JavaScript & DOM

JavaScript est un langage événementiel orienté objet à prototype. Ce langage est utilisé par les programmeurs pour introduire de nouveaux comportements sur l'affichage des documents Web au sein d'un navigateur Internet. Pour ce faire, le navigateur Web met une interface appelée DOM (**D**ocument **O**bject **M**odel) à la disposition du programmeur. Le DOM s'appuie sur les notions d'objets, de prototypes et d'événements en JavaScript.

Sommaire

1. Les Objets

1.1 Préambule

A Noter : Nous avons opté pour une approche progressive du langage. Nous verrons plus tard que les types de données, tels qu'ils vous sont présentés ici, ne sont pas tout à fait conforme à la réalité. Cet approche simplifiée du langage s'étoffera au fur et à mesure pour être conforme à la réalité.

En JavaScript, à ce point du cours, on distingue pour l'instant « 6 types de données » que nous avons décrit comme suit :

1. Les « codes d'erreur » :

```
undefined; //Non défini : correspond à une variable non définie ou à une valeur jamais initialisée.
null; //Nul : correspond à une variable initialisée avec une valeur vide.
Infinity; //Infini : correspond au résultat d'une division par zéro : valeur symbolique.
NaN; //Not A Number, n'est pas un nombre : correspond à un résultat d'opération primitive qui ne serait pas un nombre : valeur symbolique.
```

1. Les nombres (Number) :

```
100; //Un nombre entier : Number.
42.24; //Un nombre flottant : Number.
```

1. Les chaînes de caractère (String) :

```
'Ceci est un texte'; //Une chaîne de caractère : String.
```

1. Les booléens (Boolean) :

```
true; //Un booléen : Boolean.
```

1. Les fonctions (Function) :

```
(function(){
   //Instructions
  ;;;;;;;;;;;;;;;
}); //Une fonction, ici une fonction anonyme : Function.

var maFonctionAnonyme = function(){
   //Autres instructions
  ;;;;;;;;;;;;;;;
}; //Une fonction, ici une autre fonction anonyme affectée à une variable : Function.

var maFonction = function maFonctionNommee(){
   //Encore d'autres instructions
  ;;;;;;;;;;;;;;;
}; //Une fonction, ici une fonction nommée affectée à une variable : Function.
```

1. Les tableaux (Array) :

```
[]; //Un tableau, ici un tableau vide : Array.
[99,'hello !',false,function(){},[]]; //Un tableau, ici un tableau contenant des valeurs : Array.
```

Nous allons introduire un nouveau type de données : **l'objet**. En anglais, Object.

Les objets sont des structures de données complexes, un peu à la manière des tableaux. Les objets peuvent être **créés par le programmeur** à l'aide d'une syntaxe particulière ou être **fournis par un ou plusieurs systèmes**. Lorsqu'il sont fournis, on peut les utiliser de diverses façon au sein d'un programme. On parle alors d'API (Application Programming Interface - ensemble de moyens permettant de programmer) objet.

1.2 Notation Objet JavaScript : JSON

La façon la plus simple de créer un objet est d'utiliser **JSON**. JSON est l'acronyme de **J**avaScript **O**bject **N**otation; en français, Notation Objet JavaScript.

Exemple de **création** d'un objet en JSON :

```
{}; //Un objet, ici un objet vide : Object.
```

1.2.1 Notion de propriété :

Un objet peut contenir des **propriétés** nommées. Les propriétés d'un objet peuvent contenir n'importe quel « type de données » définis au préambule.

Exemple d'objet en JSON avec des propriétés nommées :

```
{
  aa:null, //Propriété aa contenant null.
  bb:49.9, //Propriété bb contenant un nombre.
  cc:'Hello', //Propriété cc contenant une chaîne de caractères.
  dd:function(){alert('Ceci est un test !');}, //Propriété dd contenant une fonction.
  ee:[2013,2014,2015] //Propriété ee contenant un tableau.
}; //Un objet, ici un objet contenant des propriétés : Object.
```

On notera que les noms des propriétés doivent être **uniques** au sein d'un même objet. Les noms des propriétés ne peuvent pas contenir de caractères spéciaux (comme les tirets, les espaces, ...).

Une **référence** vers l'objet peut être affectée à une variable. Par exemple :

```
var maVariable = {
  aa:null, //Propriété aa contenant null.
  bb:49.9, //Propriété bb contenant un nombre.
  cc:'Hello', //Propriété cc contenant une chaîne de caractères.
  dd:function(){alert('Ceci est un test !');}, //Propriété dd contenant une fonction.
  e:[2013,2014,2015] //Propriété ee contenant un tableau
}; //Un objet, ici un objet contenant des propriétés : Object.
```

Pour **accéder aux propriétés d'un objet** dont a affecté la référence à une variable, on utilisera l'opérateur « `.` » (le point). Exemple :

```
maVariable.aa; //Contient null.
maVariable.bb; //Contient 49.9.
maVariable.cc; //Contient 'Hello'.
maVariable.dd; //Contient function(){alert('Ceci est un test !');}.
maVariable.ee; //Contient [2013,2014,2015].
```

On peut également utiliser une notation à base de « `[]` » (crochets) en fournissant la chaîne de caractères correspondant à la variable. Exemple :

```
maVariable['aa']; //Contient null.
maVariable['bb']; //Contient 49.9.
maVariable['cc']; //Contient 'Hello'.
maVariable['dd']; //Contient function(){alert('Ceci est un test !'>);}.
maVariable['ee']; //Contient [2013,2014,2015].
```

On a tendance à appeler parfois les objets JavaScript « tableaux associatifs » mais ce sont bel et bien des objets !

1.2.2 Utilisation des propriétés :

On peut **utiliser les propriétés** d'un objet « comme n'importe quelle variable ». Exemple :

```
maVariable.aa; //Contient le type null.
maVariable.aa = 'Kitty !'; //Ici on affecte la chaîne de caractères 'Kitty !' à la propriété aa de l'objet référencé dans maVariable.
maVariable.aa; //Contient désormais 'Kitty !'.

maVariable.cc; //Contient la chaîne de caractères 'Hello'.
maVariable.cc = maVariable.cc + ' ' + maVariable.aa; //Ici on affecte la concaténation des chaînes de caractère de la propriété cc ('Hello'), la chaine de caractère ' ' et de la propriété aa ('Kitty !') à la propriété cc de l'objet référencé dans maVariable.
maVariable.cc; //Contient la chaîne de caractères 'Hello Kitty !'.

maVariable.dd; //Contient la fonction function(){alert('Ceci est un test !');}.
maVariable.dd(); //Exécute la fonction contenue dans dd.
maVariable['dd'](); //Exécute également la fonction contenue dans dd.

maVariable.ee; //Contient le tableau [2013,2014,2015].
maVariable.ee[1]; //Contient le nombre 2014.
maVariable.ee[1] = 2001; //Affecte le nombre 2001 à l'emplacement 1 du tableau dans la propriété ee.
maVariable.ee[2] = maVariable.ee[1] + 9; //Affecte la somme de 2001 et de 9 à l'emplacement 2 du tableau dans la propriété ee.
maVariable.ee[2]; //Contient 2010.
```

Si on essaie d'accéder à une propriété qui **n'existe pas**, le moteur nous donne « `undefined` ». Exemple :

```
maVariable.nExistePas; //Contient undefined.
```

En revanche, si on affecte une valeur à une propriété qui n'existe pas dans l'objet, le moteur l'**ajoute automatiquement** dans l'objet. Exemple :

```
maVariable.sortieDeNullePart = 'Monolithe'; //Affecte la chaîne de caractère 'Monolithe' à la propriété sortieDeNullePart qui n'existait pas.
maVariable.sortieDeNullePart; //Contient 'Monolithe'.
maVariable['je viens du vide'] = 'Big Bang'; //Affecte la chaîne de caractère 'Big Bang' à la propriété 'je viens du vide' qui n'existait pas.
maVariable['je viens du vide']; //Contient 'Big Bang'.
```

On remarque que la notation avec les « `[]` » présente l'avantage de permettre la création de propriétés dont les noms contiennent des caractères spéciaux.

