Introduction à EaselJS

version 080413

David Roche
Lycée G Fichet, Bonneville (74)

À noter :
Tous les exemples ont été testés "en local" avec Firefox (version 18.0.1). Pour les autres navigateurs, des problèmes sont toujours possibles "en local" notamment à cause de la "same origin policy".

- Ce document est destiné à des élèves de lycée qui souhaitent développer des applications (dont des jeux ;-)) en utilisant le couple HTML5+JavaScript
- Vous pouvez directement tester les exemples ici
- Pour plus d'informations, vous pouvez consulter la documentation officielle : ici
- Pour plus d'exemples : ici
- David Rousset a consacré une série d'articles (et une vidéo) sur la construction de A à Z d'un jeu à l'aide d'EaselJS : ici

1ère partie
Les bases

Chapitre I
Pour commencer

EaselJS ? Qu'est ce que c'est ?
EaselJS (site officiel) est une librairie JavaScript développée par Grant Skinner (son blog).
Elle permet une utilisation simplifiée de la balise HTML5 canvas (dessin, création de jeu, manipulation des pixels d'une image....). Pour en savoir plus sur la balise canvas, n'hésitez pas à consulter le site de Mozilla : site Mozilla
Comment utiliser EaselJS ?
EaselJS n'est pas un "logiciel", vous allez donc produire du code (en JavaScript). Votre code va nécessairement être couplé avec une page HTML. Cette page HTML devra "faire" deux choses : définir un canvas et lancer les scripts JavaScript. Pourquoi **les** scripts ? Au minimum nous aurons deux scripts à lancer depuis le HTML : votre propre code bien sûr, mais aussi le code correspondant à EaselJS (il ne faut pas perdre de vu qu'une librairie JavaScript est un programme écrit en JavaScript contenant, entre autres, des méthodes (la notion de méthode est directement liée à la notion d'objet que nous aborderons plus loin dans ce document)).
Voici le code HTML minimum nécessaire à la bonne utilisation d'EaselJS :

code HTML de base :

```
<!DOCTYPE html>
<html lang="fr">
<head>
	<meta charset="utf-8"/>
	<title>Introduction à EaselJS</title>
	<script src="easeljs-0.6.0.min.js"></script>
	<script src="monCodeJS.js "></script>
</head>
<body>
	<canvas id='mon_canvas' width='400px' height='150px' style='border : 1px solid'></canvas>
</body>
</html>
```

Quelques remarques sur ce code :

- Le "doctype" vous montre que nous sommes bien en HTML5.
- Au moment de la rédaction de ce document (début avril 2013) EaselJS est en version 0.6
- Vous trouverez le fichier "EaselJS_v0_6_0.zip" (ou plutôt "EaselJS_vX_X_X.zip" (avec le numéro de version à la place des X)) à cette adresse. Une fois ce fichier "dézippé", il vous faudra récupérer le fichier "**easeljs-X.X.X.min.js**" (il devrait normalement se trouver dans le dossier "lib"). Ce fichier devra être placé dans le même répertoire que le fichier HTML et que votre propre programme JavaScript (**monCodeJS.js**).
- N'oubliez pas de modifier le code HTML ("**easeljs-0.6.0.min.js**" et "**monCodeJS.js**") en fonction de la version d'EaselJS utilisée et du nom de votre fichier JavaScript.
- A chaque fois que vous écrirez un nouveau programme, il vous faudra un nouveau fichier HTML (en remplaçant "**monCodeJS.js**" par le nom de votre fichier JavaScript (par exemple : codeJS_1.js, codeJS_2.js,.....)
- Comme déjà dit plutôt, veillez bien à ce que tous vos fichiers soient dans le même répertoire (bien évidemment il est possible de les mettre dans des répertoires différents à condition de modifier les liens relatifs)
- Nous ajoutons une balise canvas, cette balise canvas est obligatoire pour pouvoir travailler avec EaselJS. Le canvas a une largeur de 400px et une hauteur de 150px (ces valeurs peuvent être modifiées à volonté). Le "**border : 1px solid**" est optionnel, il est juste là pour que le canvas soit visible à l'écran même s'il est vide.

Premier code

Voici le premier fichier JavaScript, ce programme n'affichera rien, mais tous les éléments de base seront en place.

Code 1_1

```
//définition des variables
var monCanvas;
var scene;
function init(){
	//nous récupérons le canvas
	monCanvas = document.getElementById("mon_canvas");
	//nous créons une scène
	scene=new createjs.Stage(monCanvas);
	scene.update();
	}
//on attend que la page soit chargée avant de lancer la fonction init
window.onload=init;
```

Nous définissons 2 variables "`monCanvas`" et "`scene`". Ces 2 variables vont contenir des objets (au sens informatique du terme). Les objets possèdent des méthodes et des attributs.
Pour donner une analogie avec les objets du monde réel, une voiture possède une couleur (c'est un attribut) et la faculté d'accélérer (nous pouvons dire que la voiture possède une méthode "accelerer"). Les méthodes sont des fonctions liées à un objet. Nous aurons de nombreuses fois l'occasion de revenir sur cette notion d'objet. Si vous désirez d'ores et déjà plus d'informations sur la programmation orientée objet, je vous invite à lire le chapitre 8 de ce document.

Nous définissons ensuite la fonction "`init()`" (notez que grâce à la ligne "`window.onload=init;`" cette fonction ne sera appelée qu'au moment où la page HTML sera complètement chargée, il est très important de suivre cette procédure).

La ligne "`monCanvas = document.getElementById("mon_canvas");`" crée un objet de type canvas (en utilisant le canvas défini dans le code HTML : le paramètre de "`getElementById`" doit donc correspondre à l'id de la balise canvas du code HTML. Par la suite nous n'aurons quasiment jamais l'occasion d'utiliser cet objet canvas directement, EaselJS s'en chargera pour nous.

Nous créons ensuite un objet "`scene`" grâce à la ligne "`scene=new createjs.Stage(monCanvas);`" (l'opérateur "`new`" est très souvent utilisé pour créer des objets, la méthode qui suit l'opérateur "`new`" (ici "`Stage`") est appelée constructeur). Cet objet porte bien son nom, car il va contenir tous les "acteurs" (pour l'instant nous n'avons pas "d'acteur" à ajouter à la scène). Notez que "`Stage`" prend l'objet "`monCanvas`" en paramètre (nous plaçons notre "scene" dans le canvas).

Vous avez aussi sans doute remarqué le "`createjs.`" juste devant le "`Stage(monCanvas)`". Il faudra systématiquement placer "`createjs.`" devant les éléments propres à EaselJS (pour ceux qui connaissent, cela a un rapport avec la gestion du "namespace" d'EaselJS).

L'objet "`scene`" est mis à jour grâce à sa méthode "`update()`". Vous aurez très souvent l'occasion de rencontrer ce que l'on appelle la "notation pointée" : "`scene.update();`" qui veut tous simplement dire : "applique la méthode "`update()`" à l'objet "`scene`" (pour notre voiture nous pourrions donc avoir voiture.accelerer()). Cette mise à jour n'a pour l'instant aucun intérêt, la "scene" n'ayant pas été modifié depuis sa création.

