStyle = "#FF0000";
19
20        context.beginPath();
21        context.moveTo(10, 10);
22        context.lineTo(100, 100);
23        context.lineTo(190, 10);
24        context.lineTo(10, 10);
25
26        context.fill();
27        context.closePath();
```

```

26     </script>
27     <h1>Page HTML avec un canvas</h1>
28     <p>
29     </p>
30 </body>
31 </html>

```

A.2 Exemple d'animation dans un *canvas*

Voici un exemple qui réalise une animation à l'aide d'un timer qui exécute la fonction `animate` toutes les *20ms*, soit 50 fois par seconde.

exemples/canvas/ex02_animation.html

```

1  <!doctype HTML>
2  <html lang="fr">
3    <head>
4      <meta charset="UTF-8" />
5      <title>Mon premier canvas HTML5</title>
6    </head>
7    <body>
8      <!-- Déclaration d'un canvas vide avec son id -->
9      <canvas id="monCanvas" width="2000" height="1000" style="position :absolute;"
10        ></canvas>
11      <script>
12        var timer = setInterval(animate, 20);
13
14        function animate() {
15
16          // On récupère le canvas pour dessiner
17          var canvas = document.getElementById("monCanvas");
18          // On récupère un contexte du canvas pour utiliser les méthodes de
19          // dessin
20          var context = canvas.getContext("2d");
21          // couleur de remplissage rouge
22          context.fillStyle = "#FF0000";
23          context.beginPath();
24          var d = new Date();
25          var n = d.getTime();
26          // nombre de millisecondes depuis le 01/01/1970
27
28          var sec = n / 1000.0;
29          context.clearRect(0, 0, canvas.width, canvas.height);
30
31          context.save();
32          context.translate(200+500 * (1+Math.cos(0.5 * sec)), 200+200 * (1.0+
33            Math.sin(sec)));
34          // l'angle de rotation doit être entre 0 et 2*Math.PI
35          context.rotate(sec - 2*Math.PI*Math.round(sec/(2*Math.PI)));
36          context.moveTo(0, 0);
37          context.lineTo(100, 100);
38          context.lineTo(200, 0);
39          context.lineTo(0, 0);
40
41          context.fill();

```

```
39         context.closePath();
40         context.restore();
41     }
42     </script>
43     <h1>Page HIML avec un canvas</h1>
44     <p>
45
46     </p>
47 </body>
48 </html>
```


Annexe B

Programmation Événementielle en *JavaScript*

B.1 Rappel sur la Gestion d'Événements en *CSS*

Dans un style *CSS*, on peut mettre des styles différents sur une balise *HTML* donnée, suivant le contexte utilisateur, via la notion d'événement. Dans l'exemple suivant, le style d'un lien est modifié suivant que le lien a déjà été cliqué, ou si la souris survolle le lien (événement *hover*).

```
1  /* style par défaut des liens */
2  a:link {
3      text-decoration : none;
4      color : #00e; /* bleu clair */
5  }
6  /* style des liens visités */
7  a:visited {
8      text-decoration : none;
9      color : #c0c; /* mauve */
10 }
11 /* style des liens visités */
12 a:hover {
13     text-decoration : underline; /* souligné */
14     color : #e40; /* rouge vif */
15 }
```

Voici un autre exemple, dans lequel un élément *HTML* (ici une balise `` et son contenu) apparaît en *popup* pour afficher les détails d'une personne lors du survol de nom de la personne.

La balise *span* (au sein d'un paragraphe d'une classe *CSS* spécifique appelé `popupDetails`) est par défaut invisible (propriété `display` à `none`). Cette même balise *span* devient visible lorsque le paragraphe est survolé.

workspace_progWeb_2a/events/ex01_popup_html_css.html

```
1 <!doctype html>
2 <html lang="fr">
3   <head>
4     <meta charset="utf-8"/>
5     <link rel="stylesheet" href="./myStyle.css"/>
6     <style>
```

