Chapter 10 : Script et DOM

Ce que nous allons examiner dans ce chapitre

- Un survol du langage ECMAScript et des composants dont nous aurons besoin.
- Le DOM (Document Object Model)
- Ajouter et enlever des éléments à notre document SVG
- Ajouter de l'interactivité
- Gérer les événements souris
- Quelques exemples

Le langage ECMAScript

> *ECMAScript est un langage de programmation orienté objet permettant de manipuler les objets dans un environnement Web. Ce langage est aussi nommé JavaScript.*

Pour transformer un document statique en un document interactif, le W3C nous fournit une structure de document arborescente le '**Document Object Model'** ou **DOM**. Pour communiquer avec ces objets, nous avons besoin d'un langage de programmation. Selon les spécifications SVG, le langage '**ECMA script'** doit être supporté par les visualiseurs pour nous permettre d'accéder aux éléments de notre document.

Le mot est lâché, programmation. “Je suis un concepteur de pages Web, je ne suis pas programmeur ” est une réaction fréquente. Toutefois comme graphiste, vous ne pouvez échapper aux possibilités qu'offre les scripts. Après avoir lu ce chapitre vous serez capable d'utiliser efficacement des librairies existantes, d'adapter des exemples et si le virus vous gagne, de créer vos propres scripts pour vos projets.

Commençons par voir ce qu'est ECMAScript?

Les fondamentaux d'ECMAScript

Il y a déjà très longtemps, Netscape décida d'ajouter un langage de script à son navigateur. Avec ce langage, les concepteurs de pages Web pourraient ajouter de l'interactivité et du dynamisme à leurs pages sans passer par des scripts serveur comme CGI ou des applets Java. Ce langage devait ressembler à C++ et Java. D'un autre côté, il devait être simple à apprendre et utilisable par des non-programmeurs. Comme tout le monde parlait de Java à l'époque, ils lui donnèrent le nom de '**Javascript'**, quoique qu'il présente peu de points communs avec Java en dehors de la syntaxe.

Javascript fut un succès et les scripts proliférèrent sur le Web. Avec le succès, apparut la nécessité de coordonner et de standardiser les efforts des uns et des autres. Aussi Netscape, Microsoft, et d'autres acteurs majeurs formèrent un comité pour établir un standard. Les spécifications du langage devinrent un standard ouvert sous les auspices de l'association '**European Computer Manufacturers’Association**’ (**ECMA**).

Aujourd'hui, la troisième version de ce standard '**ECMAScript'** est considérée stable et efficace. Tous les navigateurs respectent ces standards, ou du moins le prétendent. Le langage Javascript de Netscape et le Jscript de Microsoft sont des exemples d'intégration de ECMAScript dans leurs navigateurs respectifs.

A propos de script

Un langage de script est différent d'un langage traditionnel comme FORTRAN, PASCAL, ou C++.
Un script n'est pas compilé comme un programme exécutable. Il est interprété par étapes par un interpréteur intégré au navigateur ou au serveur. ECMAScript est un langage de script interprété et orienté objet. Bien qu'il ait moins de possibilités qu'un langage comme C++, ECMAScript est tout à fait suffisant pour un langage de script utilisé dans des pages Web.

Les bases de ECMAScript

Ce chapitre ne prétend pas tout traiter à propos de ECMAScript. Nous allons nous concentrer sur ce qui nous est nécessaire pour nos documents SVG. Un programme ECMAScript est au format texte. Ce texte est organisé en déclarations, commentaires et blocs de programme. Chaque bloc peut lui même contenir déclarations, commentaires et blocs. Pour chaque déclaration nous trouvons des types de données, des variables et des expressions. Voyons un exemple, le rythme de la semaine vu par ECMAScript pourrait être:

```
var action = "", today = DayOfWeek();
if (today == "Sunday")
   action = "sleeping";
else if (today == "Saturday")
   action = "shopping";
else                          // waiting for weekend ..
   action = "working";
```

Dans cet exemple, nous avons des déclarations (les blocs if (…) else (…)), un commentaire (waiting for weekend), des expressions (today==”Sunday”) et des variables (action). Un type de données est utilisé également (today = DayOfWeek()).

Types de données

ECMAScript est un langage faiblement typé contrairement à d'autres langages. Ceci signifie que le programmeur ne se préoccupe pas trop des types de données qu'il manipule, le langage se chargeant lui-même de déterminer le type de données implicite d'une variable ou du résultat d'une fonction. Mais nous pouvons avoir quelques surprises désagréables, si nous écrivons 1+1, le résultat est-il 2 ou 11? Tout dépend du type de données que le langage va appliquer, nombres (le résultat sera évidemment 2) ou caractères (le résultat est logiquement 11). Ceci dit nous déclarons nos variables avec la déclaration **var**. Voyons ces différents types de données.

Nombre

Pour des valeurs numériques 'number' est le seul type de donnée. Il n'y a pas de distinction entre entiers, rationnels et réels.

```
-273, 1024, 0              // entiers ..
-6.28, 0.25, 66.666667     // réels ..
0.001, .001, 1.0e-3, 1E-3  // écritures avec puissances de 10
```

Chaîne

Pour du texte 'string' est le type de données. ECMAScript ne fait pas de différence entre une chaîne de caractères et un caractère isolé (au contraire de C++ et Java). Les chaînes sont entourées de guillemets simples (') ou doubles (").

Quoiqu'il soit possible de choisir ' ou " pour délimiter votre chaîne, les symboles d'ouverture et de fermeture de la chaîne doivent être les mêmes. Choisissez l'un des deux et gardez l'autre quand il fait lui même partie de la chaîne.Si vous ouvrez votre chaîne avec ' et la fermez avec ", vous aurez un message d 'erreur.

```
"a short text", 'another text'     // délimiteurs de texte ..
"mike's bike", 'he said: "hello"'  // ' ou " dans la chaîne ..
'c', 'C', "4", '"', " "            // caractères ..
"", ''                             // la chaîne vide ..
"\"", "\n", '\'', '\\'             // caractères spéciaux
```

Booléen

Ce type de donnée n'a que deux valeurs possibles.

```
true, false     // les deux valeurs ..
```

Avec ECMAScript les valeurs '`true`' et '`false`' correspondent à '`1`' et '`0`'.

Objet

Le type 'object' est le plus important — celui qui permet de dire que ECMAScript est un langage orienté objet. Il n'y a pas de valeurs assignées directement, un objet peut être considéré comme un container d'autres types de données ou d'autres objets. Nous verrons cette notion en détail.

Commentaires

Les commentaires permettent d'expliquer le code. Un commentaire ne fait rien, il n'est pas exécuté.
ECMAScript permet les commentaires sur une seule ligne qui commencent par "//" comme sur cet exemple

```
else     // waiting for weekend ..
```

le commentaire se termine avec la fin de la ligne.

Nous pouvons également mettre des commentaires sur plusieurs lignes. Ils commencent par "/*" et se terminent avec "*/", en voici un exemple.

```
/* ce commentaire
   se poursuit
 sur plusieurs lignes */
```

Tout le texte entre /* .. */ n'est pas considéré par ECMAScript comme faisant partie du code du programme. Je vous recommande de n'utiliser que des commentaires sur une seule ligne. Vous pourrez utiliser /* et */ pour mettre entre parenthèses une partie du code si vous voulez tester les autres parties de votre code.

Variables

Comme dans tous les langages de programmation, les variables jouent un rôle essentiel. Voyez une variable comme un container qui porte le nom de la variable et qui contient une valeur. Vous accédez à la valeur en utilisant le nom du container. Une variable peut contenir n'importe quel type de donnée, et comme ECMAScript est très peu typé, ce type de variable peut même changer au cours du programme.

```
var pi = 3.14, radius = 100;        // définit deux variables ..
var circumference = 0, area = 0;    // .. et deux autres ..
circumference = 2*pi*radius;        // assigne une valeur ..
area = pi*radius*radius;            // .. à chaque variable
pi = 'hello';                       // le type de donnée change ..
```

Une variable est définie par le mot réservé **var,** et, après cette définition, peut être utilisé dans une expression. Dans la plupart des situations, ECMAScript permet l'utilisation de variables même si elles n'ont pas été déclarées. Pour une pratique rigoureuse de la programmation il est recommandé de toujours les déclarer.

Vous pouvez utiliser presque n'importe quelle chaîne pour donner un nom à vos variables, la longueur de la chaîne n'est pas limitée.

> *Une pratique judicieuse est de donner aux variables des noms significatifs qui à la lecture du programme permettent de savoir ce qu'elles représentent.*

Pour nommer les variables, il y a cependant quelques règles:

- Le premier caractère doit être une lettre, minuscule ou majuscule, ou un souligné.
- Les caractères permis sont les lettres, les chiffres et les soulignés.
- Les mots réservés à ECMAScript's (var, for, if, else, ..) ne peuvent nommer une variable

Quand vous définissez une variable (avec var) il n'est pas indispensable de lui assigner une valeur, cependant je vous recommande de le faire. Cela vous évitera des erreurs d'exécution dûes à des valeurs qui n'ont pas été initialisées. Par défaut, une variable prend la valeur 'undefined' et bloque les calculs.

Expressions

Une expression est un morceau de code ECMAScript qui, quand il est évalué renvoie une valeur.

```
var pi = 3.14, radius = 100, str = "hello";     // définir quelques . .
var color = "green";                            // . . variables
2+3/2                                           // 3.5
(2+3)/2                                         // 2.5
radius*2/10                                     // 20
2*pi*radius;                                    // 628
radius == 100                                   // true
str == "blue"                                   // false
str + " Mary"                                   // "hello Mary"
str.length                                      // 5
Math.sin(Math.PI/2)                             // 1
color == "red" || color == "blue"               // false
```

Une expression est constituée de

- Nombres, Chaînes, Booléens
- Opérateurs `(+, -, *, /, …)`
- Variables
- Appels de fonctions
- Appels de propriétés ou de méthodes des objets

La valeur de l'expression peut être un nombre, une chaîne ou un booléen.

Les opérateurs

ECMAScript définit un grand nombre d'opérateurs, la plupart binaires (comme + -) avec deux opérandes. Nous avons aussi beaucoup d'opérateurs avec un seul opérande comme i++ avec un opérande i et l'opérateur d'incrémentation ++. Nous classons ces opérateurs par catégories.

Categorie	Opérateur	Signification	Exemple	Valeur renvoyée
Opérateurs arithmétiques	+	Addition	1 + 2	3
	-	Soustraction	8 - 6	2
	*	Multiplication	2*3	6

	/	Division	10/3	3.3333333333
	%	Modulo	10%3	1
	++	Incrément	x = 3; x++	x=4
	--	Décrément	x = 5; x--	x=4
	-	Négation	-2	negative 2
Opérateurs pour assigner une valeur	+=	Addition	x += y	x=x+y
	-=	Soustraction	x -= y	x=x-y
	*=	Multiplication	x *= y	x=x*y
	/=	Division	x /= y	x=x/y
	%	Modulo	x %= y	x=x%y
Opérateurs sur les chaînes	+	Concaténation	"hello "+"world"	"hello world"
	+=	Concaténation	a += b	a=a+b
Opérateurs de comparaison	==	est égal à	2 == 3	false
	!=	n'est pas égal à	2 != 3	true
	>	est plus grand que	5 > 3	true
	>=	est plus grand ou égal à	5 >= 3	true
	<	est plus petit que	5 < 3	false
	<=	est plus petit ou égal à	5 <= 3	false
Opérateurs logiques	&&	ET	(3 <= 5) && (5 < 8)	true
	||	OU	(5 > 3) || (3 > 5)	true
	!	NON	5 != 3	true

En ECMAScript l'opérateur '+' agit non seulement sur des nombres comme dans 3+7, mais aussi sur des chaînes de caractères, ainsi "SVG "+"est "+"cool" renverra "SVG est cool". Ceci pourra vous causer beaucoup d'ennuis quand une valeur récupérée sur l'attribut d'un élément SVG ne sera pas explicitement numérique.

Déclarations

Une déclaration peut être considéré comme un ordre pour le langage ECMAScript. Ces déclarations utilisent intensivement les expressions. Quand l'ordre est exécuté, une certaine action est déclenchée, aussi nous pouvons considérer un programme comme une suite d'ordres pour accomplir une tâche donnée.

```
var name = "Robin";
document.write("Hello " + name + " how are you?");
```

Le point virgule ";" est considéré comme la fin d'une déclaration, ces déclarations peuvent donc s'écrire sur plusieurs lignes ou au contraire nous pouvons mettre plusieurs déclarations sur la même ligne. Pour une bonne pratique, il est recommandé de ne pas mettre plusieurs ordres sur une même ligne.

Il est recommandé de mettre ; à la fin de chaque déclaration. Vous serez certain que vos ordres seront compris comme vous le désirez.

Un programme ECMAScript est exécuté du début à la fin, partant de la première déclaration jusqu'à la dernière. Avec cette exécution séquentielle, vous ne pouvez écrire que des programmes rudimentaires. ECMAScript — comme les autres langages — supporte les sauts et les répétitions dans l'ordre des déclarations.

Déclarations conditionnelles

La déclaration conditionnelle **if** permet l'exécution d'ordres suivant la réalisation d'une condition.

```
if (<condition>)
{
```

```
   <statements>;
}
```

Les déclarations suivant if(..)(le bloc **if**) ne seront exécutées que si la condition — une expression booléenne — mise entre parenthèses renvoie la valeur 'true'. Quand le bloc 'if' est composé de plusieurs ordres, ils sont placés entre { et }. Si nous n'avons qu'un ordre, nous pouvons ne pas écrire ces parenthèses cursives.

Exécuter différentes déclarations avec `if…else`

La déclaration if..else ajoute la possibilité de spécifier des ordres alternatifs, exécutés si la condition n'est pas vérifiée. Par exemple, dans ce code, nous cherchons si un point (x, y) est à l'intérieur d'un carré avec l'angle supérieur gauche en (100, 200) et un côté de 50.

```
if (x < 100 || y < 200 || x > 150 || y > 250)
   inside = false;
else
   inside = true;
```

Fréquemment, les déclarations if..else sont emboîtées pour tester plusieurs conditions:

```
if (background == ”black”)
   color = ”white”;
else if (background == ”white”)
   color = ”black”;
else
   color = ”blue”;
```

Répétition d'ordres

La déclaration while fonctionne comme if, mais les ordres ne sont pas exécutés une seule fois, mais tant que la condition est réalisée.

```
while (hour >= 8 && hour <= 17)
{
   action = ”working”;
}
```

Cet exemple provoquerait une boucle sans fin, car dans les ordres exécutés, la variable 'hour' n'est pas affectée.

Nous avons souvent besoin de traiter une série d'objets. Pour cela, nous utilisons un compteur incrémenté à chaque boucle.

```
var sum = 0;
for (var i=1; i<=100; i++)
{
   sum += i; // équivalent à .. sum = sum + i;
}
```

Cet exemple nous donnera la somme des 100 premiers entiers soit 5050.

Avec cet exemple, nous avons la déclaration **for**. Voyons sa syntaxe.

```
for (<initialisation>; <condition>; <mise à jour>)
{
   <ordres>;
}
```

Pour la <condition>, une expression booléenne, les valeurs de départ pour <initialisation> et souvent une incrémentation pour <mise à jour>.
Ceci fonctionne ainsi:

- Définir un ou plusieurs compteurs dans `<initialisation>`.
- Evaluer `<condition>`.
- Si `<condition>` renvoie `true`, exécuter les ordres, effectuer `<mise à jour>` et revenir à l'étape précédente, évaluer `<condition>` .
  Si `<condition>` renvoie `false`, sortir de la déclaration for.