Une propriété d'un objet peut contenir un **sous-objet** qui peut lui-même contenir des propriétés. Les propriétés du sous-objet peuvent également contenir des objets. On parle alors d'**objets imbriqués** (en anglais, "*nested objects*"). Par exemple :

```
var terre = {
  pays:{
    nom:'France',
    ville:[{
      nom:'Paris',
      type:'Capitale'
    }]
  }
}; //Déclaration d'un objet contenant un sous objet qui contient lui même un sous objet et ainsi de suite...

//Comment accède-t-on aux propriétés des sous objets ?
terre.pays; //Contient {nom:'France',ville:{nom:'Paris',type'Capitale'}}
terre.pays.nom; //Contient 'France'
terre.pays.ville; //Contient [{nom:'Paris',type'Capitale'}]
terre.pays.ville[0]; //Contient {nom:'Paris',type'Capitale'}
terre.pays.ville[0].nom; //Contient 'Paris'
```

1.2.3 Objets, propriétés et fonctions :

Une fonction peut **prendre en argument un ou plusieurs objets** comme n'importe quel autre type de données. Par exemple :

```
var quiPartDansLEspace = function(astronaute,cosmonaute){
  alert(cosmonaute.nom + ' et ' + astronaute.nom + ' décollent pour Jupiter.');
}; //de toute évidence la fonction quiPartDansLEspace attend en entrée 2 arguments qui représentent des entités distinctes.

quiPartDansLEspace({nom:'Heywood', prenom:'Floyd'},{nom:'Tatiana', prenom:'Orlov'}); //Ici on exécute la fonction quiPartDansLEspace en lui fournissant 2 objets en entrée. Cela produira l'affichage du message 'Orlov et HeyWood décollent pour Jupiter.'
```

On remarque qu'à l'intérieur de la fonction qui prend en argument les 2 objets, on peut accéder aux propriétés contenues dans ces objets (ici la propriété « `.nom` »).

Et les fonctions peuvent **retourner un objet** comme elles peuvent le faire avec n'importe quel autre type de données. Par exemple :

```
var shazam = function(){
  var objetMagique = {
    doubleFond:'colombe',
    tripleFond:'lapin blanc'
  }; //On affecte une référence vers cet objet à la variable objetMagique.
  return objetMagique; //On retourne la référence vers l'objet à l'extérieur de la fonction.
}; //Ici la fonction shazam retourne un objet qui contient 2 propriétés : doubleFond et tripleFond.

var chapeau = shazam(); //Ici on exécute la fonction shazam et on affecte la valeur de retour à la variable chapeau.
//La variable chapeau contient désormais une référence à un objet, l'objet retourné par la fonction shazam.
chapeau.doubleFond; //Contient 'colombe'.
chapeau.tripleFond; //Contient 'lapin blanc'.
```

On remarque qu'on peut accéder aux propriétés de l'objet retourné par la fonction « `shazam` » en utilisant la variable « `chapeau` » à laquelle on a affecté la valeur de retour (référence vers un objet).

On aurai pu se passer de variable « `chapeau` » et utiliser directement la référence à un objet retournée par la fonction « `shazam` » en écrivant :

```
shazam().doubleFond; //Contient 'colombe'.
shazam().tripleFond; //Contient 'lapin blanc'.
//Nous savons que la fonction shazam retourne une référence à un objet; nous utilisons donc directement cette référence
retournée par la fonction.
```

1.2.4 Notion de méthode :

Une **méthode** est une « fonction dans une propriété d'un objet ». Par exemple :

```
var voiture = {
  chevaux:4,
  modele:'R5',
  marque:'Rono',
  roule:function(){ //Ici, une propriété contenant une fonction : une méthode.
    alert('Vroooooom !');
  }
}; //Ceci est un objet sensé représenter une voiture.

//Exécution de la méthode :
voiture.roule(); //Affiche Vroooooom ! dans une boite de dialogue.
```

Les méthodes, comme les fonctions, peuvent prendre des arguments en entrée et fournir une valeur de retour. Par exemple :

```
var calculatrice = {
  addition:function(x,y){
     return x + y;
  }, //Méthode qui prend en argument 2 nombres et retourne la somme des deux.
  soustraction:function(x,y){
     return x - y;
  }, //Méthode qui prend en argument 2 nombres et retourne la soustraction premier par le second.
  multiplication:function(x,y){
    return x * y;
  }, //Méthode qui prend en argument 2 nombres et retourne la multiplication des deux.
  division:function(x,y){
     return x / y;
  } //Méthode qui prend en argument 2 nombres et retourne la division du premier par le second.
}; //Ceci est un objet sensé représenter une calculatrice.

//Utilisation :
var resultat = calculatrice.somme(1,2); //L'exécution de la méthode .somme() avec 1 et 2 comme valeurs d'arguments retourne le
nombre 3.
resultat; //Contient 3.

var autreResultat = calculatrice.multiplication(9,9); //L'exécution de la méthode .multiplication() avec 9 et 9 comme valeurs
d'arguments retourne le nombre 81.
autreResultat; //Contient 81.
```

1.2.5 Mot-clé « `this` » :

A l'intérieur d'une méthode et, plus généralement, d'un objet, on a la possibilité d'utiliser le mot-clé « `this` ». Ce mot-clé est une **référence à l'objet** dans lequel il est utilisé, « il pointe vers l'objet » dans lequel il est utilisé. Par exemple :

```
var lapin = {
  cri:'Groink Groink...',
  glapit:function(){

     this; //Fait référence à l'objet lui même.
     this.cri; //Contient 'Groink Groink...'.
     this.cri = 'Squiiik !'; //Remplace 'Groink Groink...' par 'Squiiik !'.
     this.cri; //Contient 'Squiiik !'

     alert(this.cri);
  }
};

//Exécution de la méthode .glapit() :
lapin.glapit(); //Affiche 'Squiiik !' dans une boite de dialogue
```

Dans cet exemple, le mot-clé « `this` » a été utilisé pour modifier une propriété de l'objet auquel il faisait référence.

Ce mot-clé ne peut être utilisé qu'à l'intérieur d'un objet. Il faut faire **très attention** en l'utilisant à bien identifier l'objet auquel il fait référence surtout dans le cas des objets imbriqués. Par exemple :

```
var stade = {
  nom:'Parc des Princes',
  sponsor:'Citroën',
  quiEstMonSponsor:function(){
     alert(this.sponsor); //Ici, this fait référence au stade.
  },
  equipe:{
    nom:'Paris Saint Germain',
    sponsor:'Nike',
    quiEstMonSponsor:function(){
       alert(this.sponsor); //Ici, this fait référence à l'équipe.
    },
    joueurs:[{
      nom:'Zlatan',
      sponsor:'Puma',
      quiEstMonSponsor:function(){
        alert(this.sponsor); //Ici, this fait référence à ce joueur.
      }
    },{
      nom:'Pastore',
      sponsor:'Nike',
      quiEstMonSponsor:function(){
         alert(this.sponsor); //Ici, this fait référence à cet autre joueur.
      }
    }]
  }
};

stade.nom; //Contient 'Parc des Princes'.
stade.quiEstMonSponsor(); //Affiche 'Citroën' dans une boite de dialogue.
stade.equipe.nom; //Contient 'Paris Saint Germain'.
stade.equipe.quiEstMonSponsor(); //Affiche 'Nike' dans une boite de dialogue.
stade.equipe.joueurs[0].nom; //Affiche 'Zlatan'.
stade.equipe.joueurs[0].quiEstMonSponsor(); //Affiche 'Puma'.
```

Dans cet exemple, tous les objets contiennent la propriété « `.sponsor` » mais cette dernière ne contient pas la même valeur selon l'objet dans lequel elle se trouve.

1.2.6 : Unicité des objets :

Imaginons que j'achète une voiture. Cette voiture est une Citroën 2 CV que mes voisins appellent couramment "deudeuche". Je suis tellement attaché à cette voiture que je lui ai donné un prenom : "Berthe". Les noms "deudeuche" et "Berthe" font référence à la même voiture. On pourrait illustrer cet exemple comme suit :

```
var voiture = {
  modele:'Citroën 2CV',
  etat:'Abimé'
}; //La variable voiture contient une référence à un objet qui correspond à un véhicule.

var deudeuche = voiture; //On copie la référence au véhicule dans la variable deudeuche.
var Berthe = voiture; //On copie la référence au véhicule dans la variable Berthe.
```

Dans cet exemple, les variables « `voiture` », « `dedeuche` » et « `Berthe` » contiennent une **référence qui pointe vers le même objet**. En d'autres termes, quel que soit le nom porté par ma voiture, c'est toujours la même voiture. On constatera donc que :

```
/*
  Si, ici, je modifie la propriété .etat de l'objet vers lequel pointe la référence contenue dans la variable.
*/
deudeuche.etat = 'Neuf';

/*
  On Constatera que :
*/
voiture === Berthe; //Donne true.
Berthe === deudeuche; //Donne true.
deudeuche === voiture; //Donne true.

dedeuche.etat; //Contiendra 'Neuf'.
Berthe.etat; //Contiendra 'Neuf'.
voiture.etat; //Contiendra 'Neuf'.
```

Lorsqu'un objet est créé en mémoire, il est **unique**. L'opérateur d'affectation « = » ne permet que d'assigner à une variable une référence à cet objet. Donc en utilisant le « = » on ne peut pas « copier » un objet. Si on souhaite créer un nouvel objet similaire au précédent, il faut le déclarer. Par exemple :

```
var laVoitureDeMonVoisin = {
  modele:'Citroën 2CV',
  etat:'Neuf'
}; //Ici, on affecte une référence à ce nouvel objet à la variable laVoitureDeMonVoisin

/*
  L'objet vers lequel pointe la référence contenue dans Berthe et l'objet vers lequel pointe la référence contenue dans Berthe
sont similaires en tous points mais ne sont pas les les mêmes. Ce sont deux objets différents dans deux emplacements mémoire
différents.
*/

laVoitureDeMonVoisin == Berthe; //Donne false, ces deux variables contiennent des références à des objets différents.
```

On verra par la suite des mécanismes permettant de "cloner" des objets ou de créer des objets sans avoir à réécrire toute la structure des objets.