Chapitre 2
Afficher une image

Commençons simple, affichons une image :

Code 1_2

```
var monCanvas;
var scene;
var monImage;
var monBitmap;
function init(){
	monCanvas = document.getElementById("mon_canvas");
	scene=new createjs.Stage(monCanvas);
	//Nous définissons un objet image
	monImage=new Image();
	//Un objet de type Image possède un attribut "src" qui doit être égal au chemin de l'image
	monImage.src="smiley.png";
	//L'objet image n'est pas facilement exploitable, nous créons donc un objet de type Bitmap
	//Le paramètre doit être un objet de type image
	monBitmap=new createjs.Bitmap(monImage);
	scene.addChild(monBitmap);
	scene.update();
	}
window.onload=init;
```

Nous commençons par définir un objet de type image "`monImage=new Image();`".
Nous définissons la variable "monImage;" en dehors de la fonction init, ce qui en fait une variable globale (variable utilisable partout dans le code et pas seulement dans la fonction où elle a été définie).
Si la variable "monImage" avait été défini dans la fonction init, elle aurait été utilisable uniquement dans la fonction init (variable locale). Pour le moment cela n'a pas grande importance, mais cela en aura par la suite.

Nous renseignons ensuite l'attribut "`src`" de l'objet "`monImage`" avec le chemin de l'image à afficher (si vos images se trouvent dans un autre répertoire (ce qui est d'ailleurs plutôt conseillé), vous devrez veiller à adapter cette ligne) : "`monImage.src="smiley.png";`".
L'objet de type image, "monImage" n'est pas directement manipulable avec EaselJS, nous devons donc créer un autre objet de type "Bitmap" : "`monBitmap=new createjs.Bitmap(monImage);`". Vous aurez remarqué par vous même que le constructeur "`Bitmap`" prend pour paramètre un objet de type image.
Il suffit ensuite d'ajouter l'objet nouvellement créé à la scène ("`scene.addChild(monBitmap);`") et de mettre à jour la scène ("`scene.update();`").

Et, cela ne fonctionne pas ! Pas d'image à l'écran !

Pour voir apparaitre le smiley à l'écran, il est nécessaire de rafraichir la page. Pourquoi ?

Le chargement de l'image "`monImage.src="smiley.png";`" peut prendre un certains temps, or, ici, le chargement de l'image et la création de l'objet de type Bitmap (qui utilise l'image) s'enchainent. Il est donc tout à fait possible que l'objet de type Bitmap soit créé avant que l'image soit chargée, d'où, le problème rencontré : nous créons un objet de type Bitmap vide, donc pas d'image affichée.
Pour résoudre ce problème, il suffit d'attendre que l'image soit totalement chargée avant de créer l'objet de type Bitmap. Comment ?
En utilisant l'événement "onload" sur l'objet, image et en créant une nouvelle fonction qui sera chargée de créer l'objet de type Bitmap.

Code 2_2

```
var monCanvas;
var scene;
var monImage;
var monBitmap;
function init(){
	monCanvas = document.getElementById("mon_canvas");
```

```
        scene=new createjs.Stage(monCanvas);
        monImage=new Image();
        monImage.src="smiley.png";
        //nous devons attendre que l'image soit chargée avant de créer l'objet de type Bitmap
        //une fois l'image chargée (onload), la fonction creationBitmap est appellée.
        monImage.onload=creationBitmap;
}
//fonction qui permet la création de l'objet de type Bitmap
function creationBitmap(){
        monBitmap=new createjs.Bitmap(monImage);
        scene.addChild(monBitmap);
        scene.update();
}
window.onload=init;
```

Tout devrait maintenant fonctionner du 1er coup (sans être obligé de recharger la page).
Par défaut notre image se trouve dans le coin haut-gauche du canvas (coordonnée (0,0) du canvas).
Il est possible de lui attribuer une nouvelle position à l'aide des attributs x et y de notre objet de type Bitmap (monBitmap.x et monBitmap.y).

Code 3_2

```
var monCanvas;
var scene;
var monImage;
var monBitmap;
function init(){
        monCanvas = document.getElementById("mon_canvas");
        scene=new createjs.Stage(monCanvas);
        monImage=new Image();
        monImage.src="smiley.png";
        monImage.onload=creationBitmap;
}
function creationBitmap(){
        monBitmap=new createjs.Bitmap(monImage);
        //nous utilisons les attributs x et y de l'objet monBitmap pour déplacer le smiley
        monBitmap.x=50;
        monBitmap.y=70;
        scene.addChild(monBitmap);
        scene.update();
}
window.onload=init;
```

Attention, même si cela n'est pas évident à voir, ce n'est pas le centre de l'image qui se trouve à la coordonnée (50,70), mais le coin haut-gauche de l'image.

À faire vous même
Modifier les valeurs des attributs x et y de l'objet monBitmap, vérifier que le smiley se déplace à chaque fois.

Il existe un autre attribut de l'objet monBitmap qui peut être utile : rotation.
Comme son nom l'indique, cet attribut permet d'effectuer une rotation d'un objet de type Bitmap

Code 4_2

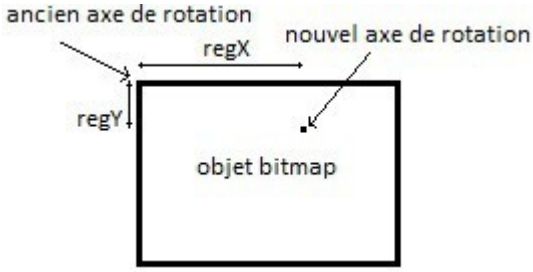
```
var monCanvas;
var scene;
var monImage;
var monBitmap;
function init(){
        monCanvas = document.getElementById("mon_canvas");
        scene=new createjs.Stage(monCanvas);
        monImage=new Image();
        monImage.src="smiley.png";
        monImage.onload=creationBitmap;
}
function creationBitmap(){
        monBitmap=new createjs.Bitmap(monImage);
        monBitmap.x=50;
        monBitmap.y=70;
```

```
        //monBitmap effectue une rotation de 30°
        monBitmap.rotation=30;
        scene.addChild(monBitmap);
        scene.update();
}
window.onload=init;
```

Avant la rotation

Après la rotation

Même si ici aussi ce n'est pas évident à voir, l'axe de rotation n'est pas au milieu de l'image, mais dans le coin haut-gauche de l'image.
Il est possible de déplacer l'axe de rotation grâce aux attributs regX et regY d'un objet de type Bitmap.



Si vous voulez avoir l'axe de rotation au centre de l'image, il suffit donc, sachant que l'image est un carré de 40 pixels de côté (en fait de 39 pixels de côté !), d'avoir regX=20 et regY=20 :

Code 5_2

```
var monCanvas;
var scene;
var monImage;
var monBitmap;
function init(){
        monCanvas = document.getElementById("mon_canvas");
        scene=new createjs.Stage(monCanvas);
        monImage=new Image();
        monImage.src="smiley.png";
        monImage.onload=creationBitmap;
}
function creationBitmap(){
        monBitmap=new createjs.Bitmap(monImage);
        monBitmap.x=50;
```

```
        monBitmap.y=70;
        //nous deplaçons l'axe de rotation au centre de l'image
        monBitmap.regX=20;
        monBitmap.regY=20;
        monBitmap.rotation=30;
        scene.addChild(monBitmap);
        scene.update();
}
window.onload=init;
```

Conséquence de l'utilisation de regX et regY : c'est maintenant le centre de l'image qui se trouve aux coordonnées (50,70).

À faire vous même
Modifier les différents attributs (x, y, regX, regY et rotation) puis constater les changements

À faire vous même
Modifier le code pour afficher 5 smileys à l'écran. Chaque smiley devra avoir une position et un angle de rotation aléatoire. Vous devrez obligatoirement utiliser une boucle for. Pas question de créer les smileys "à la main".
Pour vous aider : la méthode "Math.random()" renvoie un nombre aléatoire compris entre 0 et 1 (comment obtenir un nombre aléatoire compris entre 0 et 150 ?). La méthode "Math.floor()" retourne le plus grand entier inférieur ou égal à la valeur donnée en paramètre, exemple, "Math.floor(2.9)" renvoie 2.