Cette déclaration 'for' est utilisée très fréquemment en C++, Java ou ECMAScript, elle permet en une seule instruction de définir, initialiser et mettre à jour un compteur. Les sections `<initialisation> , <condition> ou <mise à jour>` peuvent être vierges mais vous devez mettre les ";".

L'instruction 'break' permet de sortir de 'for' sur une condition. Voyez cet exemple:

```
for (;;)  // 'forever' .. les trois sections sont vides ..
{
   if (hour >= 22)
      break;
   action = "working";
}
```

Avec ECMAScript, nous avons d'autres déclarations répétitives ou conditionnelles comme 'switch…case', 'do…while' et 'for…in' que nous laisserons de côté.

Fonctions ECMAScript

Supposons que nous pourrions …

- isoler une partie de code qui remplit une tâche précise
- lui donner un nom
- exécuter ce code n'importe où dans notre programme en donnant des valeurs particulières aux variables utilisées par ce code

C'est exactement le rôle d'une fonction. Comme les programmeurs le savent:

Les fonctions sont les briques d'un langage de programmation structuré.

Calculer la longueur d'un segment avec une fonction simple

Voyons un exemple simple et court. Un élément SVG 'line' a quatre attributs 'x1', 'y1', 'x2', 'y2' pour définir ses extrémités. Supposons que nous ayons fréquemment dans notre script à calculer la longueur du segment. Nous décidons de créer le morceau de code qui le fera.

Voici ce code inséré dans une page HTML. Remarquez comment le code est intégré avec la balise <script>.

```
<html>
 <head>
  <title> 'Calcul de la longueur du segment' </title>
 </head>
 <body>
  <h1> 'Longueur du segment' </h1>
  <script type="text/ecmascript">
    function LengthOfLine(x1, y1, x2, y2)
    {
       var dx = x2 - x1,
           dy = y2 - y1,
           lenSqr = dx*dx + dy*dy,
           len = Math.sqrt(lenSqr);
       return len;
    }
```

```
    var xpnt1 = 3, ypnt1 = 5, xpnt2 = 6, ypnt2 = 9;
    var lineLength = 0;

    lineLength = LengthOfLine(xpnt1, ypnt1, xpnt2, ypnt2);
    document.write("La longueur du segment est " + lineLength);  // 5 ..
  </script>
 </body>
</html>
```

Voyons ce code de près.

1. Nous utilisons le mot réservé '**function**'.

2. Nous donnons ensuite le nom de la fonction — ici LengthOfLine. Pour les noms de fonctions, ce sont les mêmes règles que pour les noms de variables.

```
function LengthOfLine
```

3. Nous accompagnons le nom de notre fonction de la donnée des arguments de la fonction, noms de variables séparés par des virgules. Ces variables seront utilisables par la fonction.

4. Le corps de la fonction est entre { et }. Nous y définissons quelques variables (dx, dy, lenSqr et len) les initialisons avec des nombres.

5. Un peu de mathématiques. Nous calculons les différences des coordonnées en x et en y, les affectons à dx et dy, puis nous utilisons le théorème de Pythagore ($c^2 = a^2 + b^2$) pour calculer le carré de la longueur du segment.

```
lenSqr = dx*dx + dy*dy
```

6. Pour avoir la longueur elle-même, nous faisons appel à une fonction prédéfinie en ECMAScript, la fonction Math.sqrt, qui calcule la racine carrée d'un nombre positif. Et nous terminons en indiquant la valeur que renvoie notre fonction avec l'instruction 'return'.

```
len = Math.sqrt(lenSqr);
return len
```

Utiliser les fonctions

Après avoir défini la fonction, nous pouvons l'appeler, c'est à dire exécuter son code.
Voyons exactement comment appeler une fonction

```
    var xpnt1 = 3, ypnt1 = 5, xpnt2 = 6, ypnt2 = 9;
  var lineLength = 0;
    lineLength = LengthOfLine(xpnt1, ypnt1, xpnt2, ypnt2);  // 5 ..
```

L'appel de la fonction se fait avec son nom suivi de la valeur de ses arguments entre parenthèses.

Nous utilisons les variables `xpnt1,ypnt1,xpnt2,ypnt2` comme arguments. Le type de donnée de la fonction est celui de la variable utilisée avec `return`. Nous pouvons utiliser la fonction n'importe où dans notre programme et utiliser la valeur qu'elle nous renvoie dans n'importe quelle expression.

La compréhension des fonctions est indispensable, non seulement pour la suite du chapitre, mais également pour le rôle qu'elles jouent dans la programmation.

Résumons sur les fonctions

Que retenir à propos des fonctions?

- Une fonction ne peut être définie qu'une fois avec le même nom.
- Une fonction particulière peut être appeler n'importe où dans le code.
- Une fonction peut s'appeler elle-même (récursivité) ou appeler d'autres fonctions.
- Une fonction peut avoir n'importe quel nombre d'arguments et même aucun.
- Si une fonction n'a pas d'argument, son nom est suivi de `()` pour sa définition et son appel.
- Le nombre des arguments et leur type doit correspondre entre la définition et l'appel..
- Les variables locales définies dans le corps d'une fonction ne peuvent être utilisées en dehors de la fonction.
- Les arguments de la fonction peuvent être considérés comme des variables locales.

Objets ECMAScript

ECMAScript est un langage destiné à travailler dans un environnement Web — le navigateur par exemple. ECMAScript a accès aux objets de cet environnement pour en tirer des renseignements, mais aussi pour les modifier ou en créer de nouveaux
Grâce à ceci, ECMAScript peut personnaliser l'environnement, lui donner de l'interactivité et du dynamisme.

Nous devons voir de près cette notion d'objets. Nous pouvons voir un objet comme un sac plein de données – ses constituants. Si ce constituant est une variable, nous le considérons comme une propriété de l'objet. Si ce constituant est une fonction, nous le nommerons méthode.

Pour accéder aux constituants de l'objet, nous utilisons le point (`.`). Pour les objets, nous avons besoin de la valeur **`null`**. Les variables qui ne sont pas associées à un objet auront cette valeur **`null`** pour le signaler.

```
Math.PI, list.length        // Propriétés d'objets ..
document.write(”hello”)     // Appel de méthodes d'objet ..
var group = null;           // variable sans valeur ..
```

Où trouvons nous ces objets?

- ECMAScript définit quelques objets.
- Le DOM nous fournit quantité d'objets très utiles.
- Les composants Java ou ActiveX proposent des objets.
- Nous pouvons créer nos propres objets avec ECMAScript.

Nous n'allons pas créer tout de suite nos objets, mais voyons les trois premières possibilités de rencontrer des objets en commençant par ceux fournis par ECMAScript:

- `Array`
- `Boolean`
- `Date`
- `Function`
- `Global`
- `Math`

- Number
- Object
- RegExp
- Error
- String

Si vous avez l'habitude de langages orientés objet, vous devez vous demander, pourquoi ne pas parler de classes ici. ECMAScript est un langage orienté objet prototype, tout est objet dans sa version actuelle 3.0 et il n'a pas de classes. ECMAScript 4.0 devrait les introduire dans le langage.

Les objets ECMAScript doivent être créés explicitement. C'est le contraire pour les autres types de données qui n'ont besoin que d'être définis. Pour créer un objet nous utilisons l'opérateur **new**. Le constructeur est une méthode de l'objet qui a le même nom que l'objet lui-même et est appelée par **new**.

```
var today = new Date(),
    christmas = new Date(2001, 12, 24);
```

Notez qu'un objet peut avoir plusieurs constructeurs avec un nombre différent d'arguments.

Voyons les deux objets les plus utiles — `Array` et `Math`.

L'objet array

Chaque langage de programmation a son propre type de données **array**. Nous pouvons considérer cet objet comme une variable qui peut contenir des valeurs multiples. C'est aussi un container.
Voyons un exemple:

```
<html>
 <head>
  <title> The Array </title>
 </head>
 <body>
  <h1> The Array </h1>
  <script type="text/ecmascript">
     var months = new Array(12); // crée un tableau de 12 éléments indéfinis
     var days = new Array("mon","tue","wen","thu","fri","sat","sun");
                                    // définit les jours de la semaine
     document.write(days[0] + "<br>");       // "mon" .. 1er élément de 'days'
..
     document.write(days[7] + "<br>");      // indéfini ..
     document.write(days.length + "<br>");  // nombre d'éléments: 7 ..
     months[1] = "feb";                 // affecte la valeur "feb" au 2ème mois
     document.write(months + "<br>");
                  // ,feb,,,,,,,,,,, un mois défini, les autres non
  </script>
</body>
</html>
```

Nous créons un objet de type tableau en appelant son constructeur. La propriété `length` peut nous renvoyer le nombre d'éléments du tableau. Pour accéder à l'un des éléments nous mettons l'indice entre crochets [ ] .

> *En ECMAScript les indices des tableaux commencent à 0 comme en C, C++ ou Java. Aussi pour un tableau de n éléments, le dernier indice défini est n-1*

ECMAScript permet trois types de constructeurs:

`new Array();`	crée un tableau vide (aucun argument)
`new Array(n);`	crée un tableau de n éléments indéfinis (un seul argument)
`new Array(x1,x2,..,xn);`	crée un tableau de n éléments définis (n arguments)

L'objet Math

Comme d'autres langages ECMAScript fournit au programmeur quelques fonctions mathématiques prédéfinies qui sont des méthodes d'un seul objet, l'objet **Math.** Vous n'avez pas besoin de créer cet objet pour accéder à ses méthodes, l'interpréteur le fait pour vous.

Donnons les méthodes les plus utiles:

- Constantes mathématiques
  - `Math.PI` pi avec l'approximation 3.1416.
  - `Math.SQRT_2` la racine carrée de 2, soit approximativement 1.4142.
  - `Math.E` la constante d'Euler e, approximativement 2.7183.

- Méthodes mathématiques
  - `Math.abs(n)` Valeur absolue `n`.
  - `Math.sin(n)` Sinus de `n (n en radians)`.
  - `Math.cos(n)` Cosinus de `n (n en radians)`.
  - `Math.atan2(y,x)` L'angle en radians du vecteur (x,y) avec l'axe des x.
  - `Math.sqrt(n)` La racine carrée de `n`.

Le modèle de document objet DOM

> *"Le DOM est une interface pour la programmation et les applications (API) pour les documents HTML et XML. Il définit la structure logique du document et la manière d'y accéder et de le manipuler". (W3C)*

D'une autre manière, le DOM est la réponse du W3C aux programmeurs HTML demandant aide devant la pagaille engendrée par DHTML. Mais c'est plus que cela, il est composé de modules.

Quand un document XML (rappelez vous, chaque fichier SVG est un document XML) est chargé par un analyseur XML, un modèle logique de nœuds est construit à partir de la structure du document XML. Avec ces nœuds, le DOM fournit une représentation complète du document affiché.

Ces noeuds sont des objets avec des propriétés et des méthodes que le programmeur peut utiliser. Depuis que les spécifications du DOM définissent les propriétés et méthodes devant être implantées, ces définitions sont des interfaces. Nous utiliserons ces interfaces de manière intensive dans ce chapitre, mais nous ne les étudierons pas toutes, nous verrons seulement:

- `Node` Interface commune aux objets SVG
- `Document` Interface pour le document.
- `Element` Interface pour les éléments.
- `Text` Interface pour les éléments 'text'.
- `Event` Interface pour les événements.

L'arborescence du DOM

Il est très utile de voir la structure des noeuds comme une arborescence d'éléments. Chaque objet correspond à une entrée du fichier XML. Voici un exemple d'arborescence.

```
<svg>
  |
  |__ <defs>
```

```
        |     |__ <circle>
        |
        |__ <g>
        |     |__ <line>
        |     |__ <polyline>
        |     |__ <rect>
        |
        |__ <g>
              |__ <g>
              |     |__ <text>
              |     |__ <path>
              |
              |__ <use>
              |__ <ellipse>
              |__ <text>
```

Comme vous le voyez, la racine de l'arbre est le document `<svg>` lui-même. Nous avons ensuite des branches, l'élément <defs> et les éléments <g>. Ces branches ont elles-mêmes des branches … des feuilles. Nous n'utiliserons pas un vocabulaire botanique mais un langage familial parent/ enfant/frère et soeur. Ainsi l'élément `<ellipse>` a deux frères et soeurs, un aîné avec l'élément `<use>` et un cadet avec l'élément `<text>` qui le suit. L'élément `<ellipse>` a un parent — un élément `<g>` — et aucun enfant. Avec ces termes nous pouvons décrire les relations entre ces éléments et leurs proches.
Nous ne devons pas oublier que chaque élément est aussi un nœud, donc qu'il contient les méthodes et propriétés de l'objet 'node'.

Grimper à l'arbre

Voici, résumées, quelques propriétés de l'interface 'node' qui nous permettent une navigation dans l'arbre du DOM

Document d'appartenance:

Valeur	Propriété	Description
`Element`	`documentElement`	L'objet document auquel appartient le noeud.

Information sur les noeuds:

Valeur	Propriété	Description
`String`	`nodeName`	nom du noeud (nom de sa balise).
`String`	`nodeValue`	Valeur du noeud. Souvent *null*, pour des noeuds texte, le contenu texte.

`Number`	`nodeType`	`ELEMENT_NODE = 1;`
		`ATTRIBUTE_NODE = 2;`
		`TEXT_NODE = 3;`
		`CDATA_SECTION_NODE = 4;`
		`ENTITY_REFERENCE_NODE = 5;`
		`ENTITY_NODE = 6;`
		`PROCESSING_INSTRUCTION_NODE = 7;`
		`COMMENT_NODE = 8;`
		`DOCUMENT_NODE = 9;`
		`DOCUMENT_TYPE_NODE = 10;`
		`DOCUMENT_FRAGMENT_NODE = 11;`
		`NOTATION_NODE = 12;`

Propriétés de navigation:

Valeur	Propriété	Description
`Node`	`ownerDocument`	L'objet document auquel appartient le noeud.
`Node`	`ParentNode`	Le noeud parent.
`Array`	`ChildNodes`	Les noeuds enfants sous forme de tableau.
`Node`	`FirstChild`	Le premier enfant du noeud.
`Node`	`LastChild`	Le dernier enfant du noeud
`Node`	`previousSibling`	Le noeud précédent au même niveau.
`Node`	`nextSibling`	Le noeud suivant au même niveau.

Des arbres aux trains

Pour illustrer la structure du DOM, je vais utiliser un train comme exemple. Voici le modèle.