1.3 Fonctions constructeur

Une autre façon de créer des objets est d'utiliser une **fonction constructeur** (en anglais, "*constructor function*"). Une fonction constructeur est une fonction qui décrira les caractéristiques (propriétés et méthodes) d'un objet. Elle pourra être utilisée autant de fois que nécessaire pour créer autant d'objets dotés de ces caractéristiques.

Chaque fois qu'on souhaite utiliser une **fonction constructeur** pour créer un objet, on doit utiliser le mot-clé « `new` ».

1.3.1 : Déclaration :

On peut déclarer des fonctions constructeur qui seront utilisées pour créer des objets possédant tous les **mêmes propriétés** avec les **mêmes valeurs**.

Par exemple :

```
var chaineDeMontageDeVoitures = function(){
  this.couleur = 'Blanche';
  this.marque = 'Rono';
  this.modele = 'R5';
  this.chevaux = 45;
  this.roule = function(){
    alert('Vrooom !');
  };
};
```

On peut, également, déclarer des fonctions constructeur pour créer des objets possédant tous les **mêmes propriétés** mais avec des **valeurs différentes**. Pour faire cela, on précisera des arguments. On les appelle "arguments du constructeur".

Par exemple :

```
var chaineDeMontageDeVoituresMulticolores = function(argument){
  this.couleur = argument; //Ici, la couleur devra être fournie en argument de la fonction constructeur
  this.marque = 'Rono';
  this.modele = 'R5';
  this.chevaux = 45;
  this.roule = function(){
    alert('Vrooom !');
  };
  this.quelleEstMaCouleur = function(){
    alert(this.couleur);
  };
};
```

Pour utiliser ces 2 fonctions constructeurs pour créer des objets qui posséderont les propriétés et les méthodes définies, on utilisera un mot-clé spécifique : « `new` »

1.3.2 : Mot-clé « `new` » :

Le mot-clé « `new` » sert à créer un objet à partir d'une fonction constructeur.

Par exemple :

```
var premiereVoiture = new chaineDeMontageDeVoitures(); //Premier objet.
var deuxiemeVoiture = new chaineDeMontageDeVoitures(); //Deuxième objet.
var troisiemeVoiture = new chaineDeMontageDeVoitures(); //Troisième objet.

premiereVoiture.modele; //Contient 'R5'.
deuxiemeVoiture.chevaux; //Contient 45.
troisiemeVoiture.roule(); //Affiche une boite de dialogue avec le texte 'Vrooom !'.
```

Chacune des 3 variables contient une référence à un objet différent. Les 3 objets créés sont similaires, ils ont les mêmes propriétés, les mêmes valeurs de propriétés, et les mêmes méthodes.

Autre exemple :

```
var premiereVoitureDeCouleur = new chaineDeMontageDeVoituresMulticolores('rouge'); //Premier objet.
var deuxiemeVoitureDeCouleur = new chaineDeMontageDeVoituresMulticolores('bleu'); //Deuxième objet.
var troisiemeVoitureDeCouleur = new chaineDeMontageDeVoituresMulticolores('vert'); //Troisième objet.

premiereVoitureDeCouleur.couleur; //Contient 'rouge'.
deuxiemeVoitureDeCouleur.couleur; //Contient 'bleu'.
troisiemeVoitureDeCouleur.quelleEstMaCouleur(); //Affiche une boite de dialogue avec le texte 'vert'.
```

Chacune des 3 variables contient une référence à un objet différent. Les 3 objets créés sont similaires, ils ont les mêmes propriétés et les mêmes méthodes; mais leurs propriétés ne contiennent pas les mêmes valeurs.

Si on ajoute une propriété à un de ces objets, elle ne s'ajoute pas aux autre objets.

Par exemple :

```
deuxiemeVoitureDeCouleur.prix = '8999€';
premiereVoitureDeCouleur.prix; //undefined.
```

Les fonctions constructeur peuvent représenter une économie de code par rapport à JSON lorsqu'on a besoin de créer plusieurs objets sur le même modèle.

1.3.3 : Fonctions "usine" :

Les "fonctions usine" (en anglais, "*factory function*") sont une **technique de programmation** (en anglais, "*design pattern*) qui consiste à écrire une fonction qui **retourne** une nouvel objet créé à l'aide d'une fonction constructeur ou d'un objet écrit en JSON qui fera office de modèle.

Par exemple :

```
var usineRono = function(){
  return new chaineDeMontageDeVoitures();
}; //Ici, une "fonction usine" qui retourne un nouvel objet lorsqu'elle est exécutée.

voiture1 = usineRono(); //on affecte à voiture1 la référence d'un nouvel d'objet obtenue suite à l'exécution de la fonction usineRono.
voiture2 = usineRono(); //idem.
voiture3 = usineRono(); //idem.

var usineRonoMulticolore = function(tnemugra){
  return new chaineDeMontageDeVoituresMulticolores(tnemugra);
}; //Ici, une "fonction usine" qui prend en argument une valeur et qui retourne un nouvel objet lorsqu'elle est exécutée.

voitureMulticolore1 = usineRonoMulticolore('jaune');
voitureMulticolore2 = usineRonoMulticolore('violet');
voitureMulticolore3 = usineRonoMulticolore('turquoise');
```

On remarque que la technique des fonctions "usine" permet de ne pas avoir à répéter le mot-clé « `new` » à chaque création d'objet.

La technique des fonctions "usine" permet également de créer des variables "**privées**". Une variable privée est une variable qui peut-être utilisée à l'intérieur d'un objet, dans une méthode par exemple; mais à laquelle on ne peut pas accéder directement depuis l'extérieur de l'objet.

Par exemple :

```
//On déclare une "fonction usine".
var ebenisterie = function(){
   //La variable ci-après existe uniquement dans la "fonction usine" ebenisterie.
  var compartimentSecret = 'Plan du Trésor Des Templiers'; //variable "privée", n'existe que dans la "fonction usine" ebenisterie.

   //On déclare une fonction constructeur dans cette "fonction usine".
  var constructeurDeBoite = function(){
    this.couleur = 'marron';
    this.accederAuCompartimentSecret = function(){
       //Ici on utilise la variable compartimentSecret qui est définie plus haut.
      return compartimentSecret;
    };
  }; //Ici, cette fonction constructeur permet de créer des boites avec une propriété et une méthode.

   //On retourne la référence à un nouvel objet obtenu à partir de la fonction constructeur.
  return new constructeurDeBoite();
};

typeof compartimentSecret; //undefined, la variable compartimentSecret n'existe pas dans l'espace global.

var boite = ebenisterie(); //l'execution de la "fonction usine" ebenisterie() nous retourne une référence vers un nouvel objet.

boite.couleur; //'marron', couleur est une propriété de l'objet.

boite.compartimentSecret; //undefined, la variable compartimentSecret n'est pas une propriété de l'objet.

boite.accederAuCompartimentSecret(); //'Plan du Trésor Des Templiers', la méthode .accederAuCompartimentSecret() a accès à la variable définie dans son contexte d'origine.
```

Les variables **privées** sont le plus souvent utilisées pour éviter de surcharger les objets de propriétés auxquelles le programmeur n'a pas spécialement besoin d'accéder depuis l'extérieur de ses objets.

1.4 Constructeur « `Object.create()` »

En JavaScript, le moteur met à la disposition du programmeur plusieurs fonctions, fonctions constructeur et objets (cf. **Objets Natifs**). Les objets contiennent des propriétés et des méthodes utilitaires pour l'aider à programmer avec le langage. L'objet « `Object` » fait partie des objets fournis par le langage. Il contient principalement des méthodes pour aider le programmeur à travailler avec les objets.

Nous allons travailler avec la méthode « `Object.create()` ». Cette méthode propose de créer un nouvel objet à partir d'un objet initial. Cet objet initial peut-être un objet créé à l'aide d'une fonction constructeur ou en JSON.

Exemple d'utilisation de « `Object.create()` » :

```
var lapin = {
  cri:'Squiiik !',
  glapit:function(){
     alert(this.cri);
  }
};

var autreLapin = Object.create(lapin); //On créé un nouvel objet à partir des caractéristiques (propriétés et méthodes) de l'objet initial.

autreLapin.cri = 'Gnaaarl !'; //On redéfini la propriété .cri de l'objet vers lequel pointe autreLapin.

//Exécution de la méthode .glapit() :
lapin.glapit(); //Affiche 'Squiiik !' dans une boite de dialogue
autreLapin.glapit(); //Affiche 'Gnaaarl !' dans une autre boite de dialogue
//Il s'agit bien de deux objets différents (puisque les cris des deux lapins sont différents !).
```

On peut considérer que l'objet « `lapin` » est le **prototype** de l'objet « `autreLapin` » puisqu'on a utilisé l'un pour créer l'autre.