Chapitre 3
Animation

Pour l'instant tout cela est un peu "statique". Dans ce chapitre nous allons donc apprendre à animer notre scène.
L'idée de base est très simple : créer une fonction qui sera appelée un certain nombre de fois par seconde. Tous les éléments qui contrôleront l'animation devront se trouver dans cette fonction.

Code 1_3

```
var monCanvas;
var scene;
var monImage;
var monBitmap;
function init(){
	monCanvas = document.getElementById("mon_canvas");
	scene=new createjs.Stage(monCanvas);
	monImage=new Image();
	monImage.src="smiley.png";
	monImage.onload=creationBitmap;
	//utilisations de l'objet EaselJS Ticker
	createjs.Ticker.useRAF=true;
	createjs.Ticker.setFPS(40);
	createjs.Ticker.addListener(window);
}
function creationBitmap(){
	monBitmap=new createjs.Bitmap(monImage);
	monBitmap.x=20;
	monBitmap.y=70;
	monBitmap.regX=20;
	monBitmap.regY=20;
	scene.addChild(monBitmap);
}
function tick(){
	monBitmap.x=monBitmap.x+1;
	scene.update();
}
window.onload=init;
```

Tout ce qui est animation est géré dans EaselJS par l'objet Ticker. Il est inutile d'utiliser l'opérateur "`new`" car l'objet Ticker est un objet un peu particulier, c'est un objet dit static (pour ceux qui connaissent déjà un peu la programmation orientée objet, l'objet Ticker ne doit pas être instancié, il est utilisé tel quel).

Détaillons ligne après ligne l'utilisation de l'objet Ticker :

- "`createjs.Ticker.useRAF=true;`" l'objet Ticker peut utiliser la méthode JavaScript requestAnimationFrame (d'où le RAF) qui améliore grandement la gestion des animations par rapport aux anciennes méthodes (setTimeout ou setInterval). Si l'attribut de l'objet Ticker "`useRAF`" est égal à "**true**" alors EaselJS utilisera la méthode requestAnimationFrame pour gérer les animations. Pour plus d'information sur ce sujet, vous pouvez consulter le site suivant : http://paulirish.com/2011/requestanimationframe-for-smart-animating/ (en anglais).
- "createjs.Ticker.setFPS(40);" permet de définir le nombre d'images par seconde (ici on aura 40 images par seconde)
- "`createjs.Ticker.addListener(window);`" cette ligne permettra d'appeler la fonction "`tick()`" 40 fois par seconde.

Comme dit ci-dessus la fonction "`tick()`" sera appelée 40 fois par seconde (si la machine le permet). Dans notre exemple la coordonnée x de l'objet "`monBitmap`" augmentera donc d'une unité tout le 1/40 de seconde ("`monBitmap.x=monBitmap.x+1;`"). L'affichage de la scène sera donc aussi mis à jour 40 fois par seconde ("scene.update();").

Pour ceux qui ont déjà eu l'occasion d'utiliser ce genre de fonction, vous remarquerez que la fonction tick() n'a pas à gérer l'effacement du canvas, l'objet Ticker se charge de tout. C'est un des nombreux avantages qu'apporte l'utilisation d'EaselJS.

À faire vous même
Modifier le programme pour que le smiley donne l'impression de rouler

À faire vous même
Modifier le programme pour que le smiley ne disparaisse pas quand il arrive au bord du canvas. Il doit "rebondir" sur les bords.

À faire vous même
Le smiley doit maintenant se déplacer en "diagonale" (modification de x et de y). Il doit toujours "rebondir" sur le bord du canvas (de façon réaliste).

Chapitre 4
Gestion du clavier

Le but de ce chapitre est de permettre à l'utilisateur d'interagir avec le programme par l'intermédiaire du clavier.
L'idée est de créer un objet JavaScript (plus précisément un objet littéral).
Cet objet se nommera "clavier" : "`var clavier={gauche:0, droite:0};`". Cet objet a 2 attributs "droite" et "gauche".

La méthode "`window.onkeydown`" permet de "surveiller" le clavier. Dès que l'utilisateur enfonce une touche, la fonction "`keyPress`" est exécutée.
"`e.keyCode`" est égal au code de la touche enfoncée par l'utilisateur (voir ce site pour les codes des touches : ici).
Si cette touche correspond à la "flèche gauche" (code clavier = 37) l'attribut "`gauche`" de l'objet "`clavier`" prend pour valeur 1.
Le raisonnement est identique pour "la flèche droite".
Dès que l'utilisateur relâche une touche, la fonction "`keyRelease`" est appelée. Si la touche relâchée correspond à la "flèche gauche", l'attribut "`gauche`" de l'objet "`clavier`" prend pour valeur 0.

Code 1_4

```
var monCanvas;
var scene;
var monImage;
var monBitmap;
//*****gestion du clavier : début***********
//Nous créons un objet "clavier" qui comporte 2 attributs : "droite" et "gauche"
var clavier={gauche:0, droite:0};
//quand l'utilisateur enfonce une touche, la fonction keyPress est appelée
window.onkeydown=keyPress;
//quand l'utilisateur relâche une touche, la fonction keyRelease est appelée
window.onkeyup=keyRelease;
//le paramètre "e" correspond à un événement
function keyPress(e){
	//la touche "flèche gauche" porte le numéro 37
	if (e.keyCode == 37){
	//quand l'utilisateur enfonce la touche "flèche gauche", l'attribut "gauche" a pour valeur 1
		clavier.gauche = 1;
	}
	//la touche "flèche droite" porte le numéro 39
	if (e.keyCode == 39){
	//quand l'utilisateur enfonce la touche "flèche droite", l'attribut "droite" a pour valeur 1
		clavier.droite = 1;
	}
}
function keyRelease(e){
	if (e.keyCode == 37){
	//quand l'utilisateur relâche la touche "flèche gauche", l'attribut "gauche" a pour valeur 0
		clavier.gauche = 0;
	}
	if (e.keyCode == 39){
	//quand l'utilisateur relâche la touche "flèche droite", l'attribut "droite" a pour valeur 0
		clavier.droite = 0;
	}
}
//*****gestion du clavier : fin***********
function init(){
	monCanvas = document.getElementById("mon_canvas");
	scene=new createjs.Stage(monCanvas);
	monImage=new Image();
	monImage.src="smiley.png";
	monImage.onload=creationBitmap;
	createjs.Ticker.useRAF=true;
	createjs.Ticker.setFPS(40);
	createjs.Ticker.addListener(window);
}
function creationBitmap(){
	monBitmap=new createjs.Bitmap(monImage);
	monBitmap.x=200;
	monBitmap.y=70;
	monBitmap.regX=20;
	monBitmap.regY=20;
	scene.addChild(monBitmap);
}
```

```
//la fonction deplacement gère le déplacement du smiley
function deplacement(){
	if (clavier.gauche==1){
		//si clavier.gauche=1 alors on décrémente la position x du smiley
		monBitmap.x=monBitmap.x-2;
	}
	if (clavier.droite==1){
		//si clavier.droite=1 alors on incrémente la position x du smiley
		monBitmap.x=monBitmap.x+2;
	}
}
function tick(){
	//on appelle la fonction deplacement à chaque image
	deplacement();
	scene.update();
}
window.onload=init;
```

Vous remarquerez que la fonction "`tick`" permet d'appeler la fonction "`deplacement`" à chaque image.
Cette fonction "`deplacement`" teste l'attribut "`gauche`" de l'objet clavier.
Si "`clavier.gauche`" est égal à 1(touche enfoncée), la valeur de `"monBitmap.x"` diminue de 2 unités (déplacement de 2 pixels vers la gauche du centre du smiley à chaque image).
Le raisonnement est identique pour la flèche droite.