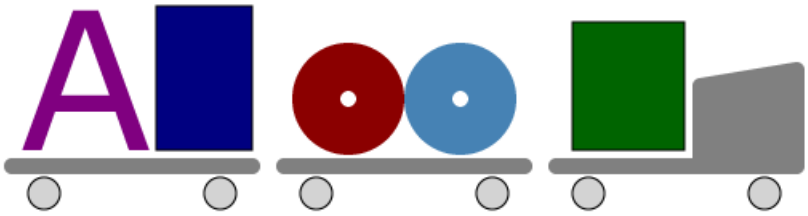

Figure 10-1. Le train de départ

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg" >
  <defs>
    <g id="essieux">
      <line x2="150" y2="0" stroke="gray" stroke-width="10" stroke-
linecap="round" />
      <circle cx="20" cy="17" r="10" fill="lightgray" stroke="black" />
      <circle cx="130" cy="17" r="10" fill="lightgray" stroke="black" />
    </g>
  </defs>
  <g id="train">
    <g id="loco" transform="translate(390, 300)" >
      <use xlink:href="#essieux" x="0" y="0" />
      <polyline id="cabine" points="150,0 150,-60 90,-50 90,0"
              stroke="gray" stroke-width="10" stroke-linejoin="round"
fill="gray" />
      <rect x="10" y="-90" width="70" height="80" fill="darkgreen"
stroke="black" />
    </g>
    <g id="wagon1" transform="translate(220,300)">
      <use xlink:href="#essieux" x="0" y="0" />
      <circle cx="40" cy="-42" r="20" stroke="darkred" stroke-width="30"
fill="none" >
      <circle cx="110" cy="-42" r="20" stroke="steelblue" stroke-width="30"
            fill="none" />
    </g>
    <g id="wagon2" transform="translate(50,300)">
      <use xlink:href="#essieux" x="0" y="0" />
      <text x="5" y="-10" font-family="Verdana" font-size="120" fill="purple" >
      A</text>
      <rect id="bluebox" x="90" y="-100" width="60" height="90" fill="navy"
          stroke="black" />
    </g>
  </g>
</svg>
```

Notre train est formé d'une locomotive, de deux wagons et de chargement — une lettre, deux caisses et deux bobines. Les essieux de la locomotive et des wagons sont identiques, aussi ils seront définis dans une section `<defs>` et réutilisés avec les éléments `<use>`. Les wagons et la locomotives seront définis comme des groupes, qui comprendront le chargement. Wagons et locomotive sont dans un groupe `train`..

Un modèle de train

Nous remplaçons le code complet par un arbre qui nous donne la structure de notre document.

```
<svg>
 |__ <defs>
 |     |__ <g id=”essieux”>
 |
 |__ <g id=”train”>
```

```
|
|__ <g id="loco">
|     |__ <use href="essieux">
|     |__ <polyline>   // cabine
|     |__ <rect>       // caisse verte
|
|__ <g id="wagon1"> // we start here !
|     |__ <use href="essieux">
|     |__ <circle>     // bobine rouge
|     |__ <circle>     // bobine bleue
|
|__ <g id="wagon2">
      |__ <use href="essieux">
      |__ <text>       // grande lettre
      |__ <rect>       // caisse bleue
```

Parcourir le train

Parcourons ce train avec l'objectif d'atteindre la caisse bleue de l'élément `wagon2`. Comme nous allons discuter les différentes possibilités d'entrer dans le DOM un peu plus tard dans ce chapitre, supposons que nous avons une variable ECMAScript `wagon1` qui contient le nœud formé par le groupe `wagon1`.

```
var wagon1;   // Par magie, le noeud wagon1 comme objet ..
```

1. Pour atteindre le document, la racine, nous écrivons simplement

```
var doc = wagon1.ownerDocument;
```

wagon1.	**ownerDocument**
<svg> |_ <defs> | |_ <g id="essieux"> | |_ <g id="train"> | | | |_ <g id="loco"> | | |_ <use href="essieux"> | | |_ <polyline> // cabine | | |_ <rect> // caisse verte | | | |_ **<g id="wagon1">** | | |_ <use href="essieux"> | | |_ <circle> // bobine rouge | | |_ <circle> // bobine bleue | | | |_ <g id="wagon2"> | |_ <use href="essieux"> | |_ <text> // grande lettre | |_ <rect> // caisse bleue	**<svg>** |_ <defs> | |_ <g id="essieux"> | |_ <g id="train"> | | | |_ <g id="loco"> | | |_ <use href="essieux"> | | |_ <polyline> // cabine | | |_ <rect> // caisse verte | | | |_ <g id="wagon1"> | | |_ <use href="essieux"> | | |_ <circle> // bobine rouge | | |_ <circle> // bobine bleue | | | |_ <g id="wagon2"> | |_ <use href="essieux"> | |_ <text> // grande lettre | |_ <rect> // caisse bleue

C'est une facilité très utile dans le DOM – accéder à l'objet `Document` depuis n'importe quel nœud d'une manière simple.

2. Vous êtes toujours dans le wagon central, et vous voulez sauter dans la locomotive. Nous écrivons

```
var loco = wagon1.previousSibling;
```

wagon1.	**previousSibling**
<svg> :	<svg> :

```
|_ <g id="train">
   |
   |_ <g id="loco">
   |  |_ <use href="essieux">
   |  |_ <polyline> // cabine
   |  |_ <rect>     // caisse verte
   |
   |_ <g id="wagon1">
   |  |_ <use href="essieux">
   |  |_ <circle>   // bobine rouge
   |  |_ <circle>   // bobine bleue
   :
```

```
|_ <g id="train">
   |
   |_ <g id="loco">
   |  |_ <use href="essieux">
   |  |_ <polyline> // cabine
   |  |_ <rect>     // caisse verte
   |
   |_ <g id="wagon1">
   |  |_ <use href="essieux">
   |  |_ <circle>   // bobine rouge
   |  |_ <circle>   // bobine bleue
   :
```

3. Nous réalisons que ce n'est pas là que nous voulions aller mais dans le wagon de queue …

```
var wagon2 = loco.nextSibling.nextSibling;
```

loco.	**nextSibling.**	**nextSibling**

<svg>
:
|_<g id="train">
  |
  |_**<g id="loco">**
  | |_<use href="essieux">
  | |_<polyline> // cabine
  | |_<rect> // caisse verte
  |
  |_<g id="wagon1">
  | |_<use href="essieux">
  | |_<circle> // bobine rouge
  | |_<circle> // bobine bleue
  |
  |_<g id="wagon2">
  |_<use href="essieux">
  |_<text> // grande lettre
  |_<rect> // caisse bleue
	<svg>
	:
	|_<g id="train">
	  |
	  |_<g id="loco">
	  | |_<use href="essieux">
	  | |_<polyline> // cabine
	  | |_<rect> // caisse verte
	  |
	  |_**<g id="wagon1">**
	  | |_<use href="essieux">
	  | |_<circle> // bobine rouge
	  | |_<circle> // bobine bleue
	  |
	  |_<g id="wagon2">
	  |_<use href="essieux">
	  |_<text> // grande lettre
	  |_<rect> // caisse bleue
		<svg>
		:
		|_<g id="train">
		  |
		  |_<g id="loco">
		  | |_<use href="essieux">
		  | |_<polyline> // cabine
		  | |_<rect> // caisse verte
		  |
		  |_<g id="wagon1">
		  | |_<use href="essieux">
		  | |_<circle> // bobine rouge
		  | |_<circle> // bobine bleue
		  |
		  |_**<g id="wagon2">**
		  |_<use href="essieux">
		  |_<text> // grande lettre
		  |_<rect> // caisse bleue

Vous voyez que `previousSibling` et `nextSibling` permettent de parcourir les noeuds d'un même niveau. N'oublions pas notre caisse bleue!

4. Pour atteindre cette caisse, nous devons descendre d'un niveau et le parcourir.

```
var blueBox = wagon2.firstChild.nextSibling.nextSibling;
```

wagon2.firstChild	**nextSibling.**	**nextSibling**

<svg>
:
|_<g id="train">
  :
  |_<g id="wagon2">
  |_**<use href="essieux">**
	<svg>
	:
	|_<g id="train">
	  :
	  |_<g id="wagon2">
	  |_<use href="essieux">
		<svg>
		:
		|_<g id="train">
		  :
		  |_<g id="wagon2">
		  |_<use href="essieux">

|_<text> // grande lettre |_<rect> // caisse bleue	|_**<text> // grande lettre** |_<rect> // caisse bleue	|_<text> // grande lettre |_**<rect> // caisse bleue**

ou plus simplement

```
var blueBox = wagon2.lastChild;
```

wagon2.	**lastChild**

<svg>
:
|_<g id="train">
  :
  |_**<g id="wagon2">**
  |_<use href="essieux">
  |_<text> // grande lettre
  |_<rect> // caisse bleue
	<svg>
	:
	|_<g id="train">
	  :
	  |_<g id="wagon2">
	  |_<use href="essieux">
	  |_<text> // grande lettre
	  |_**<rect> // caisse bleue**

Pour accéder à un descendant d'un noeud `wagon2`, nous pouvons aller au premier descendant avec `wagon2.firstChild` et parcourir ce niveau avec `nextSibling` , ou aller au dernier descendant avec `wagon2.lastChild` pour parcourir ce niveau avec `previousSibling`.

Ayant trouvé notre caisse bleue, nous voulons retourner dans la cabine de la locomotive.

5. Nous devons remonter d'un niveau, aller à la locomotive et redescendre d'un niveau pour la cabine.

```
var cabine = blueBox.parentNode.previousSibling.previousSibling.firstChild.nextSibling;
```

blueBox.parentNode.	**previousSibling.previousSibling.**	**firstChild.nextSibling**
<svg> : |_<g id="train"> | |_<g id="loco"> 	|_<use href="essieux"> 	|_<polyline> // cabine 	|_<rect> // caisse verte | |_<g id="wagon1"> 	|_<use href="essieux"> 	|_<circle> // bobine rouge 	|_<circle> // bobine bleue | |_**<g id="wagon2">** |_<use href="essieux"> |_<text> // grande lettre 	<rect> // caisse	<svg> : |_<g id="train"> | |_**<g id="loco">** 	|_<use href="essieux"> 	|_<polyline> // cabine 	|_<rect> // caisse verte | |_<g id="wagon1"> 	|_<use href="essieux"> 	|_<circle> // bobine rouge 	|_<circle> // bobine bleue | |_<g id="wagon2"> |_<use href="essieux"> |_<text> // grande lettre |_<rect> // caisse bleue	<svg> : |_<g id="train"> | |_<g id="loco"> 	|_<use href="essieux"> 	|_**<polyline> // cabine** 	|_<rect> // caisse verte | |_<g id="wagon1"> 	|_<use href="essieux"> 	|_<circle> // bobine rouge 	|_<circle> // bobine bleue | |_<g id="wagon2"> |_<use href="essieux"> |_<text> // grande lettre 	<rect> // caisse

bleue		bleue

Nous pouvons remonter d'un niveau depuis n'importe quel noeud avec `parentNode` — nous ne sommes pas obligés de nous mettre au début ou à la fin de la liste de noeuds de même niveau. Ce qui nous donne une autre manière d'arriver à la cabine.

6. Nous pouvons remonter de deux niveaux pour le noeud `train` et de là redescendre au lieu de parcourir tout le train.

```
var cabine = blueBox.parentNode.parentNode.firstChild.firstChild.nextSibling;
```

Nous pouvons maintenant aller d'un endroit à un autre de notre train. Toutefois, imaginons que le train ait de très nombreux wagons, et que de l'un des wagons à peu près au milieu du train, nous voulions regagner la locomotive. Plutôt que d'utiliser maintes fois `nextSibling`, le W3C nous fournit une alternative pour ce code.

7. Cette alternative consiste à atteindre tous les descendants d'un noeud sous forme de tableau. Ce que nous pouvons obtenir avec

```
var wagons = loco.parentNode.childNodes;
```

Et nous allons directement au 142^{ième} wagon simplement avec

```
var wagon142 = wagons.item(141);
```

Les tableaux commencent avec l'indice 0 pour ECMAScript et le DOM. Le 142^{ième} élément aura l'indice 141.

Sur ces bases, je vous propose une fonction qui permet de visiter tous les noeuds descendants. Cette fonction est récursive, elle s'appelle elle-même.

```
function VisitChildren(node)
{
   for (var child = node.firstChild; child != null; child =
child.nextSibling)
   {
      // faire quelque chose pour ce noeud (sauf le supprimer!) ..
      if (child.hasChildNodes())
         VisitChildren(child);
   }
}
```

Cette fonction peut vous être très utile dans vos projets.

Préparons un exemple complet

Nous allons construire un document complet. Nous devons tout d'abord résoudre deux choses
- Nous avons besoin d'un événement que nous verrons plus loin dans ce chapitre.
- Nous devons gérer les nœuds additionnels – les nœuds avec un espace ou un retour chariot.

Pour l'événement, nous l'utiliserons et repoussons son explication. Pour les nœuds additionnels, c'est moins évident. Quels sont ces nœuds?

```
<g id="carriage">¶
  <line x2="150" y2="0" stroke="gray" stroke-width="10" stroke-linecap="round" />¶
  <circle cx="20" cy="17" r="10" fill="lightgray" stroke="black" />¶
  <circle cx="130" cy="17" r="10" fill="lightgray" stroke="black" />¶
</g>¶
```

Chaque document XML peut avoir des noeuds texte – noeuds uniquement composés de texte. Ce texte peut n'être constitué que d'espaces (·) ou de retours chariot (¶). L'interpréteur XML interprétera ces nœuds comme des nœuds texte – nous les nommerons '**whitespace node**'.

Sur ce fragment SVG, vous pouvez voir un certain nombre de 'whitespace nodes'. Le problème est que si de l'élément 'line' nous utilisons nextSibling, nous ne nous retrouvons pas sur le noeud de l'élément 'circle' mais sur un 'whitespace node'.

Que faire?

- Traiter avec eux.
- Les supprimer, une fois le document chargé.

Nous décidons d'utiliser la deuxième solution – supprimer tous ces 'whitespace nodes'. Pour cela, nous créons une fonction qui le fait

```
function RemoveWhiteSpaceChildNodesOf(node)
{
  if (node != null)
  {
    var child = node.lastChild;
    while (child != null)
    {
      if (child.nodeType == 3 && child.nodeValue.match(/\S/) == null)
      {
        var previous = child.previousSibling;
        child.parentNode.removeChild(child);
        child = previous;
      }
      else
      {
        RemoveWhiteSpaceChildNodesOf(child);
        child = child.previousSibling;
      }
    }

  }
}
```

Vous ne comprenez peut-être pas tout dans cette fonction pour l'instant. Mais nous supposons qu'elle fonctionne et comme nous voulons pouvoir la réutiliser dans d'autres documents nous la mettons dans un fichier externe "RemoveWhiteSpace.js". Nous pouvons intégrer cette fonction à notre document simplement avec:

```
  <script type="text/ecmascript" xlink:href="RemoveWhiteSpace.js" />
```

Notons que ces noeuds "vides" ne nous posent des problèmes qu'avec ce genre de navigation utilisant 'firstChild', 'lastChild', 'nextSibling', 'previousSibling'.

Voici notre parcours dans le train comme un document complet

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
onload="Init(evt);">
  <script type="text/ecmascript" xlink:href="RemoveWhiteSpace.js"/>
  <script type="text/ecmascript">
  <![CDATA[
    function Init(evt)
    {
       RemoveWhiteSpaceChildNodesOf(evt.target.ownerDocument.documentElement);

       var blueBox = evt.target.ownerDocument.getElementById("bluebox");
```

```
      var cabine =
blueBox.parentNode.parentNode.firstChild.firstChild.nextSibling;
      window.alert(cabine.nodeName);
  ]]>
  </script>
  <defs>
    <g id="essieux">
       <line   x2="150"   y2="0"   stroke="gray"   stroke-width="10"   stroke-
linecap="round" />
       <circle cx="20" cy="17" r="10" fill="lightgray" stroke="black" />
       <circle cx="130" cy="17" r="10" fill="lightgray" stroke="black" />
    </g>
  </defs>
  <g id="train">
    <g id="loco" transform="translate(390, 300)" >
      <use xlink:href="#essieux" x="0" y="0" />
      <polyline id="cabine" points="150,0 150,-60 90,-50 90,0"
                stroke="gray" stroke-width="10" stroke-linejoin="round"
fill="gray" />
      <rect x="10" y="-90" width="70" height="80" fill="darkgreen"
stroke="black" />
    </g>
    <g id="wagon1" transform="translate(220,300)">
      <use xlink:href="#essieux" x="0" y="0" />
      <circle cx="40" cy="-42" r="20" stroke="darkred" stroke-width="30"
fill="none" />
      <circle cx="110" cy="-42" r="20" stroke="steelblue" stroke-width="30"
fill="none" />
    </g>
    <g id="wagon2" transform="translate(50,300)">
      <use xlink:href="#essieux" x="0" y="0" />
      <text x="5" y="-10" font-family="Verdana" font-size="120" fill="purple"
>A</text>
      <rect id="bluebox" x="90" y="-100" width="60" height="90" fill="navy"
stroke="black" />
    </g>
  </g>
</svg>
```

Notez:

- L'événement `onload` qui appelle la fonction `Init(evt)` quand le document est chargé (Ce qui ne veut pas dire que tout le DOM est construit!)
- L'utilisation de l'élément `<script type="text/ecmascript">`. Nous pouvons omettre l'attribut `type="text/ecmascript`, car `="text/ecmascript"` est sa valeur par défaut.
- La section `<![CDATA[ .. ]]`, qui fait que son contenu n'est pas pris en compte par l'interpréteur XML. Ceci nous permet d'utiliser les caractères réservés à XML comme '`<`' dans notre code.