```

7      body{
8          font-family : "Comic Sans MS";
9          font-size : 120%;
10     }
11     h1{
12         margin : 0 auto;
13         text-align : center;
14     }
15     p.popupDetails{
16         background-color : yellow;
17         position : relative; /* pour positioner le span en absolu */
18         max-width : 200px;
19     }
20     p.popupDetails span {
21         display : none;
22     }
23     p.popupDetails:hover span {
24         position : absolute;
25         left : 200px;
26         top : -30;
27         min-width : 500px;
28         background-color : black;
29         color : white;
30         border-radius : 20px;
31         padding : 10px;
32         display : block;
33     }
34     </style>
35     <title>Popups en HTML et CSS</title>
36 </head>
37 <body>
38     <!-- début du corps HTML -->
39     <h1><i>Popup</i> en <i>HTML</i> et <i>CSS</i></h1>
40     <p class="popupDetails">
41         Scarlett Johansson
42         <span>née le 22 novembre 1984 à New York,
43             est une actrice et chanteuse américaine.<br/>
44             (source&nbsp;: wikipédia)
45         </span>
46     </p>
47 </body>
48     <!-- fin du corps HTML -->
49 </html>
50 <!-- fin du code HTML -->

```

B.2 Événements en *Javascript*

B.2.1 Le principe des événements en *Javascript*

Les événements en *Javascript* permettent, en réponse à un événement sur un élément *HTML* du document, d'appeler une fonction *callback* en *Javascript*. Ceci suffit à créer une interface homme machine (*IHM*) côté client, basée sur de la programmation événementielle en *Javascript*.

Une liste (non exhaustive ; Voir sur le *web* pour la liste complète)

1. Événements souris

- (a) `onclick` : sur un simple clic
- (b) `ondblclick` : sur un double clic
- (c) `onmousedown` : lorsque le bouton de la souris est enfoncé, sans forcément le relâcher
- (d) `onmousemove` : lorsque la souris est déplacée
- (e) `onmouseout` : lorsque la souris sort de l'élément
- (f) `onmouseover` : lorsque la souris est sur l'élément
- (g) `onmouseup` : lorsque le bouton de la souris est relâché

2. Événements clavier

- (a) `onkeydown` : lorsqu'une touche est enfoncée
- (b) `onkeypress` : lorsqu'une touche est pressée et relâchée
- (c) `onkeyup` : lorsqu'une touche est relâchée

3. Événements formulaire

- (a)
- (b) `onblur` : à la perte du focus
- (c) `onchange` : à la perte du focus si la valeur a changé
- (d) `onfocus` : lorsque l'élément prend le focus (ou devient actif)
- (e) `onreset` : lors de la remise à zéro du formulaire (via un bouton "reset" ou une fonction `reset()`)
- (f) `onselect` : quand du texte est sélectionné
- (g) `onsubmit` : quand le formulaire est validé (via un bouton de type "submit" ou une fonction `submit()`)

B.2.2 Exemple de mise à jour d'un élément

workspace_progWeb_2a/events/ex02_updateElementOnChange.html

```
1 <!doctype html>
2 <html lang="fr">
3   <head>
4     <meta charset="utf-8"/>
5     <link rel="stylesheet" href="./myStyle.css"/>
6     <style>
7       body{
8         font-family : "Comic Sans MS";
9         font-size : 120%;
10      }
11      h1{
12        margin : 0 auto;
13        text-align : center;
14      }
15    </style>
```