À faire vous même
Faire les modifications nécessaires pour permettre à l'utilisateur de déplacer le smiley dans les 4 directions (gauche, droite, haut et bas).

Mini-Projet
Il y aura à l'écran 2 smileys : un contrôlé par le joueur (touches clavier) et un qui décrira une trajectoire rectiligne uniforme . Le joueur devra "attraper" le plus rapidement possible le 2e smiley. Pour vous aider : on pourra considérer que le 2e smiley sera "capturé" si la distance entre les centres des objets de type Bitmap est inférieure à une certaine valeur. Si le centre de l'objet1 a pour coordonnées x1,y1 et que l'objet2 a pour coordonnées x2,y2, d'après le théorème de Pythagore , la distance entre les centres d est égale à $\sqrt{(x2-x1)^2+(y2-y1)^2}$

Chapitre 5
Gestion de la souris

Après le clavier, occupons nous de la souris. Ici aussi nous avons des méthodes qui vont "surveiller" la souris.

Code 1_5

```
var monCanvas;
var scene;
var monImage;
var monBitmap;
function init(){
	monCanvas = document.getElementById("mon_canvas");
	scene=new createjs.Stage(monCanvas);
	monImage=new Image();
	monImage.src="smiley.png";
	monImage.onload=creationBitmap;
}
function creationBitmap(){
	monBitmap=new createjs.Bitmap(monImage);
	monBitmap.x=200;
	monBitmap.y=70;
	monBitmap.regX=20;
	monBitmap.regY=20;
	monBitmap.onClick=clickSouris;
	scene.addChild(monBitmap);
	scene.update();
}
function clickSouris(){
	alert('ça marche ;-)');
}
window.onload=init;
```

Remarquez que la méthode "`onClick`" s'applique ici à l'objet "`monBitmap`". En cas de clic sur l'objet "`monBitmap`" la fonction "`clickSouris`" est exécutée.

À faire vous même
Il existe d'autres méthodes qui permettent de gérer la souris : onDoubleClick, onMouseOut, onMouseOver .
*Testez ces autres méthodes. Attention pour "*onMouseOut*" et "*onMouseOver*" il faudra ajouter la ligne suivante : "scene.enableMouseOver();" à votre programme.*

À faire vous même
Après chaque click le smiley change de position (nouvelle position aléatoire).

Il est possible grâce à la souris de déplacer le smiley dans le canvas.
Pour ce faire nous allons combiner 2 événements : "`onPress`" et "`onMouseMove`". Nous allons aussi utiliser la notion de fonction anonyme (fonction qui n'a pas de nom !) :
"`e.onMouseMove=function(ev){....`" quand la souris bouge, la fonction qui suit le signe égal est exécutée.

Code2_5

```
var monCanvas;
var scene;
var monImage;
var monBitmap;
function init(){
	monCanvas = document.getElementById("mon_canvas");
	scene=new createjs.Stage(monCanvas);
	monImage=new Image();
	monImage.src="smiley.png";
	monImage.onload=creationBitmap;
	scene.enableMouseOver();
	createjs.Ticker.useRAF=true;
	createjs.Ticker.setFPS(40);
	createjs.Ticker.addListener(window);
}
function creationBitmap(){
```

```
        monBitmap=new createjs.Bitmap(monImage);
        monBitmap.x=200;
        monBitmap.y=70;
        monBitmap.regX=20;
        monBitmap.regY=20;
        //si l'utilisateur appuie sur le bouton de la souris
        //et que l'objet sous le curseur est le smiley
        //alors la fonction PressSouris est appelée
        monBitmap.onPress=PressSouris;
        scene.addChild(monBitmap);
}
//le paramètre de la fonction PressSouris "e" correspond à l'événement "appuie sur le bouton"
function PressSouris(e){
        //nous utilisons ici une fonction dite "anonyme" (sans nom)
        //quand la souris se déplace (et que l'événement "appuie sur le bouton" est toujours en
cours)...
        //....la fonction qui suit le signe égal est exécutée
        e.onMouseMove=function(ev){
                //e.target.x = coordonnée x de l'objet qui se trouvait sous le curseur de la
souris...
                //....au moment de l'événement (c'est à dire ici l'événement "appuie sur le bouton")
                //"ev.stageX" correspond à la coordonnée X de la souris tout au long du déplacement
de cette dernière
                //"ev" est aussi un événement=>l'événement "la souris bouge"
                e.target.x=ev.stageX;
                e.target.y=ev.stageY;
        }
}
function tick(){
        scene.update();
}
window.onload=init;
```

Le principe est relativement complexe :
L'utilisateur clique (sans relâcher le bouton) sur le smiley : la fonction "`PressSouris`" est exécutée. Si maintenant l'utilisateur bouge la souris (tout en maintenant le bouton enfoncé), la fonction anonyme est exécutée : les coordonnées du smiley ("`e.target.x`" et "`e.target.y`") sont modifiées ("`e.target`" correspond au smiley car c'est un clic sur le smiley qui a déclenché tout le processus).
"`ev.stageY`" est égal à tout instant, à la position du curseur de la souris. Conséquence : tant que la bouton de la souris reste enfoncé, le smiley suit le curseur de la souris.

À faire vous même
Nous voulons avoir 5 smileys à l'écran, tous ces smileys devront être "déplaçables" grâce à la souris.

Chapitre 6
Gestion des sprites

Beaucoup de jeux utilisent les sprites pour créer les animations des personnages. Mais qu'est-ce qu'un sprite ?

D'après wikipedia :
"Un ***sprite*** (ou *lutin*) est dans le jeu vidéo un élément graphique qui peut se déplacer sur l'écran. En principe, un sprite est en partie transparent, et il peut être animé (c'est-à-dire qu'il est formé de plusieurs bitmaps qui s'affichent les uns après les autres)."

C'est la dernière parenthèse de cette définition qui va retenir particulièrement notre attention.

Comment animer un sprite avec EaselJS ?

Avec ou sans EaselJS, la technique d'animation est toujours la même : nous allons utiliser le principe du dessin animé, faire défiler des images fixes, ce qui donnera la sensation de mouvement. Voici les séries d'images fixes que nous allons utiliser dans ce chapitre.

Comme vous pouvez le constater, cette image au format png va nous permettre de gérer plusieurs animations. Par exemple si nous prenons les 6 premières représentations de Sonic (en partant en haut à gauche), il est évident que leur enchainement rapide à l'écran donnera l'impression que Sonic est en train de marcher.

Sans EaselJS, même si le principe n'est pas très complexe, le code à écrire pour mettre en place cette animation sera relativement long. Avec EaselJS cela sera un jeu d'enfant.

Pour commencer nous allons (par la pensée) découper notre image .png (que nous appellerons désormais "spritesheet") en petites vignettes de 48 pixels de large et 48 pixels de haut. À l'aide d'un logiciel de dessin standard, vérifiez que chaque petite vignette contient bien un dessin de Sonic. Ces vignettes vont être ensuite numérotées en partant d'en haut à gauche. Par exemple la première vignette de la deuxième ligne portera le numéro 11 (la première vignette de la première ligne porte le numéro zéro).
Au niveau de EaselJS il y a plusieurs étapes à bien respecter pour créer un sprite animé :

1) Création d'un objet littéral qui va contenir toutes les informations nécessaires à la mise en place de l'animation. Nous l'appellerons "data_ss" :

```
data_ss = {
//image utilisée pour créer le sprite sheet
images:[monImage],
//taille des images dans le sprite sheet
```

```
frames:{width:48, height:48, regX: 24, regY:24},
//définition des animations
animations:{marche:[0, 5,true,7]}
}
```

images : doit correspondre au "spritesheet" (image .png)
frames : donne des indications sur les vignettes (regX et regY permettent de préciser le centre de la vignette)
animations : {nom_animation:[vignette début, vignette fin, animation en boucle, fréquence]}
Attention : plus la fréquence est élevée, plus l'animation est lente

2) Création d'un objet de type "`SpriteSheet`" :
"`var ss_sonic = new createjs.SpriteSheet(data_ss);`". Le paramètre doit correspondre à l'objet littéral défini en 1)
3) Création d'un objet de type :
"`BitmapAnimation`" : "`sonic = new createjs.BitmapAnimation(ss_sonic);`".
Le paramètre doit correspondre à l'objet de type "`SpriteSheet`" défini en 2)
Un objet de type "`BitmapAnimation`" se manipule comme un objet de type "`Bitmap`".