Ceci nous amène une notion fondamentale du modèle objet de JavaScript, le prototypage.

2. Le Prototypage

1. **Préambule**
2. **Notion de prototype**
3. **Création de prototype**

2.1 Préambule

Les langages orientés objet tels que le C++ ou le Java basent leur modèle objet sur la notion de classe et d'instance.

1. Un classe définit des propriétés et des méthodes.
1. Une instance est un objet créé à l'aide d'une classe.

Dans ce cadre, on peut définir ce qu'on appelle un **héritage de classes**. Il s'agit de définir une classe « *parent* » et une classe « *enfant* » qui **hérite** des caractéristiques (propriétés et méthodes) de la classe parent. Si on créé une instance (c'est à dire un objet à partir) de la classe « *parent* », celui-ci aura les caractéristiques définies par la classe « *parent* ». Si on créé une instance (c'est à dire un objet à partir) de la classe « *enfant* », celui-ci aura les caractéristiques définies par la classe « *enfant* » et les caractéristiques de la classe dont elle hérite, la classe « *parent* ».

Le langage JavaScript base son modèle objet sur la notion de **prototype**. Dans ce type de modèle objet, on ne fait **pas de distinction** entre classe et instance, on ne parle que d'**objets**. Dans ce cadre, on peut définir ce qu'on appelle un objet **prototype**. Il s'agit d'un objet qui sert de **modèle** à la construction d'autres objets.

2.2 Bases du prototypage

En JavaScript, **tous les objets sont créés à partir d'un objet prototype** même si le programmeur n'en a pas explicitement défini un. Ainsi, dans le cas d'un objet créé en JSON, il s'agit du prototype de l'**objet « `Object` »**. Cet objet, vide, qui fait office de prototype et qui est imposé par le moteur JavaScript, a lui-même un prototype: l'objet « `null` ». L'objet « `null` », quant à lui, n'a pas de prototype. Tous les objets sont dotés d'une propriété spéciale « `.__proto__` » qui permet d'accéder à leur prototype. On parle de **chaîne de prototypes**.

Par exemple :

```
var poule = {
  cri:'Cot Cooot !',
  caquette:function(){
    alert(this.cri);
  }
}; //cet objet déclaré en JSON définit les caractéristiques d'une poule.

typeof poule; //object : la variable poule contient une référence à un objet.
poule; //{cri:'Cot Cooot !', caquette:function(){this.cri();}}.

typeof poule.__proto__; //object : le prototype de poule est le prototype de l'objet Object.
poule.__proto__; //{}, fait référence à un objet vide.

typeof poule.__proto__.__proto__; //object : le prototype de l'objet vide qui est un objet.
poule.__proto__.__proto__; //null, fait référence à l'objet null.

/*
L'objet null n'a pas de prototype. On ne peut donc pas écrire poule.__proto__.__proto__.__proto__ sans faire planter le moteur !
*/

/*
On pourrait représenter l'enchaînement des prototypes comme suit :
poule ----> Object.prototype ----> null
*/
```

** L'opérateur « `typeof` » renvoi le type de son opérande (de l'information "en argument")*.

On peut redéfinir les prototypes par défaut (fournis par le moteur). On va, par exemple, redéfinir le prototype de l'objet « `poule` ». Toutes les poules sont issues d'une poule préhistorique qui, comme chacun sait, avait des dents. Exemple :

```
var poulePrehistorique = {
  dents:true,
  cri:'Crooark!',
  caractere:'Sadique',
  caquette:function(){
    alert(this.cri);
  }
}; //cet objet déclaré en JSON définit les caractéristiques d'une poule préhistorique.

poule.__proto__ = poulePrehistorique; //Ici, on remplace le prototype par défaut par l'objet correspondant à une poule
préhistorique.

/*
On pourrait représenter l'enchaînement des prototypes comme suit :
poule ----> poulePrehistorique ----> Object.prototype ----> null
*/
```

Un objet à accès **directement** à ses propriétés et ses méthodes, aux propriétés et méthodes de son prototype, aux propriétés et méthodes du prototype de son prototype, etc. Par exemple :

```
poule.caractere; //'sadique' : la propriété caractere n'est pas une propriété de la poule, mais de la poule préhistorique.

poule.dents; //true : la propriété dents n'est pas une propriété de la poule, mais de la poule préhistorique.
```

Les propriétés d'un objet qui auraient le même nom que des propriétés de son prototype « masquent » les propriétés du prototype. Par exemple :

```
poule.dents; //true : la propriété dents n'est pas une propriété de la poule, mais de la poule préhistorique.
poule.dents = false; //Tout le monde sait que les poules n'ont pas dents, donc on met false dans la propriété dents.

poule.dents; //false, la propriété dents de l'objet poule contient false.
poule.__proto__.dents; //true, la propriété dents de l'objet prototype contient toujours true.
```

À l'aide du prototypage, on peut donc créer un **héritage d'objets**. C'est-à-dire des objets ont leurs propres propriétés et méthodes mais qui héritent de celles de tous les objets de leur **chaîne de prototypes**.

Pour des raisons de performance, **on n'utilise pas** la propriété spéciale .__proto__ pour définir un prototype. On préférera d'autres approches.

2.3 Création de prototype

Dans la partie précédente nous avons altéré la chaîne des prototypes d'un objet **après** l'avoir créé. Cela impacte grandement les performances du moteur. On nous recommande ainsi de définir les objets prototypes d'un autre objet **avant** sa création. Pour faire cela, on peut utiliser les fonctions constructeur avec le mot-clé « `new` » ou utiliser le constructeur « `Object.create()` ».

2.3.1 Avec une fonction constructeur

En JavaScript, les fonctions créées à l'aide du mot-clé « `function` » sont traités **comme des fonctions** mais aussi **comme des objets** de type « `Function` » par le moteur JavaScript (cf. **Objets Natifs**). Cela signifie, qu'après sa création, une fonction se voit dotée d'un ensemble de propriétés et de méthodes (comme si elle était un objet).

Les fonctions possèdent, entre autre, une propriété « `.prototype` ». On peut affecter à cette propriété une référence à un objet. Si on utilise la fonction comme constructeur, l'objet affecté à la propriété « `.prototype` » sera ajouté à la chaîne des prototypes de l'objet créé.

Par exemple :

```
var statue = function(nom){
  this.sculpteur = nom;
}; //Ici on déclare une fonction constructeur.

statue.prototype = {
  matiere:'bronze'
}; //Ici on déclare un objet qui sera le prototype de tous les objets créés à l'aide de la fonction constructeur.

var lePenseur = new statue('Auguste Rodin'); //On créé un objet à l'aide de la fonction constructeur.
var mannekenPis = new statue('Jérôme Duquesnoy'); //On créé un autre objet à l'aide de la fonction constructeur.

lePenseur.sculpteur; //'Auguste Rodin', propriété du constructeur.
mannekenPis.sculpteur; //'Jérôme Duquesnoy', propriété du constructeur.

lePenseur.matiere; //'bronze', propriété du prototype.
mannekenPis.matiere; //'bronze', propriété du prototype.

/*
On pourrait représenter l'enchaînement des prototypes comme suit :
statue ----> Function.prototype {matiere:'bronze'} ----> Object.prototype {} ----> null
*/
```

On remarquera que si on modifie le prototype qu'on a ajouté à la fonction constructeur, cela modifiera en conséquence les objets créés avec ce constructeur.

Par exemple :

```
statue.prototype.matiere = 'bronze oxydé'; //On modifie une propriété du prototype.

lePenseur.matiere; //'bronze oxydé', propriété modifiée du prototype.
mannekenPis.matiere; //'bronze oxydé', propriété modifiée du prototype.
```

On constate que l'utilisation de la fonction constructeur demande de la déclarer avant de lui affecter un objet prototype. Or, dans la réalité, on commence généralement par concevoir un prototype avant de passer à la conception du produit fini... Pour adopter un mode de conception plus proche de la réalité, on utilisera le constructeur « `Object.create()` »

2.3.2 Avec le constructeur « `Object.create()` »

On peut donc créer des objets à partir de leur prototype à l'aide de la méthode « `.create()` » de l'objet natif « `Object` ». La méthode « `.create()` » prend en argument un objet et retourne un nouvel objet dont le prototype est l'objet fourni en argument.