```

16 <title>Mise à Jour Par Événement</title>
17 </head>
18 <body>
19 <!-- début du corps HTML -->
20 <h1>Mise à Jour Par Événement <code>onchange</code></h1>
21 <p class="popupDetails">
22 <input id="myInputId" type="text" size="15"
23 <code>onchange="fonctionMiseAJour( 'myInputContent', 'myInputId' )"/>
24 <br/>
25 <span id="myInputContent"></span>
26 </p>
27 <script>
28 <code>function fonctionMiseAJour(elementId, inputId){
29 <code>document.getElementById(elementId).innerHTML
30 <code>= document.getElementById(inputId).value;
31 <code>}
32 </script>
33 </body>
34 <!-- fin du corps HTML -->
35 </html>
36 <!-- fin du code HTML -->

```

B.2.3 Formulaires Dynamiques an *Javascript*

Nous voyons ici un exemple d'utilisation du *javascript* pour créer un formulaire dont les attributs dépendent de la valeur d'un premier champ. Lorsqu'on sélectionne "deuxième année", un nouveau champ apparaît. Pour cela, on utilise l'événement `onchange` sur l'`input` de l'année, qui est géré par la fonction `anneeChange`. On teste alors la valeur de l'attribut, puis le cas échéant on génère un nouveau champ dans un `div` d'id `attributSupplementaire`.

Nom

Première année

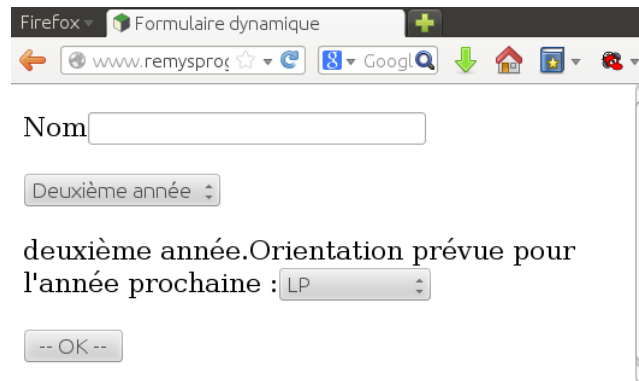
premiere année.

workspace_progWeb_2a/events/formulaire_dynamique.html

```

1 <!doctype html>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8"/>
5 <title>Formulaire dynamique</title>
6 </head>
7 <body>
8 <form method="post" action="reception.php">
9 <p>
10 <label for="nom">Nom</label><input name="nom" id="nom"/>

```



```

11     </p>
12     <p>
13         <select name="annee" id="annee" pattern="(premiere) | (deuxieme)"
14             onchange='anneeChange(); '>
15         <option value="choisissez" selected disabled>— choisissez —</option>
16         <option value="premiere">Première année</option>
17         <option value="deuxieme">Deuxième année</option>
18     </select>
19     </p>
20     <div id="attributSupplementaire">
21
22     </div>
23     <p>
24         <input type="submit" value="— OK —"/>
25     </p>
26 </form>
27 <script>
28     function anneeChange() {
29         var paragraphe = document.getElementById("attributSupplementaire");
30         paragraphe.innerHTML=document.getElementById("annee").value+" année.<br />";
31         if (document.getElementById("annee").value == "deuxieme"){
32             paragraphe.innerHTML+="

```

workspace_progWeb_2a/events/reception.php

```

1 <!doctype html>
2 <html lang="fr">
3 <head>

```

```
4      <meta charset="UTF-8"/>
5      <title>Formulaire dynamique</title>
6  </head>
7  <body>
8  <?php
9      $nom= ( isset($_POST["nom"]) ) ? $_POST["nom"] : "nom indéterminé";
10     $annee = ( isset($_POST["annee"]) ) ? $_POST["annee"] : "année indéterminée";
11         echo "Nom : ".$nom."<br />";
12         echo "Année : ".$annee."<br />";
13     if ($annee=="deuxième")
14         echo " Orientation : ".$_POST["orientation"];
15
16
17 ?>
18 </body>
19 </html>
```

Annexe C

Gestion des fenêtres

C.1 Charger un nouveau document

workspace_progWeb_2a/window/ex01_classes_telephone.js

```
1 // constructeur
2 function Telephone(tell){
3   // test de téléphone fran_ais à 10 chiffres
4   // 1) supprimer les espaces, 2) tester les chiffres
5   if (tell.replace(/\s/g, '').match(/^(|\+33|0)[0-9]{9}$/g))
6     this.tell=tell;
7   else
8     throw new Error("Numéro de téléphone invalide");
9 }
10
11 Telephone.prototype.affiche = function(){
12   document.write("Téléphone 1 : "+this.tell+"<br/>");
13 }
```

workspace_progWeb_2a/window/ex01_loadNewDoc.html

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Charger un document</title>
6 <script src="./ex01_classes_telephone.js"></script>
7 </head>
8 <body>
9 <p>
10 <script>
11 try{
12   var numero = prompt("Merci d'entrer un numéro de téléphone en France mé
    tropolitaine");
13   var tel = new Telephone(numero);
14   tel.affiche();
15 }catch (err){
16   location = "ex01_error.html";
17 }
18 </script>
19 <p>
```

```
20 </body>
21 </html>
```