Recherche approfondie

La navigation pas à pas dans le DOM est très utile si nous

- parcourons de courtes distances dans l'arbre.
- connaissons la structure de l'arbre.
- devons parcourir l'arbre entièrement.

Ce n'est pratiquement jamais le cas. Le plus souvent nous cherchons un noeud particulier et voulons nous y rendre. Nous possédons un peu d'information sur ce nœud. L'objet `Document` offre des possibilités intéressantes pour résoudre ce problème.

Recherche dans le document

Ces méthodes nous permettent d'aller directement à l'élément voulu.

Navigation dans le document:

Valeur	Méthode	Description
`Array`	`getElementsByTagName(tagName)`	Un tableau des éléments de ce type.

Element	getElementsById(elementId)	Retourne l'élément avec cet 'id'.

1. Supposons par exemple que nous savons que le noeud que nous souhaitons visiter est un élément <g>. Depuis la locomotive, nous écrivons

```
var elements = loco.ownerDocument.getElementsByTagName(”g”);
```

la variable 'elements' sera un tableau avec tous les éléments <g>.

2. Nous pouvons parcourir cette liste avec une boucle 'for'.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
onload="Init(evt);">
  <script type="text/ecmascript" xlink:href="RemoveWhiteSpace.js"/>
  <script type="text/ecmascript">
  <![CDATA[
    function Init(evt)
    {
       RemoveWhiteSpaceChildNodesOf(evt.target.ownerDocument.documentElement);

       var loco = evt.target.ownerDocument.getElementById("loco");
       var elements = loco.ownerDocument.getElementsByTagName("g");
       for (var i=0; i < elements.length; i++)
          window.alert("Elément " + i + " de type: " +
elements.item(i).nodeName);
    }
  ]]>
  </script>
  <!-- éléments du train -->
</svg>
```

Nous aurons alors les résultats suivants:

```
Elément 0 de type: g
Elément 1 de type: g
Elément 2 de type: g
Elément 3 de type: g
Elément 4 de type: g
```

Si nous regardons les éléments du train nous comptons cinq éléments <g> y compris le groupe `essieux` dans la section `<defs>`. Bien, nous connaissons peut-être un renseignement supplémentaire sur l'élément cherché — la couleur ou un autre attribut.

Mais il y a une autre méthode offerte par l'objet `document`. Et c'est la solution la plus efficace. Supposons que nous avons repéré notre élément avec un identificateur unique défini par l'attribut `id`. Nous pouvons utiliser la méthode `getElementById` de l'objet `document` pour aller directement à cet élément.

3. Toujours depuis notre locomotive, nous décidons de visiter le dernier wagon, nous écrivons

```
var mostDistantWagon = loco.ownerDocument.getElementById(“wagon2”);
```

Nous y sommes. Nous avons là une méthode simple et efficace. Un élément quelconque de l'arborescence est accessible rapidement. Pour que cette méthode fonctionne, il faut que tous les éléments aient un attribut 'id' avec un identificateur unique.

La valeur de l'attribut 'Id' doit être unique dans tout le document XML et obéir aux règles sur les noms de variables de ECMAScript.

Accéder aux attributs

Nous avons évoqué cette question, ce n'est pas très mystérieux. Il y a deux méthodes de l'élément pour nous le permettre.

Accès aux attributs des éléments:

Valeur	Méthode	Description
`String`	`getAttribute(attrName)`	La valeur de l'attribut nommé comme chaîne
`void`	`setAttribute(attrName, newValue)`	Affecte à l'attribut nommé la nouvelle valeur

Examinons la méthode `getAttribute` .

Revenons à notre étape de tout à l'heure, nous avions une liste des éléments <g> et nous voulions trouver le groupe qui avait l'identificateur `wagon2`. Négligeons la méthode `getElementById` et utilisons nos connaissances toutes fraîches sur les attributs. Nous pouvons parcourir le tableau et comparer l'identificateur de chaque groupe avec la chaîne `"wagon2"`.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
onload="Init(evt);">
  <script type="text/ecmascript" xlink:href="RemoveWhiteSpace.js"/>
  <script type="text/ecmascript">
  <![CDATA[
    function Init(evt)
    {
       RemoveWhiteSpaceChildNodesOf(evt.target.ownerDocument.documentElement);
       var loco = evt.target.ownerDocument.getElementById("loco");
       var groups = loco.ownerDocument.getElementsByTagName("g");
       var wagon = null;  // null tant qu'il n'est pas trouvé ..
       var currentGroup = null;
       for (var i=0; wagon == null && i < groups.length; i++)
       {
          currentGroup = groups.item(i);
          if (currentGroup.getAttribute("id") == "wagon2") // trouvé :-)
          wagon = currentGroup;
       }

       if (wagon != null)
          window.alert("wagon nommé = '" + wagon.getAttribute("id") + "'
trouvé!");
    }
  ]]>
  </script>
  <!-- éléments du train -->
</svg>
```

Nous avons le résultat suivant:

```
      wagon nommé = 'wagon2' trouvé!
```

Cette technique de sélection des éléments en regardant leurs attributs est utile quand nous ne pouvons utiliser la méthode `getElementById` parce que par exemple, des éléments n'ont pas d'identificateur.

Modifier des éléments

Comme nous savons naviguer à travers le DOM, il est temps de voir comment modifier les attributs des éléments rencontrés. La méthode `setAttribute` est exactement faite pour cela.

Modifier les attributs

Votre téléphone sonne et votre client a changé d'avis, il voudrait un Y a la place du A et puis aussi que la caisse de la locomotive soit deux fois plus haute et de couleur or. Avec vos

nouvelles connaissances du DOM, vous décidez de le faire en manipulant les objets par le script au lieu de corriger vos attributs directement dans les éléments.

1. Modifier la caisse est très simple. Vous n'avez qu'à changer les attributs correspondants.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
onload="Init(evt);">
  <script type="text/ecmascript" xlink:href="RemoveWhiteSpace.js"/>
  <script type="text/ecmascript">
  <![CDATA[
    function Init(evt)
    {
       RemoveWhiteSpaceChildNodesOf(evt.target.ownerDocument.documentElement);
       var loco = evt.target.ownerDocument.getElementById("loco");
       var greenBox       = loco.lastChild,
           greenBoxHeight = parseFloat(greenBox.getAttribute("height")),
           greenBoxY      = parseFloat(greenBox.getAttribute("y"));
       greenBox.setAttribute("height", 2*greenBoxHeight);
       greenBox.setAttribute("y", greenBoxY - greenBoxHeight);
       greenBox.setAttribute("fill", "gold");
    }
  ]]>
  </script>
  <!-- éléments du train -->
</svg>
```

2. Sagement assis dans la locomotive, vous admirez le travail.

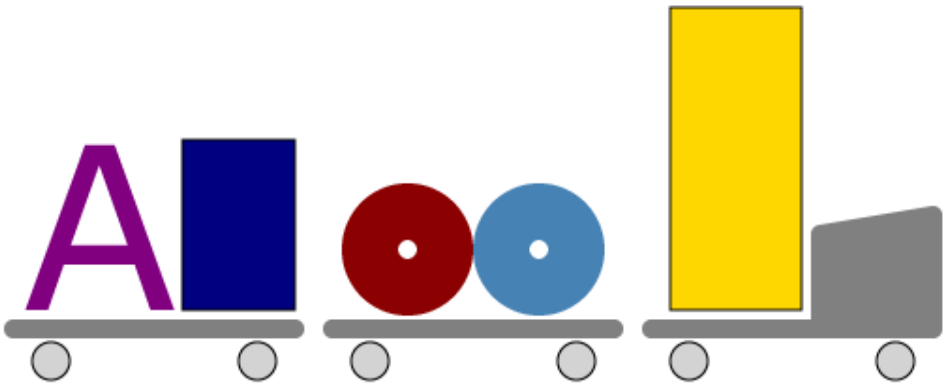

Figure 10-2. La caisse de la locomotive est modifiée

C'est bien le résultat souhaité. Regardons le code de près.

3. Notons que le second argument de la méthode setAttribute peut être aussi bien un nombre qu'une chaîne. Nous ne nous occupons pas du type de donnée, si une conversion est nécessaire, elle sera faite par la méthode elle-même. Comme nous changeons la hauteur de la caisse et que sa position est donnée par son angle supérieur gauche, nous devons modifier la position de ce point en compensant le changement de hauteur.

Nous devons maintenant remplacer A par Y. Nous devons aller dans le dernier wagon, nous connaissons son identificateur, c'est très facile. Nous décidons de chercher tous les éléments <text> descendants de `wagon2`. Nous ne devrions en trouver qu'un, notre lettre A.

4. Le contenu du texte est le seul descendant du noeud <text>

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
onload="Init(evt);">
  <script type="text/ecmascript" xlink:href="RemoveWhiteSpace.js"/>
  <script type="text/ecmascript">
  <![CDATA[
    function Init(evt)
    {
       RemoveWhiteSpaceChildNodesOf(evt.target.ownerDocument.documentElement);
       var loco = evt.target.ownerDocument.getElementById("loco");
       var wagon2 = loco.ownerDocument.getElementById("wagon2");
       var texts = wagon2.getElementsByTagName("text"); // éléments texte
       if (texts.length > 0)                              // ok, c'est notre 'A'
..
       {
          var letter = texts.item(0).firstChild; // chaîne comme descendant
          letter.nodeValue = "Y";                // la lettre est Y.
       }
          else                    // oops
             window.alert("Où est passé la lettre A!");
    }
  ]]>
  </script>
  <!-- éléments du train initiaux -->
</svg>
```

Le résultat est bien celui escompté.

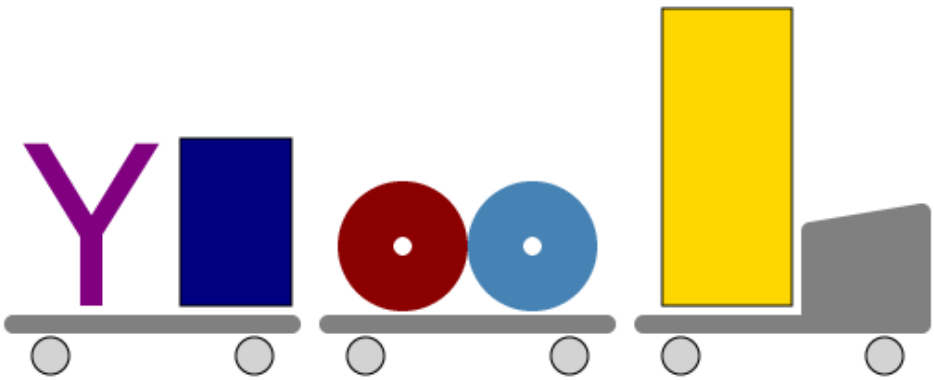

Figure 10-3. Changement de la lettre, Y au lieu de A

Nous pouvons maintenant essayer de supprimer des éléments ou des attributs.

Supprimer éléments et attributs

Avec le DOM, comme dans la réalité, il est toujours plus facile de détruire que de construire.

Méthode pour supprimer un noeud:

`Node`	`removeChild(node)`	supprime ce noeud de l'arborescence.

Méthode pour supprimer un attribut:

`Node`	`removeAttribute(attr)`	supprime l'attribut de l'élément.

Avant de passer à la destruction, voyons ces règles:

> *Tout noeud - et avec lui tout élément - peut être supprimé par son parent.*
> *Un attribut ne peut être supprimé que par l'élément auquel il appartient.*

Comme toujours, nous comprendrons mieux avec un exemple. Vous êtes toujours dans la locomotive, vous arrivez chez votre premier client et vous avez une bonne chance de débarquer vos colis par script! Voilà notre plan:

- Aller de `loco` à `wagon1`.
- Décharger les colis en combinant les méthodes `lastChild` et `removeChild`.
- Enlever l'emballage de la lettre Y avec la méthode `removeAttribute`.

1. D'où immédiatement ce code:

```
var wagon1 = loco.ownerDocument.getElementById("wagon1");
wagon1.removeChild(wagon1.lastChild); // décharge la bobine bleue ..
wagon1.removeChild(wagon1.lastChild); // décharge la bobine rouge ..
```

2. Et le résultat:

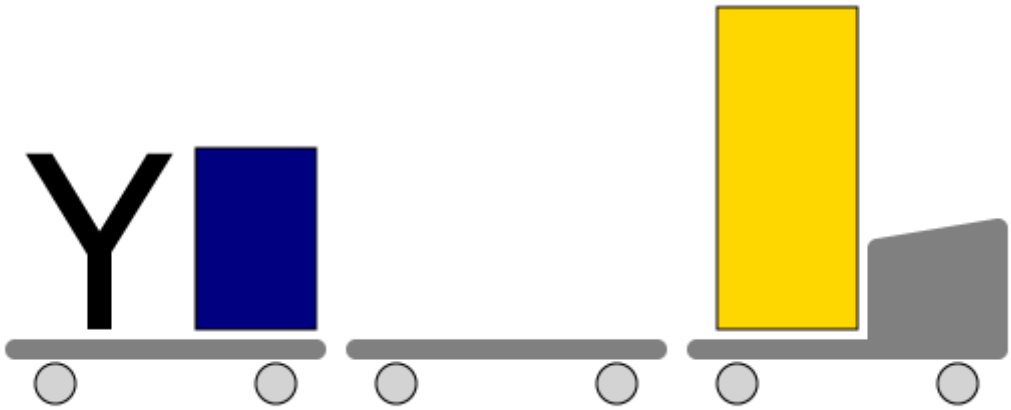

Figure 10-4. Déchargement du premier wagon

Bien sûr, mais ce n'est pas si facile que nous pourrions le croire.

3. Examinons la branche `wagon1`.

```
|__ <g id="wagon1">
|     |__ <use href="essieux">
|     |__ <circle>     // bobine rouge
|     |__ <circle>     // bobine bleue
|
```

4. Après le déchargement de la bobine bleue, la branche devient:

```
|__ <g id="wagon1">
|     |__ <use href="essieux">
|     |__ <circle>     // bobine rouge
|
```

Qu'arrive-t-il si vous écrivez ce code:

```
wagon1.removeChild(wagon1.lastChild.previousSibling);
```

pour décharger la bobine rouge. Elle est devenue le dernier descendant, et vous supprimer les essieux.