Par exemple :

```
var australopitheque = {
 nom:'Australopithecus africanus',
 genre:'humain',
 classe:'mammifère',
 ere:'-3500000 ans',
 outils:false,
 stature:'courbée'
}; //On créé un objet qui fera office de prototype.

var homoHabilis = Object.create(australopitheque); //On créé un nouvel objet à partir de notre prototype.

//Chaine de prototypes : homoHabilis ----> australopitheque ----> Object.prototype ----> null.

//On modifie certaines propriétés de notre nouvel objet.
homoHabilis.nom = 'Homo Habilis';
homoHabilis.ere = '-2500000 ans';
homoHabilis.outils = true;

var homoErectus = Object.create(homoHabilis); //On créé un nouvel objet à partir de l'objet homoHabilis qu'on utilise comme
prototype.

//Chaine de prototypes : homoErectus ----> homoHabilis ----> australopitheque ----> Object.prototype ----> null.

homoErectus.nom = 'Homo Erectus';
homoErectus.ere = '-1500000 ans';
homoErectus.stature = 'debout';

var homoSapiens = Object.create(homoErectus); //On créé un nouvel objet à partir de l'objet homoErectus qu'on utilise comme
prototype.

//Chaine de prototypes : homoSapiens ----> homoErectus ----> homoHabilis ----> australopitheque ----> Object.prototype ---->
null.

homoSapiens.nom = 'Homo Sapiens';
homoSapiens.ere = '-40000 ans';

homoSapiens.genre; //Contient 'humain', propriété du prototype initial « australophitheque »
homoSapiens.outil; //Contient true, propriété du prototype « homoHabilis »
homoSapiens.stature; //Contient 'debout', propriété du prototype « homoErectus »
homoSapiens.ere; //Contient '-40000 ans', propriété de l'objet « homoSapiens »
```

Grâce à « **`Object.create()`** », on arrive à créer une chaîne de prototypes en partant d'un prototype initial. À la fin, chaque objet dispose de ses propres propriétés et, pour celles qu'il ne possède pas, hérite de celles du prototype suivant dans la chaîne de prototypes.

3. Objets Natifs

1. **Retour sur les types de données**
2. **Objets fondamentaux et coercition**

3.1 Retour sur les types de données

Comme indiqué précédemment, les types données en JavaScript ne sont pas tout à fait conformes à ce qui avait été présenté. En réalité, d'après la norme ECMAScript 5 à laquelle se conforme JavaScript, nous pouvons identifier 6 types de données :

1. **5** types de données primitifs :
    2. Le type `null` pour la valeur nulle.
    3. Le type `undefined` pour la valeur non définie.
    4. Le type `Number` pour les nombres (dont font partie les valeurs `NaN` et `Infinity`).
    5. Le type `String` pour les chaînes de caractère.
    6. Le type `Boolean` pour les booléens.
1. et Le type `Object` (dont font partie le type `Array` et le type `Function`).

Première surprise, les valeurs `NaN` et `Infinity` font partie du type `Number`.

En effet, on peut constater concernant la valeur `NaN` que si :

```
var nombre = 2 * 'un texte au hasard'; //On effectue une opération qui va nous donner la valeur NaN.

nombre; //Contient la valeur NaN.

nombre = (nombre - 1) / 100; //On peut utiliser la valeur NaN contenue dans la variable nombre pour faire une nouvelle opération sans faire planter le moteur.

nombre; //Contient NaN.

/*
On peut donc en déduire que le moteur traite la valeur NaN comme si c'était un nombre avec lequel on peut réaliser n'importe quelle opération mathématique. Évidemment, toutes ces opérations donnent toujours en sortie la valeur NaN.
*/
```

Et par ailleurs, on peut constater concernant la valeur `Infinity` que si :

```
var nombre = 17 / 0; //On effectue une opération qui va nous donner la valeur Infinity.

nombre; //Contient la valeur Infinity.

nombre = (nombre - 1) / 100; //On peut utiliser la valeur Infinity contenue dans la variable nombre pour faire une nouvelle opération sans faire planter le moteur.

nombre; //Contient Infinity.

/*
On peut donc en déduire que le moteur traite la valeur Infinity comme si c'était un nombre avec lequel on peut réaliser n'importe quelle opération mathématique. Évidemment, toutes ces opérations donnent toujours en sortie la valeur Infinity.
*/
```

Deuxième surprise, on considère que les tableaux (`Array`) et, surtout, les fonctions (`Function`) sont des objets.

Concernant les tableaux, le moteur JavaScript met à la disposition du programmeur une **fonction constructeur** `Array` (cf **1.3. Fonctions constructeur**). Le moteur met également à disposition du programmeur une **forme raccourcie** pour l'appel de cette fonction constructeur sous la forme de l'**utilisation des crochets** `[]`. On peut donc écrire :

```
var objetTableau = ['un', 'deux', 'trois']; //utilisation de la syntaxe raccourcie avec [] pour déclarer un objet tableau.

objetTableau; //Contient ['un', 'deux', 'trois'].

objetTableau = new Array('un', 'deux', 'trois'); //utilisation de la fonction constructeur Array pour déclarer un objet tableau identique au précédent.

objetTableau; //Contient ['un', 'deux', 'trois'].

//Les deux syntaxes permettent d'obtenir la même chose, un objet tableau.
```

La fonction constructeur `Array` prend en argument l'ensemble des éléments qu'on souhaite organiser sous la forme d'un tableau. Elle nous permet d'obtenir un objet tableau qui contient des **propriétés** relatives aux valeurs qu'il contient et des **méthodes** pour le manipuler. Par exemple :

```
objetTableau; //Contient ['un', 'deux', 'trois'], donc 3 éléments.

objetTableau.length; //Contient 3, la propriété .length de l'objet tableau contient le nombre d'éléments contenus dans le tableau.

objetTableau.push( 'quatre', 'cinq'); //Retourne 5, la méthode .push() de l'objet tableau prend en argument n arguments qui devront être ajouté au tableau et retourne le nombre total d'éléments contenus dans le tableau.

objetTableau; //Contient désormais ['un', 'deux', 'trois', 'quatre', 'cinq'], soit 5 éléments.
```

Les objets tableaux obtenus à l'aide de la fonction constructeur `Array` ou de la syntaxe simplifiée à base de crochets `[]` on pour prototype un objet fourni par la moteur JavaScript. La méthode `.push()`, par exemple, appartient à ce prototype. L'ensemble des méthodes appartenant au prototype (cf **2. Le Prototypage**) des objets créés avec la fonction constructeur `Array` est consultable sur la documentation officielle de JavaScript **ici**.

Concernant les fonctions, en JavaScript, chaque fonction est un objet créé à l'aide de la fonction constructeur `Function`. Cela signifie que les expressions de fonction créées l'aide du mot-clé `function` sont également des objets qui ont la particularité de pouvoir être exécutés. La fonction constructeur `Function` prend en arguments les arguments de la fonction à créer ainsi que les instructions à exécuter sous la forme de chaînes de caractères.

Par exemple :

```
var objetFonction = function(nom, prenom){
  alert('Bonjour ' + prenom + ' ' + nom);
}; //déclaration de la fonction, création de l'objet fonction

objetFonction('Jean', 'Tille'); //Exécution de la fonction/objet, affiche une boite de dialogue contenant le texte : 'Bonjour Jean Tille'.

objetFonction = new Function('nom','prenom', 'alert(\'Bonjour \' + prenom + \' \' + nom);'); //déclaration de la fonction, création de l'objet fonction.

objetFonction('Jean', 'Tille'); //Exécution de la fonction/objet, affiche une boite de dialogue contenant le texte : 'Bonjour Jean Tille'.

//Les deux syntaxes permettent d'obtenir la même chose, un objet fonction.
```

N'importe quelle fonction, qu'elle soit créée par le programmeur, fournie par le moteur JavaScript, que ce soit une fonction « basique » ou une fonction constructeur est à la fois une fonction et à la fois un objet. Elle dispose donc de propriétés et de méthodes. Les fonctions ont toutes, par exemple, une propriété `.prototype` qui contient le prototype de l'objet fonction. Par défaut, celui-ci est vide. On peut, cependant, le redéfinir lorsqu'on écrit une fonction constructeur de telle sorte qu'on puisse obtenir un objet qui hérite du prototype défini (cf **2.3.1 Le prototypage avec une fonction constructeur**). On dispose également d'autres propriétés et méthodes comme par exemple :

```
objetFonction.length; //Contient 2, la propriété .length contient un nombre qui représente le nombre d'arguments attendus par la fonction.
```

Les objets fonction héritent également des caractéristiques du prototype de leur fonction constructeur `Function` consultable sur la documentation officielle de JavaScript **ici**. En JavaScript, lorsqu'on déclare une fonction constructeur, on parle de :

1. **Propriétés et méthodes** de la fonction quand on fait référence aux propriétés et aux méthodes de l'objet fonction constructeur considéré.
1. **Propriétés et méthodes d'instance** quand on fait référence aux propriétés et aux méthodes de l'objet retourné par la fonction constructeur considérée.

Les fonctions constructeur / objets `Array` et `Function` font partie des **objets fondamentaux** du langage.

3.3 Objets fondamentaux et coercition

En JavaScript, on dispose de façon native de plusieurs objets fondamentaux qui sont globaux, c'est-à-dire accessibles quel que soit le contexte, et qui offrent aux programmeurs un ensemble d'outils pour les aider à atteindre leurs objectifs. On peut distinguer, parmi ces objets fondamentaux, les objets d'« encapsulation » (en anglais, « wrapper objects ») dont on définira les caractéristiques plus tard et des objets utilitaires plus classiques.