workspace_progWeb_2a/window/ex01_error.html

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Charger un document</title>
6 <script src="./ex10_classes_telephone.js"></script>
7 </head>
8 <body>
9 <p>
10 Bonjour, Il s'est produit une erreur. Merci d'entrer un numéro valide.
11 Si le problème persiste, merci de contacter le stagiaire qui a fait le site...
12 <button onclick="location = 'ex01_loadNewDoc.html';">Retour à la saisie</button>
13 <p>
14 </body>
15 </html>
```

C.2 Naviguer dans l'historique

la propriété `history` a deux méthodes `back()` et `forward()` qui permettent respectivement de reculer ou d'avancer dans l'historique.

workspace_progWeb_2a/window/ex02_history.html

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Charger un document</title>
6 <script src="./ex10_classes_telephone.js"></script>
7 </head>
8 <body>
9 <p>
10 Bonjour, bla, bla...<br/>
11 <a href = "ex02_historyBack.html">Cliquez ici</a> pour aller à la page suivante.
12 <p>
13 </body>
14 </html>
```

workspace_progWeb_2a/window/ex02_historyBack.html

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Charger un document</title>
6 <script src="./ex10_classes_telephone.js"></script>
7 </head>
8 <body>
9 <p>
```



```
10   Bla, bla...<br/>
11   Vous avez raté quelque chose ?
12   <button onclick="history.back();">Retour à la page précédente</button>
13 <p>
14 </body>
15 </html>
```

C.3 Ouvrir une nouvelle fenêtre (popup)

workspace_progWeb_2a/window/ex03_windowOpen.html

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Ouvrir un fenêtre</title>
6 <script src="./ex10_classes_telephone.js"></script>
7 </head>
8 <body>
9 <p>
10 Bonjour, bla, bla...
11 <button onclick="window.open('ex03_windowPopup.html', 'ma popup', 'width=400,
    height=400,resizeable=yes');">
12   Plus d'infos
13 </button>
14 <p>
15 </body>
16 </html>
```