5. Enlevons également l'emballage de la lettre Y du dernier wagon avec `removeAttribute`.

```
var                                letterY                                =
loco.ownerDocument.getElementById("wagon2").firstChild.nextSibling;
letterY.removeAttribute("fill");      // enlève l'emballage pourpre ..
```

Nous avons ce que nous voulions. La branche `wagon1` est devenue:

```
|__ <g id="wagon1">
|     |__ <use href="essieux">
|
```

6. Pour supprimer tous les descendants d'un noeud, vous pouvez utiliser cette boucle pour les supprimer en commençant par le premier

```
while (parent.firstChild != null)
   parent.removeChild(parent.firstChild);
```

7. Ou en commençant par le dernier

```
while (parent.firstChild != null)
   parent.removeChild(parent.lastChild);
```

Quand vous supprimez des éléments, spécialement dans des boucles, n'oubliez pas que la suppression d'un élément change l'arborescence et l'action se poursuit sur cette nouvelle structure.

Créer des éléments

Soyons constructifs! Quelles sont les méthodes pour créer des éléments? Pour les attributs nous avons vu la méthode `setAttribute`.

Création d'éléments:

Valeur	Méthode	Description
`Element`	`createElement(tagName)`	Crée un nouvel élément du type indiqué.

Création de noeuds:

Valeur	Method	Description
`Node`	`cloneNode(deep)`	Crée un clone.

Insertion des noeuds:

Valeur	Méthode	Description
`Node`	`insertBefore(node, child)`	Insère *node* avant *child* dans la liste des descendants.
`Node`	`appendChild(node)`	Ajoute *node* à la fin de la liste des descendants.
`Node`	`replaceChild(node, child)`	Remplace *child* par *node* dans la liste des descendants.

L'objet `Document` possède les méthodes de création d'éléments. Nous commençons donc par chercher l'objet `Document` — ce qui se fait facilement avec la propriété `ownerDocument` de n'importe quel noeud — pour ensuite utiliser la méthode `createElement`. Nous lui passons comme argument "`rect`", "`circle`" ou tout autre nom d'élément que nous voulons créer.

Un objet `Node` a quelques possibilités de création — la possibilité de se cloner. L'argument `deep` est de type booléen, avec la valeur `true`, le clone aura les mêmes descendants que l'original et avec la valeur `false` seul le noeud lui-même sera cloné sans ses descendants.

Etendre le DOM

Nous pouvons enrichir le DOM avec ces deux étapes:
1. Créer un nouveau noeud avec la méthode `createElement` de l'objet document.
2. Placer ce nouveau nœud dans l'arborescence avec l'une des méthodes d'insertion.

Si nous voulons seulement déplacer un noeud, nous pouvons le supprimer puis l'insérer à la place souhaitée.

Revenons à notre train

Reprenons notre exemple, vous venez de décharger les bobines et votre client vous demande de transporter une pyramide olive et un wagon défectueux. Vous décidez donc de la stratégie suivante:

- Faire passer la caisse bleue du dernier wagon au premier pour pouvoir mettre le wagon défectueux sur ce dernier wagon.
- De même pour la lettre **Y** .
- Prendre le wagon défectueux.
- Le charger sur le dernier wagon
- Prendre la pyramide.
- La mettre sur le wagon défectueux.

1. Pour déplacer la caisse et la lettre, vous écrivez

```
var wagon1 = loco.ownerDocument.getElementById("wagon1"),
    wagon2 = loco.ownerDocument.getElementById("wagon2");

wagon1.appendChild(wagon2.lastChild); // la caisse bleue sur le 1er  wagon.
wagon1.appendChild(wagon2.lastChild); // La lettre Y égalemement
```

“Mais pourquoi la caisse et la lettre ont-elles changé de place? Vous n'avez changé aucune coordonnée!” demande le client. Vous expliquez patiemment à ce SVGiste amateur, que les wagons sont des groupes avec un système local de coordonnées. La caisse et la lettre ont localement la même position. S'il insiste, vous lui conseillez de relire le chapitre 6 de ce livre.

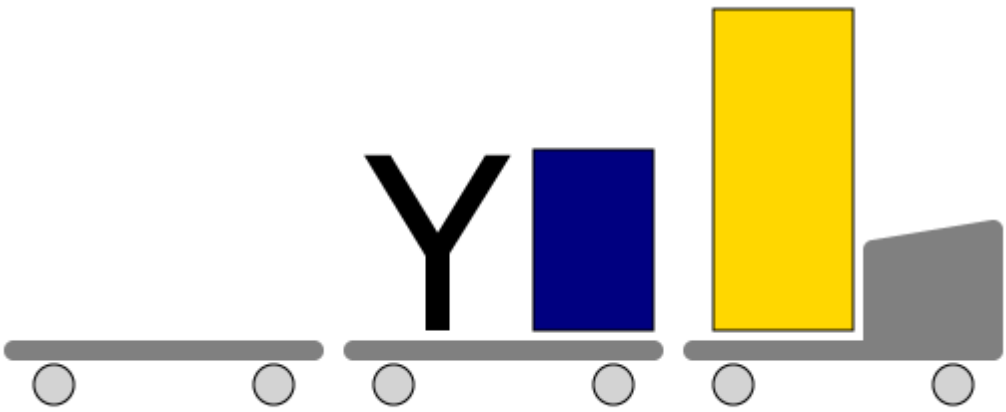

Figure 10-5. Déplacement des objets

2. La suite de notre plan — charger le wagon défectueux sur le dernier wagon désormais vide. Nous clonons le dernier wagon pour créer ce wagon, l'insérons avec la méthode `appendChild`.

```
var wagon1 = loco.ownerDocument.getElementById("wagon1"),
    wagon2 = loco.ownerDocument.getElementById("wagon2");
    wagon1.appendChild(wagon2.lastChild);
     wagon1.appendChild(wagon2.lastChild);

     var defectWagon = wagon2.cloneNode(true);
     defectWagon.setAttribute("transform", "translate(0,-35)");
     wagon2.appendChild(defectWagon);
```

3. Nous devons également créer la pyramide — comme un élément `<polygon>` — et la placer sur le wagon défectueux.

```
var wagon1 = loco.ownerDocument.getElementById("wagon1"),
    wagon2 = loco.ownerDocument.getElementById("wagon2");

wagon1.appendChild(wagon2.lastChild);
wagon1.appendChild(wagon2.lastChild);

var defectWagon = wagon2.cloneNode(true);
defectWagon.setAttribute("transform", "translate(0,-35)");

var pyramide = loco.ownerDocument.createElement("polygon");
pyramide.setAttribute("points", "5,-10 145,-10 75,-100");
pyramide.setAttribute("fill", "olive");
defectWagon.appendChild(pyramide);

wagon2.appendChild(defectWagon);
```

4. Voyons le résultat.

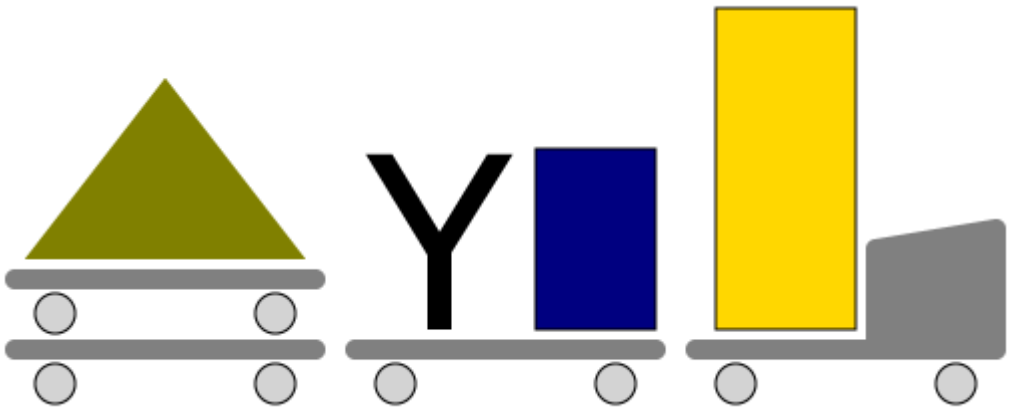

Figure 10-6. Ajouter des objets

Nous venons de voir comment créer de nouveaux éléments SVG et les placer dans le DOM existante. Vous devez aussi savoir que l'objet `Document` possède de nombreuses autres méthodes de création. Vous pouvez même créer un nouveau `Document` à partir de l'objet `DOMImplementation` . Mais nous ne décrirons pas toutes ces méthodes ici.

> *Vous devez cependant savoir que les opérations de création et de suppression se font en mémoire. Ces opérations sont gourmandes en temps d'exécution et en mémoire et il est préférable de se contenter ce modifier les attributs quand c'est possible*

Une méthode plus économique

Si vous devez faire apparaître et disparaître des éléments, au lieu de les créer et les supprimer, vous pouvez utiliser les propriétés de visibilité. Ils restent dans le DOM et vous contrôlez leur visibilité avec

```
wagon1.setAttribute("visibility","hidden");
```

pour que wagon1 ne soit plus rendu. Vous avez un code simple et économe!

Pour terminer, voici le code complet de notre exemple avec toutes les actions réalisées

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
```

```
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
onload="Init(evt);">
  <script type="text/ecmascript" xlink:href="RemoveWhiteSpace.js"/>
  <script type="text/ecmascript">
  <![CDATA[
    function Init(evt)
    {
       RemoveWhiteSpaceChildNodesOf(evt.target.ownerDocument.documentElement);
       var loco = evt.target.ownerDocument.getElementById("loco");

       FindTrailer2ByAttribute(loco);
       MakeGreenBoxGolden(loco);
       ChangeAToY(loco);
       RemoveCoils(loco);
       MoveBoxAndLetterAndCreateDefectWagonAndPyramid(loco);
    }

    function FindTrailer2ByAttribute(loco)
    {
       var groups = loco.ownerDocument.getElementsByTagName("g");
       var wagon = null;
       var currentGroup = null;

       for (var i=0; wagon == null && i < groups.length; i++)
       {
          currentGroup = groups.item(i);
          if (currentGroup.getAttribute("id") == "wagon2") // found :-)
             wagon = currentGroup;
       }
       if (wagon != null)
          window.alert("group found with id: " + wagon.getAttribute("id"));
       else
          window.alert("no group found with id 'wagon2'");
    }

    function MakeGreenBoxGolden(loco)
    {
       var greenBox       = loco.lastChild,
           greenBoxHeight = parseFloat(greenBox.getAttribute("height")),
           greenBoxY      = parseFloat(greenBox.getAttribute("y"));

       greenBox.setAttribute("height", 2*greenBoxHeight);
       greenBox.setAttribute("y", greenBoxY - greenBoxHeight);
       greenBox.setAttribute("fill", "gold");
    }

    function ChangeAToY(loco)
    {
       var wagon2 = loco.ownerDocument.getElementById("wagon2");
       var texts = wagon2.getElementsByTagName("text");

       if (texts.length > 0) // ok, it must be our 'A' ..
       {
          var letter = texts.item(0).firstChild;
          letter.nodeValue = "Y";
       }
    }

     function RemoveCoils(loco)
     {
       var wagon1 = loco.ownerDocument.getElementById("wagon1");

       wagon1.removeChild(wagon1.lastChild); // décharge la bobine bleue ..
       wagon1.removeChild(wagon1.lastChild); // décharge la bobine rouge ..
       var                                  letterY                                  =
loco.ownerDocument.getElementById("wagon2").firstChild.nextSibling;
       letterY.removeAttribute("fill");          // enlève l'emballage ..
     }

     function MoveBoxAndLetterAndCreateDefectWagonAndPyramid(loco)
     {
        var wagon1 = loco.ownerDocument.getElementById("wagon1"),
            wagon2 = loco.ownerDocument.getElementById("wagon2");

        wagon1.appendChild(wagon2.lastChild);
        wagon1.appendChild(wagon2.lastChild);
```

```
        var defectWagon = wagon2.cloneNode(true);
        defectWagon.setAttribute("transform", "translate(0,-35)");

        var pyramide = loco.ownerDocument.createElement("polygon");
        pyramide.setAttribute("points", "5,-10 145,-10 75,-100");
        pyramide.setAttribute("fill", "olive");
        defectWagon.appendChild(pyramide);

        wagon2.appendChild(defectWagon);
   }
  ]]>
  </script>
  <defs>
    <g id="essieux">
     <line x2="150" y2="0" stroke="gray" stroke-width="10" stroke-
linecap="round" />
       <circle cx="20" cy="17" r="10" fill="lightgray" stroke="black" />
       <circle cx="130" cy="17" r="10" fill="lightgray" stroke="black" />
    </g>
  </defs>
  <g id="train">
    <g id="loco" transform="translate(390, 300)" >
      <use xlink:href="#essieux" x="0" y="0" />
      <polyline id="cabine" points="150,0 150,-60 90,-50 90,0"
                stroke="gray" stroke-width="10" stroke-linejoin="round"
fill="gray" />
      <rect x="10" y="-90" width="70" height="80" fill="darkgreen"
stroke="black" />
    </g>
    <g id="wagon1" transform="translate(220,300)">
      <use xlink:href="#essieux" x="0" y="0" />
      <circle cx="40" cy="-42" r="20" stroke="darkred" stroke-width="30"
fill="none" />
      <circle cx="110" cy="-42" r="20" stroke="steelblue" stroke-width="30"
fill="none" />
    </g>
    <g id="wagon2" transform="translate(50,300)">
      <use xlink:href="#essieux" x="0" y="0" />
      <text x="5" y="-10" font-family="Verdana" font-size="120" fill="purple"
>A</text>
      <rect id="bluebox" x="90" y="-100" width="60" height="90" fill="navy"
                        stroke="black" />
    </g>
  </g>
</svg>
```

Les événements

Au début de ce chapitre, je vous ai promis des documents dynamiques et interactifs. Pour l'instant il n'en est rien!

Qu'est ce que l'interactivité? Imaginons un utilisateur disant “Je veux que ce rectangle vert soit rouge!” Si ECMAScript modifie le DOM du document, et change la couleur de la forme, vous avez un document interactif. C'est ce que nous allons faire.

Le logiciel qui implémente le DOM — le visualiseur SVG ou le navigateur avec le plugin — transmet certains événements au document. Pour un événement donné, le DOM regarde si une action est enregistrée pour cet événement particulier. Un élément peut être lui même un récepteur pour cet événement avec un morceau de code ECMAScript qui est exécuté quand survient cet événement. Voici un exemple:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg">
  <circle   cx="80"   cy="120"   r="40"   fill="red"   onclick="window.alert('hello
user!');" />
```

```
</svg>
```

L'élément `<circle>`, cercle rouge est enregistré comme un récepteur grâce à la propriété `onclick` dans ce cas. La valeur de cette propriété — 'event handler' — est une déclaration ECMAScript qui sera exécutée si l'utilisateur clique sur le cercle rouge.

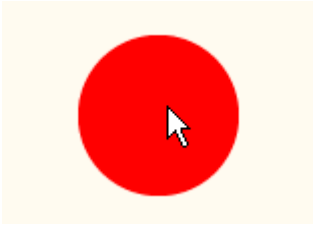

Figure 10-7. Cliquez sur le cercle rouge

Si vous connaissez les événements HTML, vous remarquerez que ce n'est pas très différent en SVG, quoique là nous ayons une collection plus fournie d'événements.

Evénements document

Evénement	Description
`onload`	Quand le SVG est chargé.

Evénements interface

Evénement	Description
`onzoom`	Quand le niveau de zoom sur le document change.

Evénements souris

Evénement	Description
`onclick`	Quand l'utilisateur presse et relâche le bouton.
`onmousedown`	Quand l'utilisateur presse le bouton.
`onmouseup`	Quand l'utilisateur relâche le bouton.
`onmouseover`	Quand le pointeur arrive sur l'objet.
`onmouseout`	Quand le pointeur sort de l'objet.
`onmousemove`	Quand le pointeur se déplace sur l'objet.