Ces objets sont :

1. objets d'« encapsulation » standards :

    2. `Number` objet (et fonction constructeur) contenant des propriétés et des méthodes pour travailler avec des nombres (**documentation**).

3. `String` objet (et fonction constructeur) contenant des propriétés et des méthodes pour travailler avec des chaînes de caractères (**documentation**).
4. `Boolean` objet (et fonction constructeur) contenant des propriétés et des méthodes pour travailler avec des booléens (**documentation**).

1. objets utilitaires standards :

2. `Array` objet (et fonction constructeur) contenant des propriétés et des méthodes pour travailler avec des tableaux (**documentation**).
3. `Object` objet (et fonction constructeur) contenant des propriétés et des méthodes pour travailler avec des objets (**documentation**).
4. `Function` objet (et fonction constructeur) contenant des propriétés et des méthodes pour travailler avec des fonctions (**documentation**).
5. `RegExp` objet (et fonction constructeur) contenant des propriétés et des méthodes pour travailler avec des expressions régulières (**documentation**).
6. `Date` objet (et fonction constructeur) contenant des propriétés et des méthodes pour travailler avec des chaînes de caractère représentant une date (**documentation**).
7. `JSON` objet (instance) contenant des propriétés et des méthodes pour travailler avec des chaînes de caractère représentant un JSON (**documentation**).
8. `Math` objet (instance) contenant des propriétés et des méthodes pour effectuer des opérations mathématiques complexes (**documentation**).

Si on consultait, par exemple, la documentation officielle de JavaScript sur le MDN **ici**, on verrait beaucoup plus d'objets fondamentaux. Attention, les objets fondamentaux présentés dans cette partie sont les objets fondamentaux **standard**, à savoir disponibles sur tous les navigateurs Internet du marché. Certains objets présentés par certaines documentations sont non-standard, c'est-à-dire qu'ils ne sont disponibles que sur certains navigateurs du marché (dans le cas du MDN, ce sont les objets standards et les objets disponibles sur Mozilla Firefox).

Qu'entend-on par objets d'« encapsulation » ?

Lorsqu'on utilise un type de données primitif comme un nombre, un booléen ou une chaîne de caractères, le moteur JavaScript peut traiter ce type de données comme un objet en l'encapsulant automatiquement dans un objet qui sera détruit par la suite. Cette opération, transparente pour le programmeur, lui fournit, chaque fois qu'il en a besoin, une « boîte à outil » pour travailler avec ses données. Par exemple :

```
var chaine;

chaine = 'Ceci est une chaîne de caractère';
//Est équivalent à :
/*
chaine = new String('Ceci est une chaîne de caractère');
*/

//Ici, on utilise la chaine de caractère comme si c'était un objet :
chaine.length; //Propriété, Contient 34, soit le nombre de caractère de la chaîne de caractères.
chaine.substr(0, 3); //Méthode, Retourne les 4 premiers caractères : 'Ceci'.
//Cela fonctionne grâce à la « magie » du moteur JavaScript qui a encapsulé au besoin la chaîne de caractère dans un objet de type String.

chaine; //Contient toujours 'Ceci est une chaîne de caractère'.
```

Comme on peut le voir dans cet exemple, lorsqu'on créé une chaîne de caractère, on peut, à tout moment, faire à appel à une propriété ou une méthode qui serait fournie par l'objet d'« encapsulation » correspondant (`String` dans ce cas). Le moteur JavaScript se charge d'instancier l'objet pour appeler la bonne propriété ou la bonne méthode. Une fois la propriété ou méthode utilisée, le moteur détruit l'objet d'« encapsulation » pour que la variable retrouve son état initial.

On appelle ce comportement la **coercition**. Le moteur JavaScript contraint momentanément un type de données (ici une chaîne de caractère) à devenir un autre type de données (une instance de `String`, un objet).

Il n'est pas recommandé d'utiliser les objets d'« encapsulation » directement. Cela peut donner lieu à des erreurs algorithmiques. Par exemple :

```
var a = 'message'; //On déclare une chaîne de caractère
var b = 'message'; //On déclare la même chaîne de caractère
var c = new String('message'); //On instancie un objet d'encapsulation contenant la même chaîne de caractère

a == b; //true, a et b contiennent une valeur qui représente la même chaîne de caractère.
a == c; //true, a et c contiennent une valeur qui représente la même chaîne de caractère.

a === b; //true, a et b contiennent une valeur qui représente la même chaîne de caractère et qui est du même type.
a === c; //false, a et c contiennent une valeur qui représente la même chaîne de caractère mais qui n'est pas du même type !

typeof a; //'string', la variable a contient une valeur du type primitif string.
typeof b; //'string', la variable b contient une valeur du type primitif string.
typeof c; //'object', la variable c contient une valeur du type object !
```

Les objets fondamentaux du langage JavaScript sont des objets disponibles dans n'importe quel contexte de programmation offert par un moteur JavaScript. Dans le cadre d'un navigateur Internet, on dispose, en plus, d'objets particuliers standardisés sous l'acronyme de DOM (« Document Objet Model »)

4. Modèle Objet du Document

1. **Une représentation structurée de document**
2. **Une interface de programmation**
3. **Utilisation du DOM**

4.1 Une représentation structurée de document

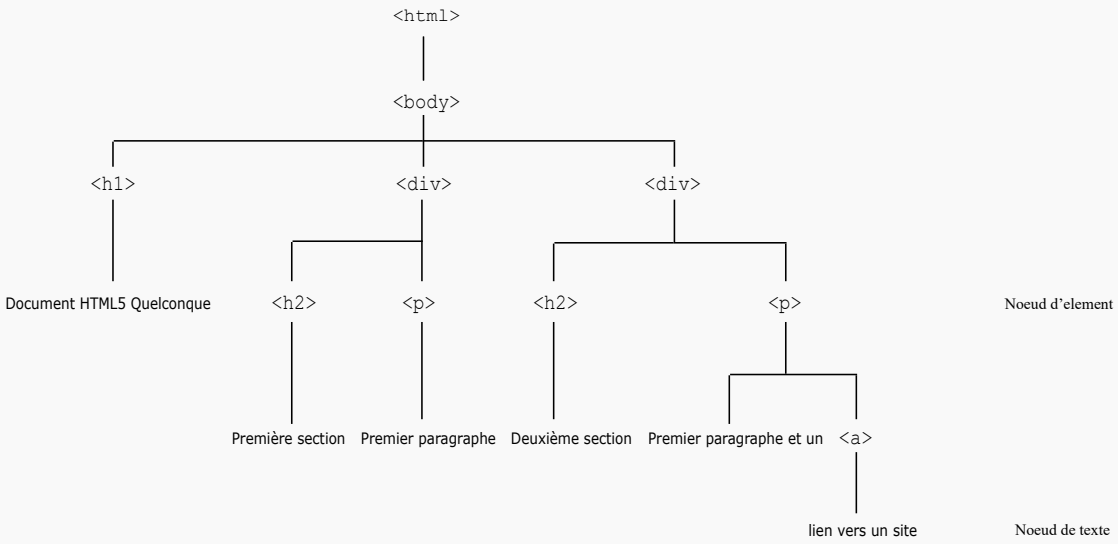
Le DOM (**D**ocument **O**bject **M**odel en anglais ou Modèle Objet du Document en français) est une représentation d'un document HTML, XML ou SVG sous la forme d'un **arbre**. Cette représentation, **indépendante de tout langage de programmation**, est normalisée par le W3C **ici**. Dans cet arbre on distinguera différents types d'éléments. Ces éléments sont appelés « **noeuds** » (*nodes* en anglais).

On peut matérialiser de cette représentation de la façon suivante :