Code 1_6

```
var monCanvas;
var scene;
var monImage;
var sonic;
function init(){
	monCanvas = document.getElementById("mon_canvas");
	scene=new createjs.Stage(monCanvas);
	monImage=new Image();
	monImage.src="sonicsprites.png";
	monImage.onload=creationPerso;
	createjs.Ticker.useRAF=true;
	createjs.Ticker.setFPS(40);
	createjs.Ticker.addListener(window);
}
function creationPerso(){
	//Nous définissons un objet littéral data_ss qui contiendra toutes les données....
	//....nécessaires à la construction d'un objet SpriteSheet
	data_ss = {
				//image utilisée pour créer le sprite sheet
				images:[monImage],
				//taille des images dans le sprite sheet
				frames:{width:48, height:48, regX: 24, regY:24},
				//définition des animations
				animations:{marche:[0, 5,true,2]}
			}
	//ss_sonic est un objet de type SpriteSheet, on passe notre objet data_ss en argument
	var ss_sonic = new createjs.SpriteSheet(data_ss);
	//nous créons un objet de type BitmapAnimation : sonic
	sonic = new createjs.BitmapAnimation(ss_sonic);
	sonic.x=200;
	sonic.y=100;
	scene.addChild(sonic);
	animation();
}
function animation(){
	//nous lançons l'animation "marche"
	sonic.gotoAndPlay("marche");
}

function tick(){
	scene.update();
}
window.onload=init;
```

L'animation est lancée grâce à la méthode "`gotoAndPlay`" qui prend en paramètre le nom de l'animation défini dans l'objet "`data_ss`".

À faire vous même
Modifier la valeur de la fréquence dans l'objet data_ss, vérifiez le résultat.

Il est tout à fait possible de définir plusieurs animations dans l'objet "`data_ss`" :
"`animations:{marche:[0, 5,true,7], course:[11,14,true,4],tourne:[44,48,true,12]}`"

Ensuite pour lancer l'animation "`course`" par exemple : "`sonic.gotoAndPlay("course");`"

À faire vous même
Mettre en place plusieurs animations. L'utilisateur pourra changer d'animation, par exemple, en appuyant sur des touches du clavier.

Et si maintenant nous voulons faire courir Sonic de la droite vers la gauche ?
Visiblement nous n'avons pas les sprites adaptés !
Ce n'est pas grave, EaselJS nous fournit une solution avec "`SpriteSheetUtils`" (objet static, comme "`Ticker`").
EaselJS va automatiquement créer une animation "miroir" (Sonic va courir de la droite vers la gauche) en utilisant la méthode "`addFlippedFrames`" de l'objet "`SpriteSheetUtils`".

Code 2_6

```
var monCanvas;
var scene;
var monImage;
var sonic;
function init(){
	monCanvas = document.getElementById("mon_canvas");
	scene=new createjs.Stage(monCanvas);
	monImage=new Image();
	monImage.src="sonicsprites.png";
	monImage.onload=creationPerso;
	createjs.Ticker.useRAF=true;
	createjs.Ticker.setFPS(40);
	createjs.Ticker.addListener(window);
}
function creationPerso(){
	data_ss = {
				images:[monImage],
				frames:{width:48, height:48, regX: 24, regY:24},
				animations:{marche:[0, 5,true,2]}
	}
	var ss_sonic = new createjs.SpriteSheet(data_ss);
	//addFlippedFrames est une méthode qui permet de créer des animations "miroirs" :...
	//...marche_h : animation "marche vers la gauche"
	createjs.SpriteSheetUtils.addFlippedFrames(ss_sonic, true,false,false);
	sonic = new createjs.BitmapAnimation(ss_sonic);
	sonic.x=200;
	sonic.y=100;
	scene.addChild(sonic);
	animation();
}
function animation(){
	sonic.gotoAndPlay("marche_h");
}

function tick(){
	scene.update();
}
window.onload=init;
```

Comme vous pouvez le constater, la nouvelle animation se nomme automatiquement "`marche_h`" (on ajoute un _h).
Attention : la méthode "`addFlippedFrames`" est "en fin de vie" (on dit qu'elle est dépréciée). Elle est en cours de remplacement par la méthode "`flip`" qui est elle au stade expérimental.

Mini-Projet
Sonic doit pouvoir se déplacer vers la droite et vers la gauche (utilisation des flèches du clavier). L'animation devra être adaptée à la situation (marche vers la gauche, marche vers la droite, attente en regardant vers la gauche, attente en regardant vers la droite)

Chapitre 7
dessiner avec EaselJS

EaselJS permet de dessiner aussi bien des formes simples (rectangles, cercles) que des formes complexes (courbes de Bézier). Nous allons uniquement aborder ici les formes simples, pour les formes complexes, je vous invite à consulter la documentation officielle.

Création d'un rectangle

Code 1_7

```
var monCanvas;
var scene;
function init(){
	monCanvas = document.getElementById("mon_canvas");
	scene=new createjs.Stage(monCanvas);
	//nous créons un objet de type Graphics
	var graph=new createjs.Graphics();
	//nous commençons notre dessin en définissant la couleur
	graph.beginFill("rgb(0,255,125)");
	//nous dessinons un rectangle de 80 pixels de largeur et de 50 pixels de hauteur
	// aux coordonnées (0,0)
	graph.drawRect(0,0,80,50);
	//nous créons un objet de type Shape
	var sha=new createjs.Shape(graph);
	scene.addChild(sha);
	scene.update();
}
window.onload=init;
```

Pour dessiner un rectangle, il est nécessaire de créer 2 types d'objets : un objet de type Graphics (`var graph=new createjs.Graphics();`) et un objet de type Shape (`var sha=new createjs.Shape(graph);`).
L'objet de type Graphics est utilisé comme argument du constructeur de l'objet Shape.

La méthode beginFill (`graph.beginFill("rgb(0,255,125)");`) permet :
- de définir le type de "graphisme" (ici, nous dessinons une figure dite "pleine").
- de définir la couleur utilisée (`rgb(0,255,125)`) grâce aux canaux Rouge, Vert, Bleu

La méthode drawRect (`graph.drawRect(0,0,80,50);`) dessine le rectangle. Les paramètres sont : (coordonnée x du coin haut-gauche du rectangle (ici 0), coordonnée y du coin haut-gauche du rectangle (ici 0), largeur du rectangle (ici 80 pixels), hauteur du rectangle (ici 50 pixels)).