Les événements sont importants, mais ne négligeons pas la partie script dans notre document. Voici un exemple de code …

```
<?xml version="1.0"?>
<svg width="600" height="400">
  <script type="text/ecmascript”>
  <![CDATA[
     var rect1 = document.getElementById("greenRect");
     rect1.setAttribute("fill", "red");
    ]]>
    </script>
  <rect id="greenRect" x="100" y="100" width="200" height="60" fill="green" />
   </svg>
```

Nous voudrions changer la couleur de notre élément `greenRect`. Si nous chargeons ce fichier, il ne se passe pas ce que nous espérions. Nous avons commis deux graves erreurs:

1. La variable globale `document` que nous utilisons en HTML n'est pas définie.
2. Même si cette variable était définie, nous ne sommes par certain d'avoir accès aux objets du DOM s'ils ne sont pas chargés

Mais celui-ci fonctionne!

```
<?xml version="1.0"?>
<svg width="600" height="400" onload="Init(evt);">
```

```
   <script type="text/ecmascript">
   <![CDATA[
      function Init(evt)
      {
         var doc = evt.target.ownerDocument;
         var rect1 = doc.getElementById("greenRect");
         rect1.setAttribute("fill", "red");
      }
  ]]></script>
  <rect id="greenRect" x="100" y="100" width="200" height="60" fill="green" />
</svg>
```

Nous corrigeons notre deuxième erreur en définissant une fonction `Init()` qui sera exécutée avec l'événement `onload` au bon moment, quand tous les objets seront chargés et accessibles.

La première erreur peut être corrigée avec l'objet `evt` fourni par le visualiseur. Nous l'utilisons en le passant comme argument à la fonction `Init(evt)`. La variable `evt` de l'objet `Event` doit avoir ce nom exact dans notre code. Avec l'objet `Event` nous avons des propriétés très utiles.

Propriétés de 'Event':

Valeur	Propriété	Description
`String`	`type`	Donne le type de l'événement sous forme de chaîne
`Element`	`currentTarget`	Le récepteur de cet événement.
`Element`	`target`	L'élément qui a reçu l'événement. Un descendant du récepteur.

Méthode de 'Event':

Valeur	Méthode	Description
`void`	`stopPropagation()`	Evite la propagation de l'événement.

Pour l'objet `MouseEvent` nous avons des propriétés supplémentaires.

Propriétés de 'MouseEvent':

Valeur	Propriété	Description
`String`	`type`	Donne le type de l'événement sous forme de chaîne
`Element`	`currentTarget`	Le récepteur de cet événement.
`Element`	`target`	L'élément qui a reçu l'événement. Un descendant du récepteur.
`Number`	`clientX`	Coordonnée en x du pointeur dans la fenêtre.
`Number`	`clientY`	Coordonnée en y du pointeur dans la fenêtre.
`Number`	`button`	Information sur le bouton pressé (0,1,2 for gauche/central/droit).
`Number`	`detail`	Nombre de clicks (avec onclick seulement).

Un objet `Event` n'est pas seulement transmis à un élément, il suit un trajet prédéfini pour s'assurer que le récepteur sera atteint. Un utilisateur clique sur un objet, si l'objet n'est pas déclaré comme récepteur, l'événement `onclick` se propage en remontant l'arborescence ('event bubbling'). Sur son parcours, il peut rencontrer un récepteur qui gérera l'événement.

> *Quand nous avons des éléments sensibles, nous devons prendre en compte le trajet des événements pour le document. La méthode **stopPropagation** peut interrompre cette transmission si nécessaire.*

Nous en savons assez pour jouer avec ces événements.

Eléments sensibles

Les événements liés à la souris ont évidemment une très grande importance.

Figure 10-8. Rectangles sensibles

Nous prenons un exemple simple pour pouvoir nous concentrer sur ces événements.

1. Ce code définit cinq rectangles

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
onload="Init(evt);">
  <script type="text/ecmascript" xlink:href="RemoveWhiteSpace.js"/>
  <script type="text/ecmascript">
  <![CDATA[
    function Init(evt)
    {
       RemoveWhiteSpaceChildNodesOf(evt.target.ownerDocument.documentElement);
    }
  ]]>
  </script>
  <g>
     <rect x="10"  y="100" width="90" height="40" fill="blue" opacity="0.4" />
     <rect x="110" y="100" width="90" height="40" fill="green" opacity="0.4" />
     <rect   x="210"   y="100"   width="90"   height="40"   fill="yellowgreen"
opacity="0.4”/>
     <rect  x="310"  y="100"  width="90"  height="40"  fill="orange"  opacity="0.4"
/>
     <rect x="410" y="100" width="90" height="40" fill="red" opacity="0.4" />
  </g>
  <text x="55"  y="160" text-anchor="middle">highlight</text>
  <text x="155" y="160" text-anchor="middle">magical</text>
  <text x="255" y="160" text-anchor="middle">changing color</text>
  <text x="355" y="160" text-anchor="middle">mutate</text>
  <text x="455" y="160" text-anchor="middle">click</text>
</svg>
```

Tous ces rectangles placés dans un même groupe ont un attribut `opacity` avec une valeur de `0.4`. Nous voulant créer un effet de surbrillance en portant cette valeur à 1.0, quand le pointeur est sur l'un de ces rectangles.

Surbrillance

Nous prenons le rectangle bleu et

1. Faisons de ce rectangle un récepteur en lui attribuant les événements
2. Nous devons définir les fonctions ECMAScript `Highlight` et `Unhighlight`.

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
onload="Init(evt);">
  <script type="text/ecmascript" xlink:href="RemoveWhiteSpace.js"/>
  <script type="text/ecmascript">
  <![CDATA[
    function Init(evt)
    {
       RemoveWhiteSpaceChildNodesOf(evt.target.ownerDocument.documentElement);
    }
    function Highlight(evt)
    {
       evt.target.setAttribute("opacity", "1.0");
    }
    function Unhighlight(evt)
```

```
    {
       evt.target.setAttribute("opacity", "0.4");
    }
  ]]>
  </script>
  <g>
    <rect x="10"  y="100" width="90" height="40" fill="blue" opacity="0.4"
          onmouseover="Highlight(evt);" onmouseout="Unhighlight(evt);"/>
    <rect x="110" y="100" width="90" height="40" fill="green" opacity="0.4" />
    <rect   x="210"   y="100"   width="90"   height="40"   fill="yellowgreen"
opacity="0.4"/>
    <rect x="310" y="100" width="90" height="40" fill="orange" opacity="0.4" />
    <rect x="410" y="100" width="90" height="40" fill="red" opacity="0.4" />
  </g>
  <text x="55"  y="160" text-anchor="middle">highlight</text>
  <text x="155" y="160" text-anchor="middle">magical</text>
  <text x="255" y="160" text-anchor="middle">changing color</text>
  <text x="355" y="160" text-anchor="middle">mutate</text>
  <text x="455" y="160" text-anchor="middle">click</text>
</svg>
```

Nous utilisons la propriété `target` de l'objet `MouseEvent` pour accéder directement à l'élément `<rect>` récepteur de cet événement. Quand le pointeur entre dans le rectangle nous donnons la valeur 1.0 à son attribut `opacity`, et la valeur 0.4 quand le pointeur quitte le rectangle.

Figure 10-9. Surbrillance du rectangle

Si nous voulons donner la même propriété aux autres rectangles, nous devons leur ajouter les propriétés `onmouseover` et `onmouseout` également.

4. Nous avons une solution plus simple. En utilisant la propagation de l'événement – *event bubbling*, nous pouvons simplement affecter les propriétés `onmouseover` et `onmouseout` à l'élément `<g>`.

```
  <g onmouseover="Highlight(evt);" onmouseout="Unhighlight(evt);">
     <rect x="10"  y="100" width="90" height="40" fill="blue" opacity="0.4" />
     <rect x="110" y="100" width="90" height="40" fill="green" opacity="0.4" />
     <rect   x="210"   y="100"   width="90"   height="40"   fill="yellowgreen"
opacity="0.4"/>
     <rect  x="310"  y="100"  width="90"  height="40"  fill="orange"  opacity="0.4"
/>
     <rect x="410" y="100" width="90" height="40" fill="red" opacity="0.4" />
  </g>
```

Ceci fonctionne très bien. Si nous voulions accéder à l'élément `<g>` dans les fonctions, nous utiliserions `evt.currentTarget` à la place de `evt.target`.

Changer la couleur

Nous voudrions que le troisième rectangle devienne noir quand nous cliquons sur lui et reprenne ensuite sa couleur.

5. Nous affectons à l'élément `<rect>` les événements `onmousedown` et `onmouseup`.

```
     <rect x="210" y="100" width="90" height="40" fill="yellowgreen"
opacity="0.4"
           onmousedown="ChangeColor(evt);" onmouseup="ChangeColor(evt);"/>
```

6. Nous devons également définir la fonction associée. Nous utiliserons la même fonction pour les deux événements en nous servant du type de l'événement. Voici le code complet.

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
onload="Init(evt);">
  <script type="text/ecmascript" xlink:href="RemoveWhiteSpace.js"/>
  <script type="text/ecmascript">
  <![CDATA[
    function Init(evt)
    {
       RemoveWhiteSpaceChildNodesOf(evt.target.ownerDocument.documentElement);
    }
    function Highlight(evt)
    {
       evt.target.setAttribute("opacity", "1.0");
    }
    function Unhighlight(evt)
    {
       evt.target.setAttribute("opacity", "0.4");
    }
    function ChangeColor(evt)
    {
       if (evt.type == "mousedown")
          evt.target.setAttribute("fill", "black");
       else if (evt.type == "mouseup")
          evt.target.setAttribute("fill", "yellowgreen");
      }
  ]]>
  </script>
  <g onmouseover="Highlight(evt);" onmouseout="Unhighlight(evt);">
    <rect x="10"  y="100" width="90" height="40" fill="blue" opacity="0.4" />
    <rect x="110" y="100" width="90" height="40" fill="green" opacity="0.4" />
    <rect   x="210"   y="100"   width="90"   height="40"   fill="yellowgreen"
opacity="0.4"
          onmousedown="ChangeColor(evt);" onmouseup="ChangeColor(evt);"/>
    <rect x="310" y="100" width="90" height="40" fill="orange" opacity="0.4" />
    <rect x="410" y="100" width="90" height="40" fill="red" opacity="0.4" />
  </g>
  <text x="55"  y="160" text-anchor="middle">highlight</text>
  <text x="155" y="160" text-anchor="middle">magical</text>
  <text x="255" y="160" text-anchor="middle">changing color</text>
  <text x="355" y="160" text-anchor="middle">mutate</text>
  <text x="455" y="160" text-anchor="middle">click</text>
</svg>
```

Figure 10-10. Rectangle noir quand nous le cliquons

Quand le pointeur est sur ce rectangle, il passe en surbrillance, quand nous pressons le bouton il devient noir et retrouve sa couleur quand nous relâchons le bouton. Il perd sa surbrillance quand le pointeur le quitte.

Cependant, si nous gardons le bouton pressé sur le rectangle, il est noir et en surbrillance, mais si nous relâchons le bouton en dehors du rectangle, il perd sa surbrillance mais reste noir. L'événement `onmouseup` n'a pas été perçu par le rectangle, le pointeur n'étant plus sur lui.

Rendre invisible

Nous voulons faire disparaître le rectangle 'mutate' quand la souris est sur le second rectangle, rectangle magique. Nous en profitons pour supprimer la surbrillance pour lui.

7. Nous lui attribuons les événements

```
<rect x="110" y="100" width="90" height="40" fill="green" opacity="0.4"
      onmouseover="Magical(evt);" onmouseout="Magical(evt);"/>
```

8. La fonction `Magical` très semblable à `ChangeColor`.

```
function Magical(evt)
{
   if (evt.type == "mouseover")
      evt.target.nextSibling.nextSibling.setAttribute("visibility", "hidden");
   else
      evt.target.nextSibling.nextSibling.removeAttribute("visibility");
   evt.stopPropagation ();
}
```

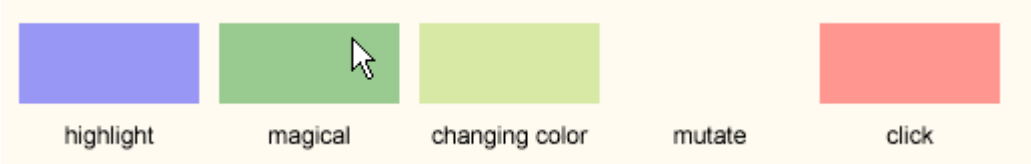

Figure 10-11. Cacher un rectangle

Nous utilisons l'élément `evt.target` pour naviguer dans le DOM tree et changer les attributs d'un autre élément. Nous stoppons la propagation de l'événement avec `evt.stopPropagation()` pour supprimer l'effet de surbrillance.

Transformer un rectangle

Nous voulons qu'en cliquant sur le quatrième rectangle, il devienne une ellipse. Comme nous l'avons vu, nous ne pouvons pas changer le type d'un élément, nous devons le remplacer par un autre élément. Nous avons également vu qu'une solution économique est d'utiliser les propriétés de visibilité. Nous ajoutons un élément `<ellipse>` à notre document et le cachons avec l'attribut `display="none"`.

9. Nous affectons l'événement au rectangle et également à l'ellipse

```
<rect x="310" y="100" width="90" height="40" fill="orange" opacity="0.4"
         onclick="Mutate(evt);" />
<ellipse cx="355" cy="120" rx="45" ry="20" fill="orange" display="none"
         onclick="Mutate(evt);" />
```

10. Nous avons simplement à faire glisser l'attribut `display="none"` de l'élément `<rect>` à l'élément `<ellipse>` pour chaque click.

```
function Mutate(evt)
{
   var elem = evt.target;

   elem.setAttribute("display", "none"); // cache l'élément récepteur ..
   if (elem.nodeName == "rect") // si le récepteur était le rectangle ..
      elem.nextSibling.removeAttribute("display"); // .. montre l'ellipse
   else                         // ellipse was be displayed ..
      elem.previousSibling.removeAttribute("display");  // montre le rectangle
}
```

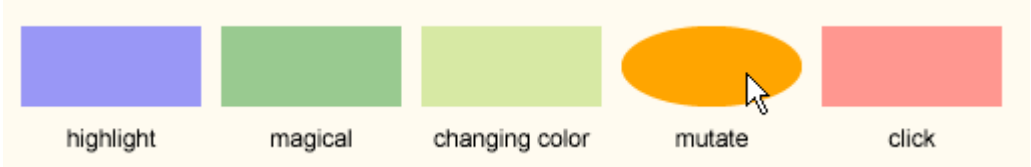

Figure 10-12. Changer la forme

Comme l'élément `<ellipse>` est aussi dans le groupe, la surbrillance fonctionne automatiquement aussi pour cet élément. Vous devez vous demander quelle est la différence entre les attributs `display` et `visibility` attribute. Une différence significative est dans la sensibilité aux événements:

- Les éléments avec `visibility="hidden"` sont invisibles mais peuvent recevoir un événement en leur ajoutant l'attribut `pointer-events="all"`.
- Les éléments avec `display="none"` sont invisibles et ne peuvent recevoir les événements.

C'est cette dernière propriété que nous voulions.

Double Clicks

Nous voulons faire la différence entre un simple click et un double click du bouton du pointeur. Si vous avez lu attentivement les propriétés de `MouseEvent`, vous avez remarqué la propriété `detail`. Elle est très utile et sa valeur dépend du type de l'événement. Avec l'événement `onclick` elle stocke le nombre de clicks que l'utilisateur fait.

11. Nous utilisons le dernier rectangle pour tester ceci.

```
<rect x="410" y="100" width="90" height="40" fill="red" opacity="0.4"
      onclick="ClickCounter(evt);"/>
```

12. Nous créons la fonction `ClickCounter` et affichons le nombre de clicks sous le rectangle.