workspace_progWeb_2a/window/ex03_windowPopup.html

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Charger un document</title>
6 <script src="./ex10_classes_telephone.js"></script>
7 </head>
8 <body>
9   <p style="font-size : 100; text-align : center;">
10     Coucou !
11   <p>
12   <p>
13     <a href="javascript :window.close();">Fermer la fenêtre</a>
14   </p>
15 </body>
16 </html>
```

Annexe D

Document Object Model (DOM)

La programmation côté client permet de modifier certaines parties d'un document *HTML* dans recharger toute la page. Il y a plusieurs avantages : on évite de surcharger le serveur et le trafic réseau et on améliore la réactivité de l'application *web* pour le plus grand bonheur de l'utilisateur.

Pour faire cela, le langage *Javascript* côté client fournit une structure de données permettant d'accéder aux éléments du document *HTML* et de modifier les éléments du document *HTML*. Cette structure de données s'appelle le *Document Object Model*, en abrégé *DOM*. Il existe un *DOM* legacy qui s'est sédimenté informellement au travers des versions successives du *javascript* en tenant compte des implémentations des différents navigateurs, qui collaboraient plus ou moins bien pour être mutuellement compatibles. Il existe aussi le *DOM* tel qu'il a été finalement spécifié par le *W3C*.

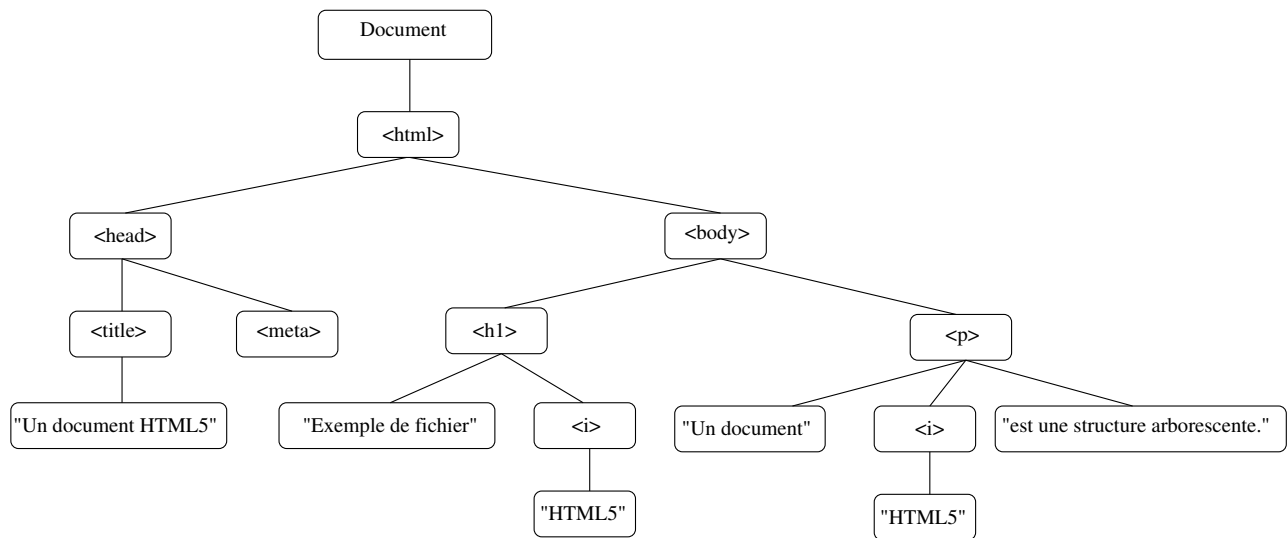
Les éléments du document *HTML* ayant, de par leur imbrication, une structure arborescente, le *DOM W3C* a une structure d'arbre. On peut accéder et manipuler via un ensemble de propriétés et de méthodes *javascript*, notamment de l'interface **Document** et de l'interface **Element** et ses classes filles, qui permettent de manipuler les éléments (*HTML* entre autres) du document.

D.1 Qu'est-ce que le *DOM* ?

Le *Document Object Model* (en abrégé *DOM*) correspond à l'arborescence des imbrications des balises *HTML* d'un document. Voici un fichier *HTML* simple et une représentation schématique du *DOM* correspondant.

workspace_progWeb_2a/dom/ex01_documentHTML5.html

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Un document HTML5</title>
6 </head>
7 <body>
8   <h1>Exemple de fichier <i>HTML5</i></h1>
9   <p>
10     Un document <i>HTML5</i> est une structure arborescente.
11   </p>
12 </body>
```

13 `</html>`

Le *DOM* dont nous parlons ici est le *DOM* du *W3C*, qui est aujourd'hui supporté par tous les grands navigateurs.

Le langage *Javascript* côté client propose une hiérarchie de classes pour parcourir et manipuler le *DOM* d'un document. Il s'agit essentiellement d'une structure de donnée d'arbre, où chaque noeud (correspondant à une balise ou commentaire ou texte, etc. du document) possède une collection de noeuds fils, qui sont les éléments ou structures imbriquées.

La bibliothèque *jQuery* permet un accès plus haut niveau au *DOM* pour sonder et manipuler le code du document.