Exemple de Document HTML

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta
charset="utf-8">
    <title>Document
HTML5
quelconque</title>
  </head>
  <body>
    <h1>Document
HTML5
quelconque</h1>
    <div>
      <h2>Première
section</h2>
      <p>Premier
paragraphe</p>
    </div>
    <div>
      <h2>Deuxième
section</h2>
      <p>Premier
paragraphe et un <a
href="#">lien vers
un site</a></p>
    </div>
  </body>
</html>
```

Représentation du DOM correspondant



Comme on peut le constater sur l'exemple qui précède, les langages de description tels que le HTML, se prêtent particulièrement bien à une représentation sous la forme d'un arbre. Dans cet arbre, les balises HTML sont appelées « **noeuds d'éléments** » et les contenus textuels sont appelés « **noeuds de texte** »

Sur un navigateur Internet, le DOM est implémenté sous la forme d'une arborescence d'**objets**. En d'autre termes, à chaque noeud correspond un objet. Le programmeur peut, en JavaScript, accéder à ce modèle objet pour consulter ou modifier les informations relatives au navigateur Internet ou au document Web considéré. Le DOM est donc également une **interface de programmation**.

4.2 Une interface de programmation

L'affichage d'un document Web se déroule en plusieurs étapes :

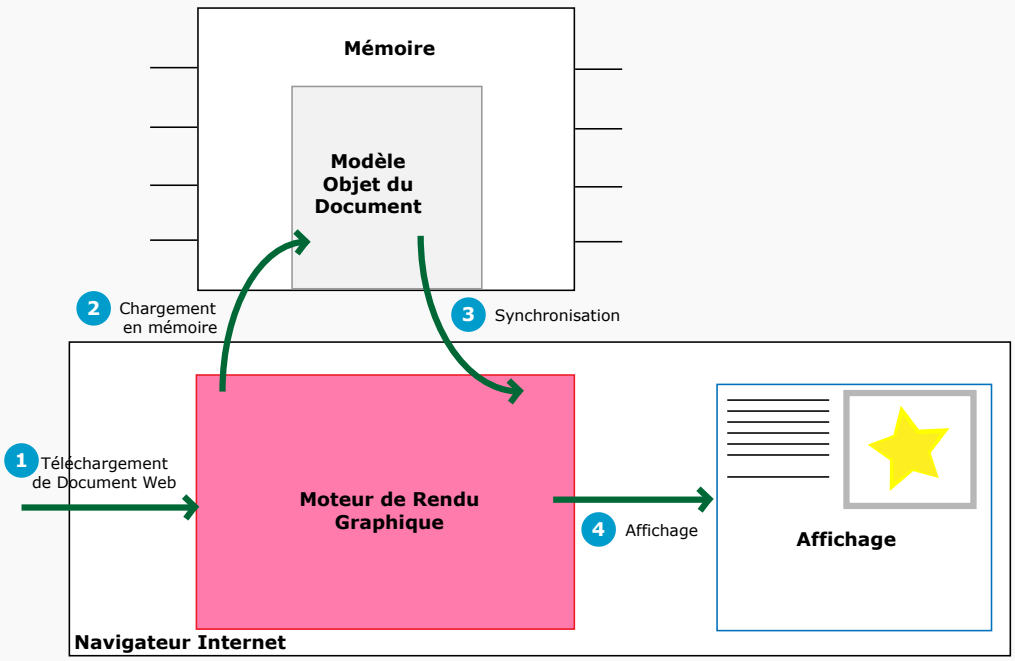
1. Le navigateur Internet télécharge un document Web obtenu dans une réponse HTTP.
2. Le moteur de rendu graphique du navigateur Internet lit et charge en mémoire le document Web sous la forme d'un modèle objet : le DOM.
3. Le moteur de rendu graphique affiche une représentation graphique du contenu de la mémoire. Cette représentation graphique est synchronisée avec le contenu de la mémoire.

Le moteur de rendu graphique d'un navigateur Internet est la « brique » technique des navigateurs Internet qui est en charge de l'affichage d'un document Web.

C'est le moteur de rendu graphique qui créé le DOM en mémoire. Il met à la disposition du moteur JavaScript une interface de programmation pour que le programmeur puisse consulter ou modifier le DOM. La représentation

graphique produite par le moteur de rendu graphique étant synchronisée avec le DOM, si le programmeur modifie le DOM, la représentation graphique (l'affichage !) est modifié en conséquence.

On pourrait représenter schématiquement ce mécanisme comme suit :



Ce n'est pas le fichier original ou l'affichage qui est modifié par le programmeur mais le modèle objet en mémoire !

Le moteur JavScript met à disposition du programmeur un objet `window` qui permet d'accéder à toutes les caractéristiques du navigateur Internet et au DOM.

4.2 Utilisation du DOM

Le DOM est donc un modèle objet accessible, en JavaScript, à tout moment via l'objet `window`.

L'objet `window` est un objet de type `Window`. Ce type d'objet est normalisé par le W3C, cela signifie qu'il est censé contenir les même propriétés et méthodes quel que soit le navigateur Internet considéré. On peut consulter la liste des propriétés et méthodes de cet objet **ici**.

On peut proposer les exemples suivant d'utilisation de méthodes de `window` :

```
//Exécution de la méthode .alert() de window :
window.alert('Ceci est une boite de dialogue.');

//Exécution de la méthode .prompt() de window :
var saisie = window.prompt('Ceci est une boite de dialogue avec un champ de saisie.');

//Exécution de la méthode .setTimeout() de window :
window.setTimeout(function(){
  window.alert('Ce message s\'affiche après 1 seconde.');
},1000);
```

Précédemment, on vous avait peut-être expliqué que `.alert()`, `.prompt()`, `.setTimeout()` étaient des « *fonctions* ». En réalité, ce sont des **méthodes** de l'objet `window`.

Si on pouvait faire appel à ces méthodes sans les préfixer par « `window.` », c'est parce que, dans chaque fenêtre ou onglet du navigateur Internet, l'objet `window` est considéré comme l'**objet de plus haut niveau** (en anglais, « top level object »), un sorte d'*objet par défaut*. On notera que chaque fenêtre ou onglet du navigateur Internet propose son propre objet `window` qui est **différent** de celui des autres fenêtres ou onglets du navigateur Internet.

L'objet `window` possède également des propriétés dont certaines contiennent des sous objets.

Par exemple `window.console`. La propriété `.console` de `window` contient un objet de type `Console` (non standard, défini par le MDN **ici**). Ce sous objet de `window` propose un ensemble de méthode pour agir sur la console de deboggage du navigateur Internet. Par exemple :

```
//Ici on exécute la méthode .log() du sous objet console de l'objet window.
window.console.log('Ceci est un message qui sera affiché dans la console');
//Cette exécution produira l'affichage du texte en argument de la méthode .log() dans la console.
```

Un autre exemple (le plus connu) est `window.document`. Cette propriété de `window` contient un objet de type `Document` (standardisé par le W3C, défini par le MDN **ici**). Cet objet contient l'ensemble des informations relatives au document affiché. Par exemple :

```
//Ici on accède à la propriété characterSet de l'objet document dans l'objet window.
window.document.characterSet;//Contient la chaine de caractère 'UTF-8' qui représente l'encodage des caractères du document.
```

Autre exemple :

Pour ce document HTML

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
  </head>
  <body>
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2>Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
  </body>
</html>
```

Instructions JavaScript (à exécuter juste avant le chevron *</body>* fermant)

```
window.document.children; //Contient un tableau avec le premier niveau de noeuds « d'enfants » du document.
window.document.children[0]; //Contient un objet de type HTMLHtmlElement (standardisé par le W3C, défini par le MDN ici) qui correspond à la balise <html>
window.document.children[0].children; //Contient un tableau avec le deuxième niveau de noeuds « d'enfants » du document.
window.document.children[0].children[0]; //Contient un objet de type HTMLHeadElement (standardisé par le W3C, défini par le MDN ici) qui correspond à la balise <head>
window.document.children[0].children[1]; //Contient un objet de type HTMLBodyElement (standardisé par le W3C, défini par le MDN ici) qui correspond à la balise <body>
window.document.children[0].children[1].children; //Contient un tableau avec le troisième niveau de noeuds « d'enfants » du document.
window.document.children[0].children[1].children[0]; //Contient un objet de type HTMLHeadingElement (standardisé par le W3C, défini par le MDN ici) qui correspond à la balise <h1>
window.document.children[0].children[1].children[1]; //Contient un objet de type HTMLDivElement (standardisé par le W3C, défini par le MDN ici) qui correspond à la première balise <div>
window.document.children[0].children[1].children[2]; //Contient un objet de type HTMLDivElement (standardisé par le W3C, défini par le MDN ici) qui correspond à la deuxième balise <div>
window.document.children[0].children[1].children[2].children[1].children[0]; //Contient un objet de type HTMLAnchorElement (standardisé par le W3C, défini par le MDN ici) qui correspond à la balise <a>
window.document.children[0].children[1].children[2].children[1].children[0].href; //Cette propriété contient '#', c'est à dire la valeur de l'attribut href de la balise.
window.document.children[0].children[1].children[2].children[1].children[0].href = 'http://www.virtuoworks.com'; //Ici, on affecte une nouvelle valeur à la propriété href de l'objet correspondant à la balise <a>.

/*
Tous les sous objets de l'objet document qui correspondent à des balises HTML correspondent à un type d'objet standardisé par le W3C. On peut consulter cette liste d'objets sur le MDN ici. Ils ont tous des propriétés et des méthodes qui leurs sont spécifiques et héritent d'un objet de type HTMLElement à partir du quel ils partagent des propriétés et des méthodes communes. Cet objet est défini ici.
*/
```

A partir de l'objet `document` on peut accéder aux sous-objets qui correspondent aux éléments HTML du document original. On peut y accéder directement comme dans l'exemple qui précède ou en utilisant des méthodes de l'objet `document`. Par exemple :

Pour ce document HTML

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
  </head>
  <body>
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2>Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
  </body>
</html>
```

Instructions JavaScript (à exécuter juste avant le chevron *</body>* fermant)

```
var toutesLesDiv = window.document.getElementsByTagName('div'); //la méthode .getElementsByTagName() retourne un tableau contenant une référence vers chacun des objet du DOM correspondant à la chaîne de caractère en argument (le tag 'div').