```
function ClickCounter(evt)
{
var clickText = evt.target.parentNode.parentNode.lastChild.firstChild;
clickText.nodeValue = evt.detail + ". click"
}
```

Figure 10-13. Compteur de clicks

Le premier click écrit ***1. click*** sous le rectangle. Le second click produira des effets différents:

- Si le temps entre les deux clicks est trop long nous aurons toujours ***1. click***.
- Si le temps est assez court mais que le pointeur s'est déplacé, nous aurons encore ***1. click***.
- Si le temps est assez court et que le pointeur n'est pas déplacé nous aurons ***2. click***.

Voici le code complet du document final:

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg"
onload="Init(evt);">
  <script type="text/ecmascript" xlink:href="RemoveWhiteSpace.js"/>
  <script type="text/ecmascript">
  <![CDATA[
    function Init(evt)
    {
       RemoveWhiteSpaceChildNodesOf(evt.target.ownerDocument.documentElement);
    }
```

```
    function Highlight(evt)
    {
       evt.target.setAttribute("opacity", "1.0");
    }

    function Unhighlight(evt)
    {
       evt.target.setAttribute("opacity", "0.4");
    }

    function ChangeColor(evt)
    {
       if (evt.type == "mousedown")
          evt.target.setAttribute("fill", "black");
       else if (evt.type == "mouseup")
          evt.target.setAttribute("fill", "yellowgreen");
    }

    function Magical(evt)
    {
       if (evt.type == "mouseover")
          evt.target.nextSibling.nextSibling.setAttribute("visibility",
"hidden");
       else
          evt.target.nextSibling.nextSibling.removeAttribute("visibility");
       evt.stopPropagation();
    }

    function Mutate(evt)
    {
       var elem = evt.target;

       elem.setAttribute("display", "none");
       if (elem.nodeName == "rect")
          elem.nextSibling.removeAttribute("display");
       else
          elem.previousSibling.removeAttribute("display");
    }

    function ClickCounter(evt)
    {
       var clickText = evt.target.parentNode.parentNode.lastChild.firstChild;
       clickText.nodeValue = evt.detail + ". click"
    }
  ]]></script>
  <g onmouseover="Highlight(evt);" onmouseout="Unhighlight(evt);">
     <rect x="10"  y="100" width="90" height="40" fill="blue" opacity="0.4" />
     <rect x="110" y="100" width="90" height="40" fill="green" opacity="0.4"
           onmouseover="Magical(evt);" onmouseout="Magical(evt);"/>
     <rect x="210" y="100" width="90" height="40" fill="yellowgreen"
opacity="0.4" onmousedown="ChangeColor(evt);" onmouseup="ChangeColor(evt);"/>
     <rect x="310" y="100" width="90" height="40" fill="orange" opacity="0.4"
           onclick="Mutate(evt);" />
     <ellipse cx="355" cy="120" rx="45" ry="20" fill="orange" display="none"
           onclick="Mutate(evt);" />
     <rect x="410" y="100" width="90" height="40" fill="red" opacity="0.4"
           onclick="ClickCounter(evt);"/>
  </g>
  <text x="55"  y="160" text-anchor="middle">highlight</text>
  <text x="155" y="160" text-anchor="middle">magical</text>
  <text x="255" y="160" text-anchor="middle">changing color</text>
  <text x="355" y="160" text-anchor="middle">mutate</text>
  <text x="455" y="160" text-anchor="middle">click</text>
</svg>
```

Déplacer les objets

Le lecteur attentif aura remarqué que nous avons oublié un événement, le très utile **onmousemove**. Avec cet événement, nous atteignons un degré supérieur dans l'interactivité. Nous allons voir comment manipuler la géométrie des objets avec la souris.

L'événement 'mousemove'

Quand l'utilisateur déplace le pointeur sur un objet, et que cet élément est sensible pour l'événement `onmousemove`, nous pouvons utiliser de nouvelles propriétés. Commençons avec une exemple simple.

1. Voici notre document SVG avec ce code

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="400" height="300" xmlns="http://www.w3.org/2000/svg" >
  <text id="coords" x="385" y="20" text-anchor="end">mouse position (?,
?)</text>
  <rect x="100"  y="50" width="250" height="200" stroke="black" fill="moccasin"
/>
  <text x="100" y="45" text-anchor="middle">(100,50)</text>
  <text x="350" y="264" text-anchor="middle">(350,250)</text>
</svg>
```

Nous avons un rectangle coloré en `moccasin` et du texte pour les coordonnées des angles de ce rectangle et pour afficher les coordonnées du pointeur.

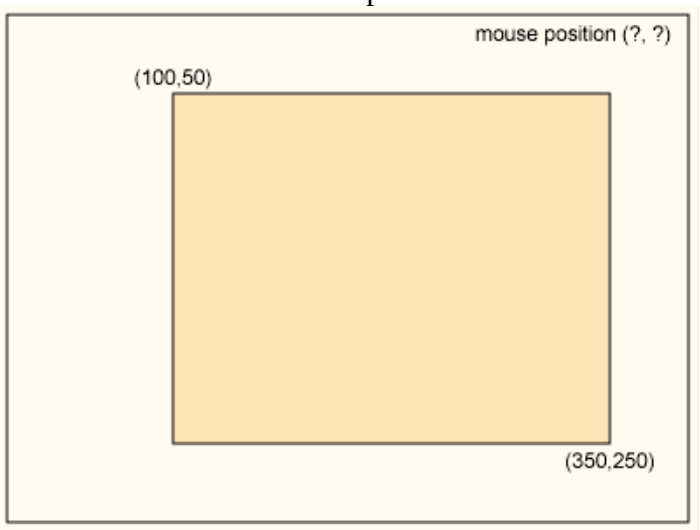

Figure 10-14. Coordonnées du pointeur

2. Nous ajoutons l'attribut `onmousemove` à l'élément `<rect>` et créons une fonction `ShowCoords` qui est appelée par le récepteur.

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="400" height="300" xmlns="http://www.w3.org/2000/svg" >
  <script type="text/ecmascript">
  <![CDATA[
    function ShowCoords(evt)
    {
       var coordText =
evt.target.ownerDocument.getElementById("coords").firstChild;
       coordText.nodeValue = "mouse position (" + evt.clientX + "," + evt.clientY +
")";
    }
  ]]>
  </script>
  <text id="coords" x="385" y="20" text-anchor="end">mouse position (?,
?)</text>
  <rect x="100"  y="50" width="250" height="200" stroke="black" fill="moccasin"
        onmousemove="ShowCoords(evt);"/>
  <text x="100" y="45" text-anchor="middle">(100,50)</text>
  <text x="350" y="264" text-anchor="middle">(350,250)</text>
</svg>
```

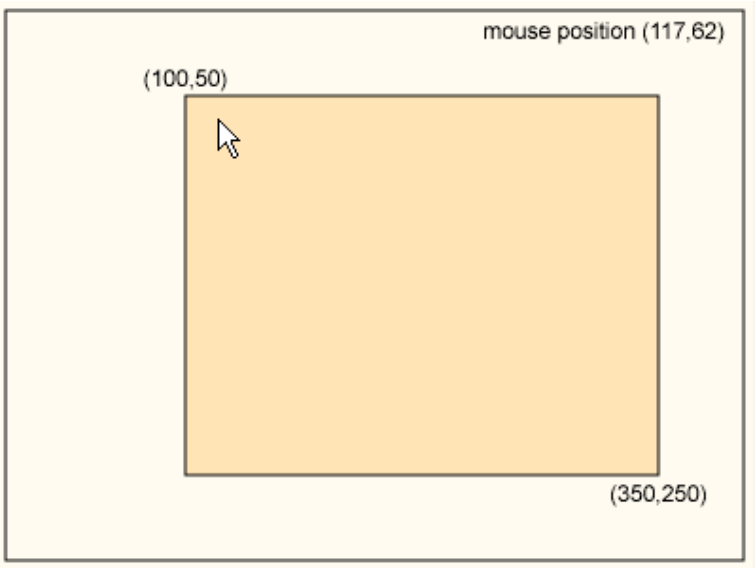

Figure 10-15. Le pointeur et ses coordonnées

Quand nous déplaçons le pointeur sur le rectangle, la chaîne de l'élément `coords` est actualisée en fonction des coordonnées du pointeur. Nous pouvons vérifier ces coordonnées pour les angles du rectangle. Mais ceci ne fonctionne plus quand nous quittons le rectangle `moccasin`.

Pas de problème, nous pouvons utiliser la propagation des événements et placer l'événement `onmousemove` dans le rectangle parent, le document lui-même.

Nous avons ce nouveau code:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="400" height="300" xmlns="http://www.w3.org/2000/svg"
     onmousemove="ShowCoords(evt);">
  <script type="text/ecmascript">
  <![CDATA[
    function ShowCoords(evt)
    {
       var coordText =
evt.target.ownerDocument.getElementById("coords").firstChild;
       coordText.nodeValue = "mouse position (" + evt.clientX + "," +
evt.clientY + ")";     }
  ]]>
  </script>
  <text id="coords" x="385" y="20" text-anchor="end">mouse position (?,
?)</text>
  <rect x="100"  y="50" width="250" height="200" stroke="black"
fill="moccasin"/>
  <text x="100" y="45" text-anchor="middle">(100,50)</text>
  <text x="350" y="264" text-anchor="middle">(350,250)</text>
</svg>
```

Mais celà ne fonctionne pas! Nous sommes un peu dépités. Voyons les spécifications SVG du W3C. Nous lisons:

"*Le récepteur est l'élément de plus haut niveau dont le contenu graphique est sous le pointeur.*"

Ce qui peut s'interpréter "*Pas de remplissage, pas d'événement.*" Nous avons besoin d'un élément avec une région graphique, dessinée ou remplie, sous le pointeur pour recevoir l'événement.

3. Un rectangle couvrant tout notre document doit fait l'affaire.

```
<rect x="5" y="5" width="390" height="290" stroke="black" fill="none" />
```

Cela ne marche toujours pas...
4. Après quelques essais, nous donnons à notre élément `<rect>` l'attribut `fill="white"` et le document fonctionne.

```
<rect x="5" y="5" width="390" height="290" stroke="black" fill="white" />
```

Mais nous avons oublié l'attribut `pointer-events`.
Il peut prendre la valeur `all` et les spécifications nous disent que dans ce cas l'élément est sensible indépendamment des propriétés `fill`, `stroke` et `visibility`.

Mais ceci ne s'applique pas à l'élément <svg>, nous avons besoin du rectangle.

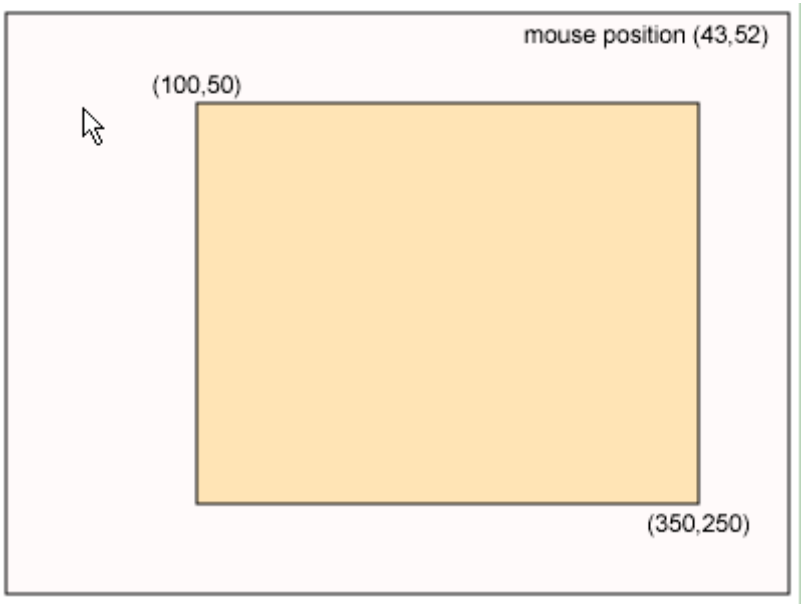

Figure 10-16. Position du pointeur dans le document

Yeux magiques

Construisons un exemple plus spectaculaire. Vous connaissez peut-être *Eyes*, une application graphique familière aux utilisateurs d'Unix. Nous avons deux yeux à l'écran qui suivent en permanence les déplacements du pointeur.

1. Nous allons également dessiner deux rayons allant des yeux au pointeur

```
<?xml version="1.0"?>
<svg width="600" height="300">
  <rect width="600" height="300" stroke="none" fill="white"/>
  <g id="leftEye">
    <circle cx="280" cy="50" r="20" stroke="black" fill="wheat" />
    <line  x1="280"  y1="50"  x2="290"  y2="50"  stroke="lightgray"  stroke-
width="0.5" />
    <circle cx="290" cy="50" r="10" stroke="none" fill="black" />
  </g>
  <g id="rightEye">
    <circle cx="320" cy="50" r="20" stroke="black" fill="wheat" />
    <line  x1="320"  y1="50"  x2="330"  y2="50"  stroke="lightgray"  stroke-
width="0.5" />
```

```
    <circle cx="330" cy="50" r="10" stroke="none" fill="black" />
  </g>
</svg>
```

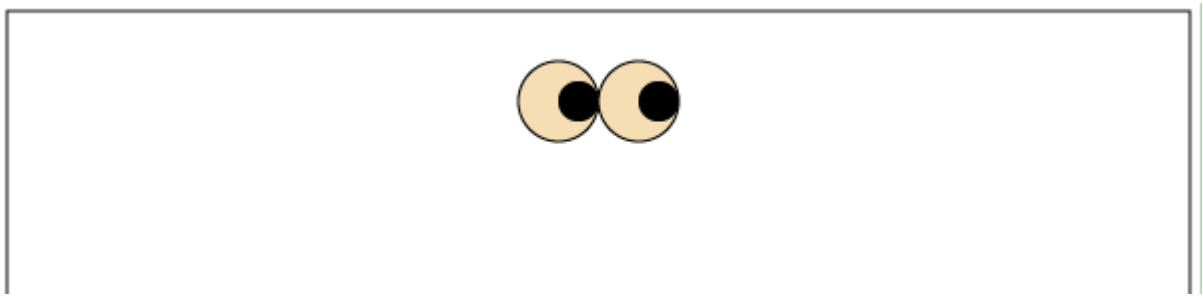

Figure 10-17. Les yeux

2. Nous avons besoin d'un fond, un rectangle. C'est le premier élément. Chaque œil est dans un groupe, il est formé d'un cercle `wheat` et d'un cercle noir pour représenter la pupille. Pour l'instant couvert par la pupille, nous avons également le rayon, une ligne grise.

3. Nous avons besoin de la position du pointeur, nous utilisons `evt.clientX` et `evt.clientY`.

```
function Eyes(evt)
{
   MoveEye(evt.target.ownerDocument.getElementById("leftEye"),
           evt.clientX, evt.clientY);
   MoveEye(evt.target.ownerDocument.getElementById("rightEye"),
           evt.clientX, evt.clientY);
}
```

4. Pour orienter les yeux et placer le rayon.

```
function MoveEye(eyeGroup, x, y)
      {
```

5. Retrouver les éléments de l'oeil.

```
 var pupil = eyeGroup.lastChild,
     ray   = pupil.previousSibling,
```

6. Calculer l'angle de la droite allant du centre de l'oeil au pointeur avec l'axe des x.

   Faire tourner la pupille de cet angle. Nous utilisons `rotate(angle, cx, cy)`.

   Dessiner le rayon.

      Passer à l'autre oeil.