D.2 Sélection et Manipulation de Base sur le *DOM*

D.2.1 Sélection de tout ou partie des éléments

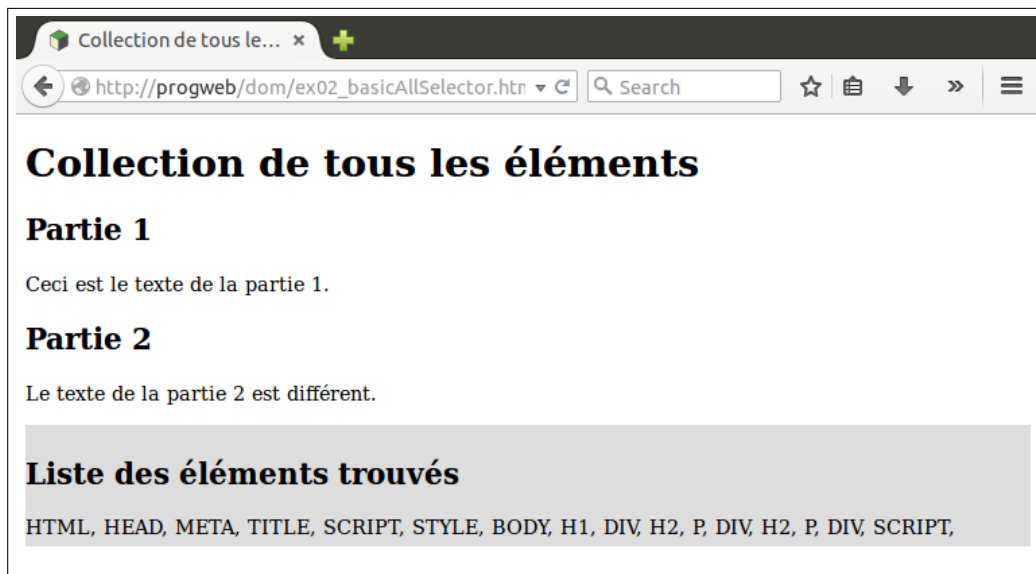
L'exemple suivant cherche tous les éléments du document et affiche leur nom de balise (`tagName` ou `nodeName`). On apprend aussi à ajouter du code *HTML* à l'intérieur d'un élément (au début ou à la fin).

exemples/dom/ex02_basicAllSelector.html

```

1 <!doctype html>
2 <html lang="fr">
3 <head>
4   <meta charset="utf-8">
5   <title>Collection de tous les éléments</title>
6   <script src="./jquery-1.10.2.js"></script>
7   <style>
8     div#javascriptOutputDiv{
9       background-color : #ddd;
10      padding : 5px 0;
11    }
12  </style>
13 </head>
14

```



```

15 <body>
16 <h1>Collection de tous les éléments</h1>
17 <div>
18   <h2>Partie 1</h2>
19   <p>Ceci est le texte de la partie 1.</p>
20 </div>
21 <div>
22   <h2>Partie 2</h2>
23   <p>Le texte de la partie 2 est différent.</p>
24 </div>
25 <div id="javascriptOutputDiv"></div>
26 <script>
27   // Récupération d'un ensemble d'éléments jQuery
28   var elements = $( "*" );
29   // Obtention d'un Array d'Elements (Interface du DOM W3C classique)
30   var arrayElements = elements.toArray();
31   // Parcours du tableau
32   for (var i=0 ; i<arrayElements.length ; i++){
33     // Ajout du nom du tag HTML dans le div d'ID javascriptOutputDiv
34     $( "#javascriptOutputDiv" ).append( arrayElements[ i ].nodeName+", " );
35   }
36   // Ajout d'un titre AU DÉBUT du div d'ID javascriptOutputDiv
37   $( "#javascriptOutputDiv" ).prepend( "<h2>Liste des éléments trouvés</h2>" );
38 </script>
39 </body>

```

L'exemple suivant montre comment sélectionner certains éléments du document, par nom de balise, classe *CSS*, etc. On apprend aussi à modifier des propriétés *CSS* des éléments.

exemples/dom/ex03_basicMultiSelector.html

```

1 <!doctype html>
2 <html lang="fr">
3 <head>
4   <meta charset="utf-8">
5   <title>Modifier le style de certains éléments</title>
6   <script src="/jquery-1.10.2.js"></script>
7   <style>