Il ne faut pas oublier d'ajouter ensuite notre objet de type Shape à la scène

Notez qu'il est possible de simplifier grandement le code permettant de dessiner ce triangle en utilisant les 2 lignes suivantes :

```
var sha=new createjs.Shape();
sha.graphics.beginFill("rgb(0,255,125)").drawRect(0, 0, 80, 80);
```

à la place de :

```
var graph=new createjs.Graphics();
graph.beginFill("rgb(0,255,125)");
graph.drawRect(0,0,80,50);
var sha=new createjs.Shape(graph);
```

Il est bien sûr possible de "déplacer" notre figure après sa création en utilisant les attributs x et y de notre objet "sha"

Code 2_7

```
var monCanvas;
var scene;
function init(){
	monCanvas = document.getElementById("mon_canvas");
	scene=new createjs.Stage(monCanvas);
	var sha=new createjs.Shape();
	sha.graphics.beginFill("rgb(255,0,0,0)").drawRect(0,0,80,50);
	scene.addChild(sha);
	sha.x=150;
	sha.y=50;
	scene.update();
}
window.onload=init;
```

A faire vous même
Modifier le programme 2_7 pour que le rectangle traverse le canvas de gauche à droite.

BeginFill permet de dessiner des figures "pleines", BeginStroke permet de dessiner uniquement les contours de la figure.

Code 3_7

```
var monCanvas;
var scene;
function init(){
	monCanvas = document.getElementById("mon_canvas");
	scene=new createjs.Stage(monCanvas);
	var sha=new createjs.Shape();
	sha.graphics.beginStroke("rgb(255,0,0)").drawRect(0,0,80,50);
	scene.addChild(sha);
	sha.x=150;
	sha.y=50;
	scene.update();
}
window.onload=init;
```

Nous allons maintenant tracer un cercle en remplaçant la méthode drawRect par la méthode drawCircle

Code 4_7

```
var monCanvas;
var scene;
function init(){
	monCanvas = document.getElementById("mon_canvas");
	scene=new createjs.Stage(monCanvas);
	var sha=new createjs.Shape();
	sha.graphics.beginStroke("rgb(255,0,0)").drawCircle(0,0,50);
	scene.addChild(sha);
	sha.x=150;
	sha.y=80;
	scene.update();
}
window.onload=init;
```

Comme vous l'avez sans doute deviné, la méthode drawCircle prend 3 paramètres : coordonnée x du centre du cercle, coordonnée y du centre du cercle, rayon du cercle)

Il est possible de dessiner beaucoup d'autres figures avec EaselJS. Je vous invite à consulter la documentation officielle d'EaselJS, notamment la partie concernant les objets de type Graphics : http://www.createjs.com/Docs/EaselJS/classes/Graphics.html

2e partie
Pour aller plus loin...

Chapitre 8
La programmation orientée objet en JavaScript

Nous allons provisoirement « abandonner » EaselJS pour nous intéresser à la programmation orientée objet en JavaScript.

JavaScript est un langage orienté objet, nous allons donc avoir à manipuler des "objets informatiques". Les objets informatiques ressemblent beaucoup aux objets de la vie de tous les jours, c'est d'ailleurs un peu l'intérêt de la chose.

Mais avant d'aller plus loin, faisons une parenthèse. Si parmi vous certains ont déjà eu l'occasion de rencontrer cette notion d'objet, notamment en travaillant sur des langages comme le Java ou le C++, il important que vous sachiez que JavaScript est un langage orienté objet par prototype. Ce mot prototype change beaucoup de choses, je vous invite donc à "oublier" (provisoirement) ce que vous avez déjà appris sur les objets (classe, instance,...), nous allons tout reprendre à zéro.

Comme déjà dit dans le premier chapitre un objet possède des attributs et des méthodes. Reprenons l'exemple de la voiture déjà évoqué et développons-le :

Une voiture a une marque et une couleur (entre autres) et bien "marque" et "couleur" seront des attributs de notre objet JavaScript voiture.

Une voiture peut accélérer, freiner, tourner..... et bien "freiner", "accélérer" et "tourner" seront des méthodes de notre objet JavaScript voiture.

Comment "fabrique"-t-on un objet ?

Il existe plusieurs façons de créer un objet, mais, toujours dans un souci d'aller au plus rapide, je ne vais vous en présenter qu'une (mais vous avez eu l'occasion d'en rencontrer une autre dans le chapitre sur la gestion du clavier : `{gauche:0, droite:0}`) .

Avant de pouvoir créer l'objet en tant que tel, nous allons devoir construire un moule. À partir de ce moule, nous pourrons fabriquer autant d'objets que nécessaire. En POO (programmation orientée objet), ce moule s'appelle un constructeur. Un constructeur est une méthode qui porte le même nom que votre "classe" (dans notre cas Voiture). Le nom du constructeur doit commencer par une majuscule.

Un exemple avec, pour commencer, uniquement des attributs (nous verrons les méthodes plus loin)

```
function Voiture (marque, couleur) {
               this.marque = marque;
               this.couleur = couleur ;

}
```

Vous pouvez constater que notre méthode "Voiture" (notre constructeur d'objet de type voiture) peut posséder des arguments (marque et couleur). Ces arguments seront renseignés au moment de la création de nos objets voitures (souvenez-vous que pour l'instant nous avons uniquement fabriqué le moule).
Passons maintenant au contenu de ce constructeur. Vous avez sans doute remarqué le "this.couleur" et le "this.marque".
"this" est un mot clé très important en POO, il désigne l'objet courant, celui que l'on est en train d'utiliser. Oui, je sais, c'est un peu compliqué, nous reviendrons dessus un peu plus tard.
Maintenant que notre "moule" (constructeur) est prêt, passons à la création de notre premier objet.

```
function Voiture (marque, couleur) {
               this.marque = marque;
               this.couleur = couleur ;

}
var voit_1 = new Voiture ("Fiat", "rouge");
```

Nous avons créé une "variable" (voit_1) contenant un objet de type Voiture, nous avons donc créé

un objet voit_1 qui est de type voiture. Pour créer cet objet, nous avons utilisé le mot clé "new" suivi du nom du constructeur (Voiture). Comme pour n'importe quelle fonction, les paramètres " Fiat" et "rouge" sont mis dans la parenthèse qui suit le nom du constructeur.

Nous pouvons créer un deuxième objet de type voiture :

```
function Voiture (marque, couleur) {
               this.marque = marque;
               this.couleur = couleur ;

}
var voit_1 = new Voiture ("Fiat", "rouge");
var voit_2 = new Voiture ("Renault", "verte");
```

Nous avons maintenant 2 "voitures" à notre disposition : une de marque Fiat et de couleur rouge et une de marque Renault et de couleur verte.

Avoir créé ces 2 objets c'est bien, mais comment les utiliser ?

Il est relativement simple d'accéder aux attributs d'un objet :
nom_de_l'objet.nom_de_l'attribut
La notation pointée (c'est son nom) n'est pas la seule possibilité, mais c'est la seule que nous verrons ici.
Dans notre exemple pour accéder à l'attribut couleur de l'objet voit_1, il faudra écrire :
voit_1.couleur
de même, pour accéder à l'attribut marque de l'objet voit_2 :
voit_2.marque
Voici un exemple :

```
function Voiture (marque, couleur) {
               this.marque = marque;
               this.couleur = couleur ;
}
var voit_1 = new Voiture ("Fiat", "rouge");
document.write("Je viens de créer une nouvelle voiture, elle est ",voit_1.couleur,". Sa marque est
", voit_1.marque);
var voit_2 = new Voiture ("Renault", "verte");
document.write("Je viens de créer une nouvelle voiture, elle est ",voit_2.couleur,". Sa marque est
", voit_2.marque);
```

résultat

Je viens de créer une nouvelle voiture, elle est rouge. Sa marque est Fiat
Je viens de créer une nouvelle voiture, elle est verte. Sa marque est Renault

NB :

document.write permet d'afficher dans un navigateur la chaîne mise en argument

Par souci de simplification je n'ai pas codé les retours à la ligne. Si vous testez ces exemples avec le code tel quel, il n'y aura pas de retour à la ligne.