```
function Eyes(evt)
{
   MoveEye(evt.target.ownerDocument.getElementById("leftEye"),
           evt.clientX, evt.clientY);
   MoveEye(evt.target.ownerDocument.getElementById("rightEye"),
           evt.clientX, evt.clientY);
}

function MoveEye(eyeGroup, x, y)
{
   var pupil = eyeGroup.lastChild,
       ray   = pupil.previousSibling,
       xc    = parseFloat(ray.getAttribute("x1")), // x du centre de l'oeil
       yc    = parseFloat(ray.getAttribute("y1")), // y du centre de l'oeil
```

```
       angle = Math.atan2(y-yc, x-xc);             // angle du rayon avec l'axe
x
   pupil.setAttribute("transform",
                      "rotate(" + angle/Math.PI*180 + "," + xc + "," + yc +
")");
   ray.setAttribute("x2", x);                      // fin du rayon ..
   ray.setAttribute("y2", y);                           // .. à la position du
pointeur
}
```

Nous rendons sensible le rectangle de fond avec l'événement `onmousemove`.

```
    <svg width="600" height="300" onmousemove="Eyes(evt);" >
```

Voici nos yeux curieux et inquisiteurs.

Figure 10-18. Une position du pointeur

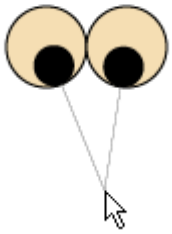

Figure 10-19. Une autre position

Vous pouvez maintenant penser à d'autres applications de cet événement.

Le code complet de l'exemple:

```
<?xml version="1.0"?>
<svg width="600" height="300" onmousemove="Eyes(evt);" >
  <script type="text/ecmascript" xlink:href="RemoveWhiteSpace.js"/>
  <script type="text/ecmascript">
  <![CDATA[
    function Eyes(evt)
    {
       RemoveWhiteSpaceChildNodesOf(evt.target.ownerDocument.documentElement);
       MoveEye(evt.target.ownerDocument.getElementById("leftEye"),
               evt.clientX, evt.clientY);
       MoveEye(evt.target.ownerDocument.getElementById("rightEye"),
               evt.clientX, evt.clientY);
    }

    function MoveEye(eyeGroup, x, y)
    {
       var pupil = eyeGroup.lastChild,
           ray   = pupil.previousSibling,
           xc    = parseFloat(ray.getAttribute("x1")),
           yc    = parseFloat(ray.getAttribute("y1")),
           angle = Math.atan2(y-yc, x-xc);
       pupil.setAttribute("transform", "rotate(" + angle/Math.PI*180 + "," + xc
                                                 + "," + yc + ")");
       ray.setAttribute("x2", x);
       ray.setAttribute("y2", y);
```

```
    }
  ]]>
  </script>
  <rect width="600" height="300" stroke="none" fill="white" />
  <g id="leftEye">
    <circle cx="280" cy="50" r="20" stroke="black" fill="wheat" />
    <line  x1="280"  y1="50"  x2="290"  y2="50"  stroke="lightgrey"  stroke-
width="0.5" />
    <circle cx="290" cy="50" r="10" stroke="none" fill="black" />
  </g>
  <g id="rightEye">
    <circle cx="320" cy="50" r="20" stroke="black" fill="wheat" />
    <line x1="320" y1="50" x2="330" y2="50" stroke="lightgrey" stroke-
width="0.5" />
    <circle cx="330" cy="50" r="10" stroke="none" fill="black" />
  </g>
</svg>
```

Animation par script

Nous avons vu un exemple au précédent chapitre, nous allons revenir en détail sur le contrôle des actions en fonction du temps avec un exemple simple, le déplacement d'un cercle.

Pourquoi ne pas nous contenter d'une simple boucle comme celle-ci:

```
var circle = doc.getElementById(”ball”);
for (var i=1; i<800; i++)
   circle.setAttribute(”cx”, i);
```

La boucle accapare toutes les ressources et nous n'avons plus la main tant qu'elle n'est pas terminée. ECMAScript prévu pour un environnement Web – le navigateur ou l'implémentation SVG – peut arrêter le script après un certain temps pour éviter les boucles sans fin des pages Web ou des documents XML. Une conséquence est que notre solution ne peut être satisfaisante.

Nous devons gérer le temps. ECMAScript ne le permet pas, mais l'objet **`window`** a d'intéressantes méthodes pour le faire. Nous pouvons utiliser ces méthodes dans un script SVG.

Propriété de l'objet 'window':

`String status` le texte de la barre d'état de la fenêtre.

Méthodes de 'window':

`void alert(str)` Affiche une fenêtre d'alerte avec le message `str`.

`void setTimeout(statement,time)` exécute `statement` après une durée de `time`

La méthode `setTimeout` semble efficace pour notre projet. Nous récrivons notre boucle.

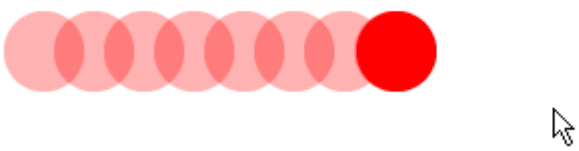

Figure 10-20. Déplacement du cercle rouge

Pour déplacer le cercle vers la droite, nous faisons varier la valeur de l'attribut `cx` du cercle. Voyons le code complet.

```
<?xml version="1.0"?>
<svg width="800" height="400" onload="Init(evt)">
```

```
  <script><![CDATA[
    var ball = null;

    function window.Animate()  // continuously get called after 10 milliseconds
..
    {
      var cx = parseFloat(ball.getAttribute("cx"));   // x du centre du cercle
..
      ball.setAttribute("cx", cx+1);                  // nouvelle valeur pour x
..
      if (cx < 200)
         window.setTimeout("window.Animate()", 10);
    }

    function Init(evt)
    {
      ball = evt.getTarget().getOwnerDocument().getElementById("ball");
      window.Animate();
    }
  ]]></script>
  <circle id="ball" cx="10" cy="200" r="20" fill="red" />
</svg>
```

Ce document SVG a uniquement un élément `circle` avec un identificateur `id="ball"` . Quand le document est chargé complètement, l'événement `onload` du document `<svg>` appelle la fonction `Init`. Cette fonction stocke l'élément `circle` dans une variable globale `ball` qui avait initialement la valeur `null`. Ensuite la fonction `window.Animate()` est appelée.

L'objet `window` a une méthode `Animate()`, car une particularité de l'objet `window` est son extensibilité. Ceci veut dire que nous pouvons lui ajouter des propriétés et des méthodes, ce que nous faisons avec:

```
function window.Animate()
{
   ...
}
```

Pourquoi utiliser une telle syntaxe? Nous allons l'expliquer, mais auparavant voyons cette fonction.

Avec cette fonction, nous stockons l'abscisse du centre du cercle dans la variable `cx`. Nous incrémentons cette valeur d'un pixel et nous l'affectons au cercle avec la méthode `setAttribute`. La dernière partie de la fonction est un appel à la méthode `setTimeout`, cet appel est conditionnel pour nous permettre une sortie de la fonction.

Voyons cet appel en détail. Le premier argument est une chaîne représentant une déclaration ECMAScript, la chaîne `"window.Animate()"`. Le second argument a la valeur `10`, en millisecondes. Nous demandons ainsi à l'objet `window` d'évaluer et d'exécuter cette instruction après un délai de 10 millisecondes. Comme cette instruction est un appel récursif de `window.Animate()`, cette méthode se répète. Comme l'objet `window` n'est pas dans notre document ou dans le DOM, mais dans le navigateur, les fonctions SVG habituelles lui sont invisibles. C'est pourquoi un appel à une fonction comme `Animate()` est impossible par cette méthode `window.setTimeout`. Nous ne devons pas confondre cette technique d'appel avec une fonction récursive, une fonction qui s'appelle elle-même. Pour éviter une répétition à l'infini du mouvement de la balle nous ne passons l'appel que si `cx` est plus petit que 200.

Le délai de 10 millisecondes n'est qu'un souhait. Si les calculs demandent plus de temps ou que le processeur est occupé, nous devrons attendre plus longtemps. La valeur minimale pour cet argument est d'une milliseconde.

Un pendule SVG

Un exemple d'animation plus élaboré.

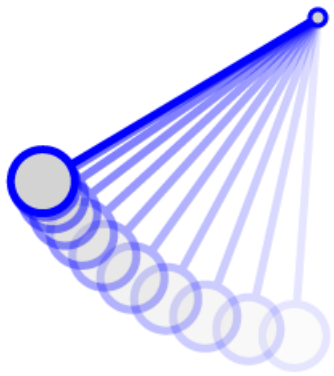

Figure 10-21. Pendule

Pas de panique! Nous avons simplement besoin de la formule du pendule.

$$\theta(t)=\alpha\cos(\sqrt{\frac{g}{l}}t)$$

Dans cette formule

- Théta angle avec la verticale.
- t temps
- g constante de gravité.
- l longueur du pendule
- Alp amplitude des oscillations

Je ne veux pas me focaliser sur la programmation de cette formule qui est semblable à l'exemple précédent, mais voir l'insertion du document SVG dans un document HTML et le contrôle de cette animation depuis le HTML.

> *Jusqu'à présent nous n'avons pas spécifié de configuration pour visualiser nos documents, navigateur et plugin. Mais ici nous devons le faire, la communication entre HTML et SVG 'embedded' n'est possible qu'avec Adobe's SVG Viewer 3.0 et Internet Explorer 5.5 ou+.*

Le code du document SVG en premier.

```
<?xml version="1.0"?>
<svg width="800" height="400" onload="Init(evt)">
  <script type="text/ecmascript">
  <![CDATA[
    var pendulum = null,
        pendulumLen = 200,
        g = 9.81 * 1000,
        amplitude = 60,
        t = 0;
    function window.parent.Animate()
    {
       t = t + 0.01;
       var teta = amplitude*Math.cos(Math.sqrt(g/pendulumLen)*t);
       pendulum.setAttribute("transform", "translate(400,50) rotate(" + teta +
")");
        if (window.parent.active)
           window.setTimeout("window.Animate()", 1);
    }
    function Init(evt)
```

```
    {
       pendulum                                                    =
evt.getTarget().getOwnerDocument().getElementById("pendulum");
       window.parent.active = false;
       window.parent.Animate();
    }
  ]]>
  </script>
  <defs>
    <g id="pendulumGroup">
       <line x1="0" y1="0" x2="0" y2="200" stroke="blue" stroke-width="5"
             stroke-linecap="round" />
       <circle cx="0" cy="200" r="20" stroke="blue" stroke-width="5"
fill="lightgray" />
    </g>
  </defs>
  <rect width="800" height="400" fill="white" />
  <use id="pendulum" xlink:href="#pendulumGroup" transform="translate(400,50)
rotate(0)" />
  <circle cx="400" cy="50" r="5" stroke="blue" stroke-width="3"
fill="lightgray" />
</svg>
```

Pour contrôler le démarrage et l'arrêt des mouvements du pendule nous ajoutons une propriété `active` à l'objet `window.parent`. Comme je vous l'ai dit, `window` est un objet du navigateur et avec `window.parent` nous avons accès à lui côté HTML. Voici le code HTML:

```
<html>
<head>
  <title> SVG pendulum </title>
  </head>
  <body>
    <center>
      <h1>SVG pendulum</h1>
      <embed src="pendulum.svg" height="400" width="800">
      <form>
        <input type=button value="Start" onClick="window.active=true;
window.Animate();">
        <input type=button value="Stop"  onClick="window.active=false;">
      </form>
    </center>
  </body>
</html>
```

Ce n'est pas si compliqué. Nous inserrons notre document SVG avec la balise `<embed>`. Nous créons une forme avec deux boutons `start` et `stop`. Avec l'événement `onclick` sur le bouton 'start', nous déclenchons l'exécution du code ECMAScript:

```
window.active=true; window.Animate();
```

La propriété `active` que nous avons attribuée à l'objet `window` du côté SVG est accessible dans le HTML également. Nous lui donnons la valeur `true` et appellons la méthode `window.Animate()` depuis le HTML. Et cela marche!

Pour arrêter l'animation, il suffit sur l'événement `onclick` d'exécuter ce code

```
window.active=false;
```

Quand la méthode `window.parent.Animate()` est exécutée la fois suivante, l'appel conditionnel

```
if (window.parent.active)
   window.setTimeout("window.parent.Animate()", 1);
```

ne se fait pas et le pendule s'arrête.

Nous pourrions continuer avec d'autres exemples mais nous arrêterons là ce chapitre. Si vous voulez animer quelques éléments SVG, pensez d'abord aux éléments d'animation conformes

aux spécifications SMIL. Dans le chapitre précédent, nous avons vu que leur utilisation permet beaucoup de choses, passez aux scripts si vous ne pouvez réaliser votre projet autrement.

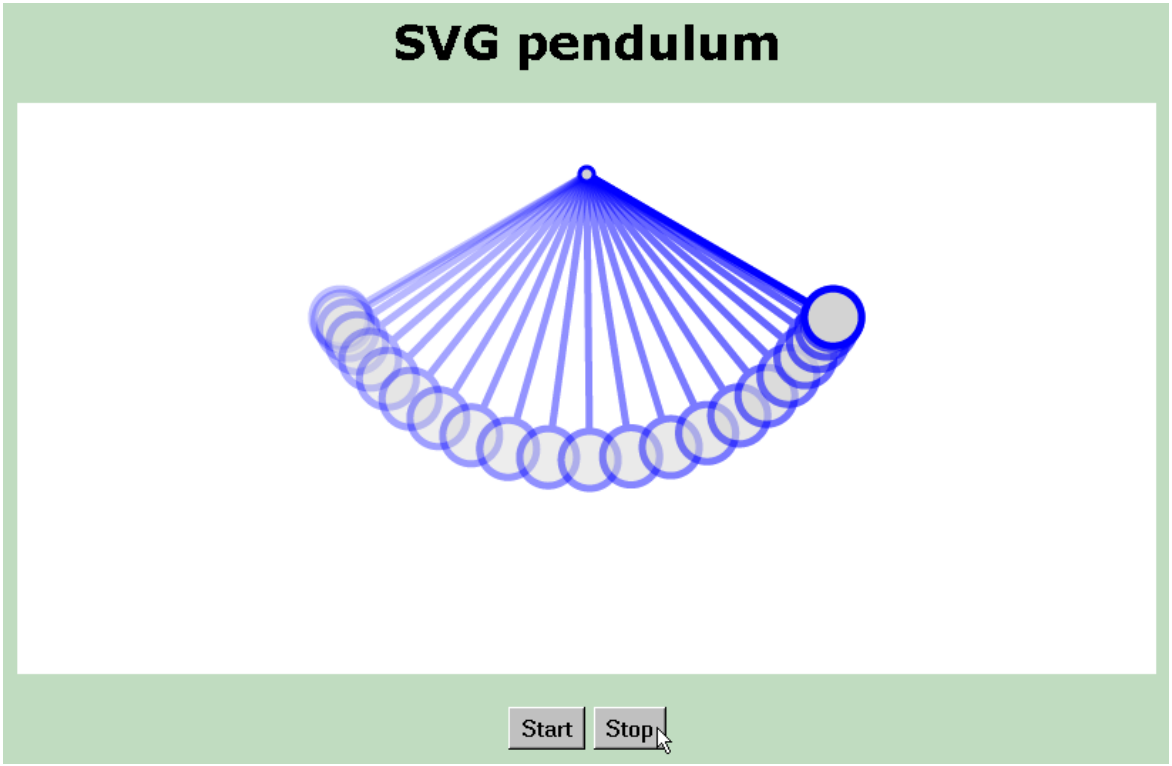

Figure 10-22. Pendule SVG