```



```

8   p.myClass {
9       background-color : #ddd;
10      padding : 10px;
11  }
12  </style>
13  </head>
14
15  <body>
16  <h1>Modifier le style de certains éléments</h1>
17  <div>
18      <h2>Partie 1</h2>
19      <p>Ceci est le texte de la partie 1.</p>
20  </div>
21  <div>
22      <h2>Partie 2</h2>
23      <p class="myClass">Le texte de la partie 2 est différent.</p>
24  </div>
25  <script>
26      // Récupération d'éléments jQuery pour les balises <p> et <h2>
27      var elements = $( "p, h2" );
28      elements.css( "border", "2px solid" );
29      // Modification du style du titre <h1>
30      $( "h1" ).css( "text-align", "center" );
31      // Modification du style du (ou des) paragraphe(s) de la classe CSS myClass
32      $( "p.myClass" ).css( "border-radius", "20px" );
33  </script>
34  </body>

```

D.2.2 Filtrage par le texte

L'exemple suivant montre comment sélectionner des éléments par mots du texte (sensible à la casse).

exemples/dom/ex04_contains.html

```

1  <!doctype html>
2  <html lang="fr">

```



```

3 <head>
4   <meta charset="utf-8">
5   <title>Filtrage du texte</title>
6   <script src="/jquery-1.10.2.js"></script>
7   <style>
8     p {
9       padding: 10px 0;
10    }
11  </style>
12 </head>
13 <body>
14 <h1>Filtrage du texte</h1>
15 <div>
16   <h2>Partie 1</h2>
17   <p>Ceci est le texte de la partie 1.</p>
18 </div>
19 <div>
20   <h2>Partie 2</h2>
21   <p>Le texte de la <em>partie 2</em> est différent.</p>
22 </div>
23 <script>
24   $( "p:contains('différent')" ).prepend( '<strong>Ce paragraphe contient le mot
25     "différent"</strong>.<br/>' ).css( "background-color", "#ddd" );
26 </script>
27 </body>

```

D.2.3 Application de Méthode aux éléments

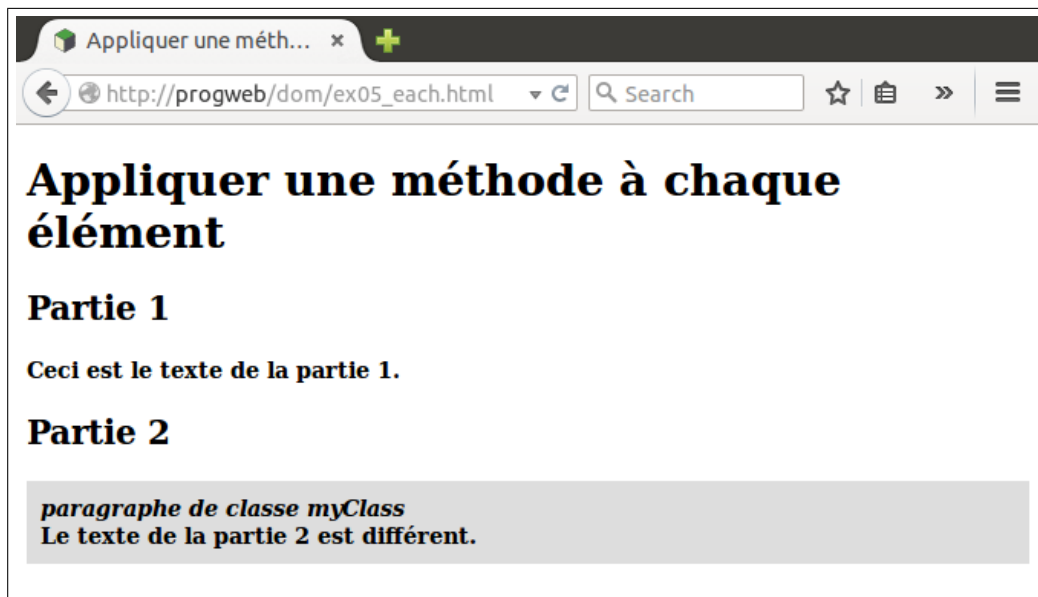
L'exemple suivant montre comment appliquer une fonction à chacun des éléments sélectionnés. Ici, on met le contenu des paragraphes en gras. On ajoute une information au début de chaque paragraphe de la classe `myClass`.