Il est bien évidemment possible de modifier l'attribut d'un objet (comme n'importe quelle variable) :

```
function Voiture (marque, couleur) {
               this.marque = marque;
               this.couleur = couleur ;

}
var voit_1 = new Voiture ("Fiat", "rouge");
document.write("Je viens de créer une nouvelle voiture, elle est ",voit_1.couleur,". Sa marque est
```

```
", voit_1.marque);
var voit_2 = new Voiture ("Renault", "verte");
document.write("Je viens de créer une autre voiture, elle est ",voit_2.couleur,". Sa marque est ",
voit_2.marque);
voit_1.couleur="jaune";
document.write("Je viens de repeindre ma ",voit_1.marque, ", elle est maintenant ",voit_1.couleur);
```

résultat

Je viens de créer une nouvelle voiture, elle est rouge. Sa marque est Fiat
Je viens de créer une autre voiture, elle est verte. Sa marque est Renault
Je viens de repeindre ma Fiat, elle est maintenant jaune

Il est maintenant possible de mieux comprendre l'utilité du mot clé this. Au moment de la création de l'objet voit_1, le mot clé this a systématiquement été remplacé par "voit_1" (même chose pour "voit_2"). Cela correspond bien à la "définition" que je vous ai donnée un peu au-dessus, "this correspond à l'objet courant" ("this=voit_1" quand on crée ou qu'on utilise voit_1 et "this=voit_2" quand on crée ou qu'on utilise l'objet voit_2).

Avant de parler des méthodes, nous devons nous intéresser aux prototypes :

En JavaScript, tous les objets sont liés à un autre objet appelé prototype. Chaque objet possède son propre prototype. Si vous créez un objet de type Voiture, le prototype de Voiture sera accessible tout simplement en écrivant « Voiture.prototype ».

A quoi sert le prototype ?

Question complexe, j'aurais l'occasion d'y répondre un peu plus loin, quand nous aborderons la notion d'héritage. Pour l'instant, vous devez uniquement connaître la procédure à suivre pour créer une méthode :

La méthode « n'appartiendra » pas directement à l'objet, mais «appartiendra » à son prototype (ne surtout pas perde de vu qu'un prototype est un objet comme un autre, il peut donc posséder des attributs et des méthodes), pour définir une méthode « accelere » utilisable par les objets de type voiture, il faudra écrire :

Voiture.prototype.accelere=function() {….}

Voici un exemple

```
function Voiture (marque, couleur) {
	this.marque = marque;
	this.couleur = couleur ;
}
Voiture.prototype.accelere = function () {
	document.write ("Votre voiture de marque ",this.marque," accélère");
}
var voit_1 = new Voiture ("Fiat", "rouge");
voit_1.accelere();
```

résultat

Votre voiture de marque Fiat accélère

Un autre exemple

```
function Voiture (marque, couleur) {
	this.marque = marque;
	this.couleur = couleur ;
}
Voiture.prototype.accelere = function () {
	document.write ("Votre voiture de marque ",this.marque," accélère") ;
}
```

```
var voit_1 = new Voiture ("Fiat", "rouge");
var voit_2 = new Voiture ("Renault", "verte");
voit_1.accelere();
voit_2.accelere();
```

résultat

Votre voiture de marque Fiat accélère
Votre voiture de marque Renault accélère

Remarquez bien l'utilisation du this dans la méthode accelere (). Comme pour les attributs, le this va successivement être « remplacé » par voit_1 puis par voit_2.

Revenons sur le processus mis en œuvre ici :
Au moment de l'utilisation de la méthode accelere (par exemple «`voit_1.accelere();`»), le moteur JavaScript va chercher une méthode une méthode accelere dans l'objet Voiture (il ne faut pas perdre de vue que voit_1 est un objet de type Voiture). Ne trouvant pas de méthode accelere dans l'objet Voiture, il va étendre sa recherche au prototype de l'objet Voiture. Il va, dans cet exemple, trouver la méthode recherchée dans ce prototype. S'il ne l'avait pas trouvé, il aurait poursuivi sa recherche dans le prototype du prototype de l'objet voiture..... Au bout d'un moment, il serait arrivé au prototype de l'objet Object (objet de base du JavaScript) et si une fois du plus sa recherche s'était révélée infructueuse, il aurait alors retourné une erreur (du type « méthode non définie »). Tout ce processus utilise ce que l'on appelle « la chaine des prototypes».

Je précise qu'il est possible de définir les méthodes directement dans le constructeur de l'objet Voiture, mais ce procédé est déconseillé (utilisation de beaucoup de mémoire) et je n'en parlerai pas ici.

Chapitre 9
L'héritage en JavaScript

Voici venir sans doute le chapitre le plus difficile de ce document, alors, accrochez-vous :

L'héritage est une notion fondamentale en POO. Pour une fois l'héritage porte bien son nom, imaginons un objet A qui possède un attribut attrA et une méthode methA. Soit maintenant un objet B qui lui possède un attribut attrB et une méthode methA. Comme vous allez le voir, l'idée d'héritage est relativement simple : Si B hérite de A, alors l'utilisateur d'un objet de type B aura à sa disposition atrB et methB (ça c'est normal !) mais il pourra aussi utiliser atrA et methA.
Si vous avez compris cela, vous avez compris le principe de l'héritage en POO.

Dans les langages orientés objet « classiques » (C++ ou Java), l'héritage est relativement simple à mettre en œuvre. En JavaScript, les choses sont plus complexes :

Il existe plusieurs façons de faire de l'héritage en JavaScript, mais nous allons ici en voir qu'une. Ce n'est pas la simplicité qui a guidé mon choix, j'ai choisi de vous exposer ici la méthode respectant (selon moi!) le plus « l'esprit JavaScript ».

Pour commencer, nous allons étudier un cas avec des constructeurs vides et des objets possédant uniquement des méthodes (pas d'attributs). Pour que l'objet de type B puisse utiliser la méthode issue de l'objet A, nous allons devoir utiliser la méthode create de l'objet Object (Object.create()).

Que fait cette méthode ?

Cette méthode permet de créer un objet, le prototype de cet objet correspondant à l'objet passé en paramètre de la méthode create. Rien compris ? Voici un exemple :

var monObj1=Object.create(monObj2)

Nous créons un objet monObj1, cet objet aura pour prototype monObj2.

Pour faire de l'héritage, nous allons faire comme suit :

var B.prototype=Object.create(A.prototype)

Nous créons (plutôt nous recréons) le prototype de l'objet B. Le prototype de l'objet B aura pour prototype le prototype de l'objet A (oui, je sais, c'est compliqué).

Voici un exemple :

```
//constructeur de A (vide mais nécessaire)
function A() {
}
//nous créons la méthode maMethodeA
A.prototype.maMethodeA = function() {
    document.write("je suis une méthode de l'objet A")
};
function B() {
}
//l'héritage est là !
B.prototype = Object.create(A.prototype);
var monObjetB = new B();
//nous utilisons la méthode maMethode avec un objet de type B
monObjetB.maMethodeA();
```

résultat

je suis une méthode de l'objet A

Parcourons la chaîne des prototypes pour l'objet de type B :

- l'objet de type B (monObjetB) ne possède pas de méthode maMethodeA, passons à la suite
- le prototype de l'objet de type B ne possède pas de méthode maMethodeA, suivant
- le prototype du prototype de l'objet de type B (c'est-à-dire le prototype de l'objet de type A) possède la méthode recherchée, fin de la recherche

Il est évidemment possible de définir des méthodes propres à l'objet B. Attention cependant a bien définir ces méthodes après la ligne permettant l'héritage («`B.prototype=Object.create(A.prototype);`»). Sinon votre méthode sera « écrasée ».