exemples/dom/ex05_each.html

```

1 <!doctype html>
2 <html lang="fr">

```



```
3 <head>
4   <meta charset="utf-8">
5   <title>Appliquer une méthode à chaque élément</title>
6   <script src="../jquery-1.10.2.js"></script>
7   <style>
8     p.myClass {
9       background-color : #ddd;
10      padding : 10px;
11    }
12  </style>
13 </head>
14
15 <body>
16 <h1>Appliquer une méthode à chaque élément</h1>
17 <div>
18   <h2>Partie 1</h2>
19   <p>Ceci est le texte de la partie 1.</p>
20 </div>
21 <div>
22   <h2>Partie 2</h2>
23   <p class="myClass">Le texte de la partie 2 est différent.</p>
24 </div>
25 <script>
26   $( "p" ).each(function() {
27     $( this ).css( "font-weight", "bolder" );
28     if ( $( this ).hasClass( "myClass" )) {
29       $( this ).prepend( "<em>paragraphe de classe myClass</em><br />" );
30     }
31   });
32 </script>
33 </body>
```

D.2.4 Événements et *Callbacks*

EL'exemple suivant montre comment, en réaction au click sur un bouton, transformer es paragraphes en div.



exemples/dom/ex06_clickEvent.html

```

1 <!doctype html>
2 <html lang="fr">
3 <head>
4   <meta charset="utf-8">
5   <title>Événement de click</title>
6   <script src="./jquery-1.10.2.js"></script>
7   <style>
8     p {
9       background-color : #ddd;
10      padding : 10px;
11    }
12    div.myClass {
13      font-weight : bolder;
14      padding : 10px;
15      border-style : dashed;
16    }
17    em {
18      font-variant : small-caps;
19      font-size : 120%;
20    }
21    button {
22      margin : 10px 0;
23    }
24  </style>
25 </head>
26
27 <body>
28 <h1>Événement de click</h1>
29 <div>

```



```

30 <h2>Partie 1</h2>
31 <p>Ceci est le texte de la partie 1.</p>
32 </div>
33 <div>
34 <h2>Partie 2</h2>
35 <p>Le texte de la <em>partie 2</em> est différent.</p>
36 </div>
37 <button>Modifier les paragraphes</button>
38 <script>
39 // Événement de click
40 $( "button" ).click(function(){
41 // Application d'une méthode à chaque paragraphe
42 $( "p" ).each(function(){
43 // Remplacer le <p> par un <div> en laissant le HTML inchangé
44 $( this ).replaceWith( '<div class="myClass">' + $( this ).html()
45 + "</div>" );
46 });
47 </script>
48 </body>

```

D.2.5 Filtrage d'un Tableau

L'exemple suivant montre comment, en utilisant les utilitaires de *jQuery* permettant de traiter des *Array Javascript* génériques :

1. Filtrer le contenu d'un tableau avec une méthode de choix booléenne pour les éléments (ici, valeur multiple de 3);
2. Générer le *HTML* en appliquant une méthode à chaque élément du tableau.



exemples/dom/ex07_filterGrep.html

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Filtrage Grep sur Tableau</title>
6 </head>
7 <body>
8   <h1>Filtrage Grep sur Tableau</h1>
9   <script src="./jquery-1.10.2.js"></script>
10
11 <p id="output"></p>
12
13 <script>
14   // Création d'un tableau avec les entiers de 0 à 19
15   var tab = new Array();
16   for (var i=0 ; i<20 ; i++){
17     tab.push(i);
18   }
19
20   // Sélection des éléments du tableau par la fonction "multiple de 3"
21   var tabMultipleDe3 = $.grep(tab, function(key, value){
22     if (key%3 == 0)
23       return true;
24     else
25       return false;
26   });
27
28   // Affichage du tableau des multiples de 3
29   var outHTML = "";
30   // Application d'une fonction (génération d'HTML)
31   // à chaque élément du tableau
32   $.each(tabMultipleDe3, function(key, value){
33     outHTML += "tab["+key+"] = "+value+"<br/>";
34   });
35   $( "#output" ).append( outHTML );
36 </script>
37 </body>
38 </html>
```