Un exemple

```
//constructeur de A (vide mais nécessaire)
function A() {
}
//nous créons la méthode maMethodeA
A.prototype.maMethodeA = function() {
    document.write("je suis une méthode de l'objet A")
};
function B() {
}
//l'héritage est là !
B.prototype = Object.create(A.prototype);
//création de la méthode spécifique à B
B.prototype.maMethodeB=function() {
    document.write("je suis une méthode de l'objet B")
};
var monObjetB = new B();
//nous utilisons la méthode maMethode avec un objet de type B
monObjetB.maMethodeA();
monObjetB.maMethodeB();
```

résultat

je suis une méthode de l'objet A
je suis une méthode de l'objet B

Parfois le constructeur de l'objet B doit appeler le constructeur de l'objet A (par exemple pour que l'objet B puisse profiter des attributs définis dans le constructeur de l'objet A). Pour effectuer cet appel, il est indispensable d'utiliser la méthode call.

Prenons un exemple

```
//constructeur de A
function A() {
    document.write("le constructeur de A a bien été appelé");
    this.monAttrA="je suis un attribut de A";
}
//nous créons la méthode maMethodeA
A.prototype.maMethodeA = function() {
    document.write("je suis une méthode de l'objet A")
};
function B() {
    //cette ligne permet d'appeler le constructeur de A
    A.call(this);
}
//l'héritage est là !
B.prototype = Object.create(A.prototype);
//création de la méthode spécifique à B
B.prototype.maMethodeB=function() {
    document.write("je suis une méthode de l'objet B")
};
var monObjetB = new B();
//nous utilisons la méthode maMethode avec un objet de type B
monObjetB.maMethodeA();
```

```
monObjetB.maMethodeB();
document.write(monObjetB.monAttrA);
```

résultat

le constructeur de A a bien été appelé
je suis une méthode de l'objet A
je suis une méthode de l'objet B
je suis un attribut de A

Le this dans «`A.call(this);`» est très important, il permet d'utiliser l'attribut de l'objet A avec un objet de type B (le this en paramètre de la méthode call permet de passer une référence à l'instance en cours).
Oui, je sais, cela devient très compliqué, mais rassurez-vous, vous n'avez pas vraiment besoin de comprendre le « pourquoi du comment », vous devez juste retenir que si vous avez besoin d'appeler le constructeur de l'objet parent (ici A) depuis le constructeur de l'objet enfant (B) il faudra utiliser call comme indiqué ci-dessus.

Et si mon constructeur parent prend des paramètres, comment faire ?

Il faut alors utiliser le deuxième paramètre de la méthode call

un exemple

```
function A(color) {
    this.monAttrA=color;
}
A.prototype.maMethodeA = function() {
    document.write(this.monAttrA)
};
function B(couleur, nombre) {
    //cette ligne permet d'appeler le constructeur de A et de lui passer un paramètre
    A.call(this,couleur);
    this.nbr=nombre
}
B.prototype = Object.create(A.prototype);
var monObjetB = new B("vert",12);
document.write("couleur : ",monObjetB.monAttrA,", nombre : ",monObjetB.nbr);
```

Je vous laisse étudier cet exemple, normalement il est assez compréhensible.

Pour terminer ce chapitre, une petite mise en garde : la méthode create est issue d'ECMAScript5, elle fonctionnera donc uniquement avec des navigateurs récents. Il existe des solutions pour contourner le problème (polyfill)
voir ici :
https://developer.mozilla.org/enUS/docs/JavaScript/Reference/Global_Objects/Object/create

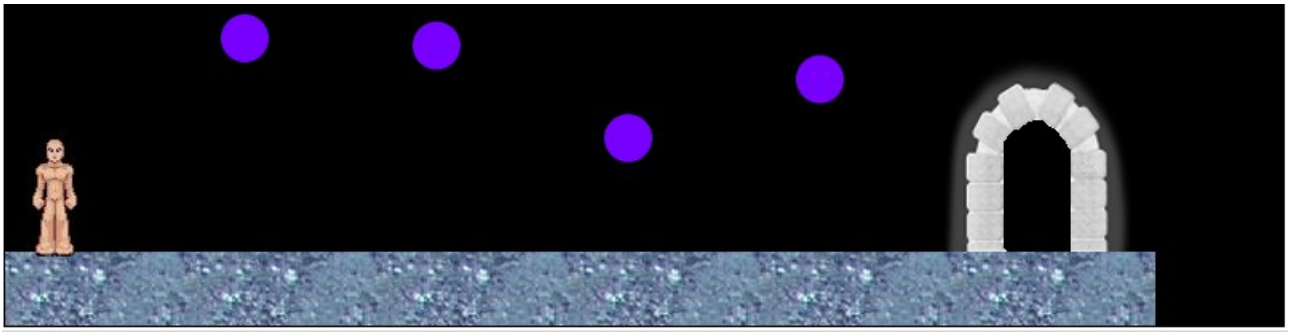
Chapitre 10
Projet : construire un jeu de A à Z

Les 2 derniers chapitres ont été très théoriques. Pour terminer ce document, nous allons faire du « 100 % pratique » puisque nous allons coder un petit jeu. Enfin, je devrais plutôt dire : vous allez coder un petit jeu.
Je vais vous donner le cahier des charges, ensuite cela sera à vous de jouer.

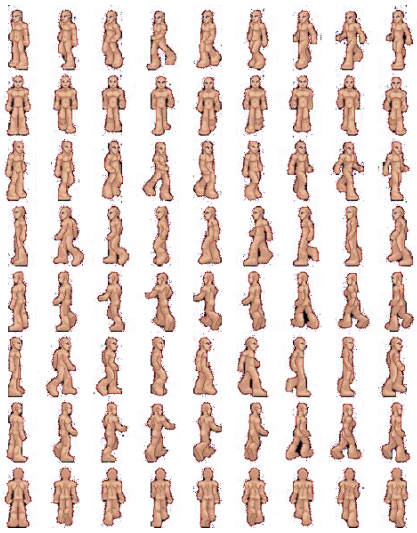
Cahier des charges

- votre jeu devra utiliser EaselJS
- votre jeu s'appuiera sur la POO
- le code devra être réparti dans plusieurs fichiers (en gros, 1 fichier par objet)

voici une capture d'écran du jeu à réaliser (si le jeu n'avait pas déjà été réalisé, un schéma aurait pu suffire)



voici les images (sprites) à utiliser



Toutes ces images sont disponibles ici :
http://webisn.byethost5.com/intro_easeljs

Le but du jeu est simple : le personnage (dirigé par le joueur à l'aide des flèches « gauche » et « droite ») doit atteindre la porte sans se faire toucher par les sphères magiques. Une fois devant la porte, le personnage doit quitter la pièce à l'aide de la touche « haut ».

D'un point de vue plus technique :
Le déplacement du personnage devra s'accompagner d'animations appropriées (en cas d'appui sur la touche « flèche haut » on devra voir le dos du personnage).

Les sphères ne devront pas se déplacer de façon synchrone (positions d'origine et vitesses différentes).

Le joueur perd si :
- il est touché par une sphère magique
- il va trop loin et tombe dans le « puits sans fond »
La mort du personnage devra être notifiée au joueur à l'aide d'une fenêtre « surgissante » (utilisation d' « alert()»).

Le joueur gagne s'il sort de la pièce par la porte. La victoire du personnage devra aussi être notifiée au joueur par une fenêtre « surgissante ».

En cas de victoire ou de défaite, la validation du joueur (au niveau de la fenêtre « surgissante ») recharge la page (et donc, relance le jeu). Pour recharger la page, vous pouvez utiliser «window.location.reload()»