toutesLesDiv; //Contient [HTMLDivElement,HTMLDivElement], un tableau avec 2 références vers les 2 objets correspondant aux 2 balises <div> du document.

toutesLesDiv[1].className = 'une_classe_css'; //On peut modifier le contenu de la propriété className de l'objet qui correspond à la deuxième balise <div>. L'affichage sera mis à jour avec une nouvelle valeur pour l'attribut HTML class de la deuxième div.
```

Les différents types d'objets du DOM sont documentés sur le MDN **ici**.

Tous les codes qui précèdent doivent être exécutés juste avant le chevron *</body>* fermant pour que le moteur de rendu graphique ait le temps de lire et de charger le document HTML en mémoire. Si on avait essayé d'exécuter les

instructions précédentes plus tôt dans le document Web, cela n'aurait pas fonctionné tout simplement parce que les objets n'auraient pas été trouvés en mémoire.

JavaScript est, cependant, un langage événementiel qui va nous permettre d'exécuter des instructions, non pas en fonction du flux de lecture du document HTML (de haut en bas), mais en réaction à des comportements du navigateur Internet ou des actions de l'utilisateur. Cela va nous permettre de choisir quand déclencher des instructions et, en particulier, des instructions qui entraînent des modifications du DOM après que celui ci ait été chargé en mémoire.

5. Gestion des événements

1. **Gestion des événements depuis le document Web**
2. **Gestion des événements depuis le DOM**
3. **Gestion des événements et Objet d'événement**

5.1 Gestion des événements depuis le document Web

Les balises HTML peuvent porter des attributs. Ces attributs peuvent être attributs d'identification (*id*), des attributs de style (*style*, *class*), des attributs d'accessibilité (*rel*, *alt*), ... Nous allons, ici, nous intéresser plus particulièrement aux **attributs d'événement** qui ont vocation à être utilisés avec du JavaScript. Ils proposent au programmeur d'exécuter du code JavaScript en réaction à un comportement du navigateur Internet ou un comportement de l'utilisateur.

En HTML les attributs se présentent généralement sous la forme *attribut="valeur"*. Il en va de même pour les attributs d'événements avec la particularité d'avoir pour valeur 1 ou plusieurs instructions JavaScript. On peut illustrer cela comme suit avec l'attribut d'événement `onclick` qui permet d'exécuter du code en réaction à un clic de l'utilisateur sur la balise portant l'attribut :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 avec des balises portant des attributs</title>
  </head>
  <body>
    <h1 class="titre" >Document HTML5 avec des balises portant des attributs</h1>
    <div id="contenu" onclick="window.alert('Ce message s\'affichera uniquement si l'utilisateur clique sur la zone couverte par
cette div.');" >
      <h2 title="section" >section</h2>
      <p style="color:#FFD800;font-weight:bold;">paragraphe</p>
    </div>
  </body>
</html>
```

Sur cet exemple, si l'utilisateur clique sur la zone couverte par la *div*, le code contenu dans l'attribut d'événement `onclick` sera exécuté **par** le navigateur Internet et, dans ce cas, produira l'affichage d'une boite de dialogue.

L'attribut d'événement `onclick` fait partie des événements qui se produisent en réaction à une action de l'utilisateur. On peut en citer d'autres comme `onmouseover` qui se déclenche quand la souris survole l'élément, `onmouseout` qui se déclenche quand la souris ne survole plus l'élément, `ondblclick` qui se déclenche quand le bouton de la souris est pressé 2 fois sur l'élément, ...

On peut maintenant proposer un attribut d'événement qui correspond à un comportement du navigateur Internet et qui est très fréquemment employé, l'attribut d'événement `onload`. Il s'utilise comme sur l'exemple suivant :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 avec des balises portant des attributs</title>
  </head>
  <body onload="window.alert('Ce message s\'affichera uniquement le navigateur Internet à fini de charger le document Web en
mémoire.');">
    <h1 class="titre" >Document HTML5 avec des balises portant des attributs</h1>
    <div id="contenu" >
      <h2 title="section" >section</h2>
      <p style="color:#FFD800;font-weight:bold;">paragraphe</p>
    </div>
  </body>
</html>
```

Sur cet exemple, l'instruction JavaScript dans l'attribut `onload` sera exécutée **par** le navigateur Internet quand le document, les feuilles de style CSS et les scripts JavaScript associés sont entièrement chargés en mémoire. Cet attribut d'événement est très fréquemment utilisé pour déclencher des programmes qui vont agir le DOM. En effet, comme on l'a dit précédemment, on ne peut pas modifier le DOM tant que le navigateur Internet ne l'a pas entièrement chargé en mémoire.

L'attribut d'événement `onload` ne peut être utilisé **que sur certaines balises** comme `<body>`, `<img>`, `<audio>`, `<video>`, ...

Généralement, on écrit peu de code dans un attribut d'événement. On créé une fonction dans l'entête du document et on appelle cette fonction dans l'attribut d'événement. De cette façon, on sépare au maximum le code JavaScript du code HTML. Exemple :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 avec des balises portant des attributs</title>
    <script>
      var afficheMessage = function(){
        window.alert('Ce message s\'affichera uniquement le navigateur Internet à fini de charger le document Web en mémoire.');
      };
    </script>
  </head>
  <body onload="afficheMessage();">
    <h1 class="titre" >Document HTML5 avec des balises portant des attributs</h1>
    <div id="contenu" >
      <h2 title="section" >section</h2>
      <p style="color:#FFD800;font-weight:bold;">paragraphe</p>
    </div>
  </body>
</html>
```

Le règles à suivre pour utiliser les attributs d'événement sont les suivantes :

1. On ne **peut pas** avoir plusieurs fois le même attribut d'événement sur la même balise. Il est, par exemple, **interdit** d'écrire : `<div onclick="1;" onclick="2;">`

1. On **peut** avoir plusieurs attributs d'événement différents sur la même balise. On peut, par exemple, écrire : `<div onclick="1;" onmouseover="2;" onmouseout="2;">`

1. **On ne doit pas sauter de lignes** lorsqu'on écrit des instructions dans un attributs d'événement. Les instructions doivent être écrites **à la suite**.

1. On doit **vérifier** qu'une balise supporte bien l'attribut d'événement qu'on souhaite utiliser dessus. Généralement, une recherche sur Internet associant le nom de la balise avec l'expression « attribut d'événement » (en anglais, « event attribute ») permet de trouver facilement la liste des attributs d'événement supportés par la balise.

Bien que l'utilisation d'attributs d'événements sur les documents Web reste encore très fréquente, **elle n'est pas recommandée** comme on peut le constater sur le MDN **ici**. On préférera gérer les événements directement en JavaScript en utilisant les possibilités offertes par le DOM.

5.2 Gestion des événements depuis le DOM

L'ensemble des données relatives à un document Web sont chargées en mémoire par le navigateur Internet et ceci inclus l'ensemble des balises et l'ensemble, leurs attributs et leurs valeurs respectives. En d'autres termes, on peut retrouver dans le DOM tous les attributs des balises d'un document Web sous la forme de propriétés. On peut modifier ces propriétés.

Parmi les attributs de balises, on sait qu'on peut avoir des **attributs d'événements**. Et, on sait que les attributs d'événement peuvent contenir du code JavaScript. En JavaScript et dans le DOM, les objets, qui correspondent à des balises, ont des propriétés qui correspondent aux attributs d'événements et qui sont, en réalité, des **méthodes**.

On aura, par exemple :

Gestion d'un événement `onload` à partir du document HTML

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
  </head>
  <body onload="window.alert('Ce message s\'affichera uniquement le navigateur Internet à fini de charger le document Web en mémoire.');">
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2>Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
  </body>
</html>
```

Gestion du même événement à partir du DOM

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
    <script>
      window.onload = function(){
        window.alert('Ce message s\'affichera uniquement le navigateur Internet à fini de charger le document Web en mémoire.');
      };
    </script>
  </head>
  <body>
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2>Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
  </body>
</html>
```

Pour pouvoir gérer les événements qui pourraient survenir sur un élément HTML, il faut **impérativement** que le DOM ait été entièrement **chargé en mémoire**. On a donc 2 possibilités :

1. Soit on accède au DOM juste avant la balise `<body>` fermante pour définir la fonction relative à l'événement. Par exemple :

Gestion d'un événement `onclick` à partir du document HTML

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
  </head>
  <body>
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2 onclick="window.alert('Ce message s\'affichera uniquement si on clique sur ce titre.');">Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
  </body>
</html>
```

Gestion du même événement à partir du DOM (1ère possibilité)

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
  </head>
  <body>
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2 id="titrePremiereSection">Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
    <script>
      var objetHTMLh2 = window.document.getElementById('titrePremiereSection'); //Ici on utilise la méthode .getElementById() de l'objet document pour récupérer une référence à l'objet du DOM qui correspond à la balise h2 identifiée dans le document HTML chargé.
      objetHTMLh2.onclick = function(){ //Ici on définit la méthode .onclick() qui correspond au code qui devra être exécuté en réaction à l'événement onclick.
        window.alert('Ce message s\'affichera uniquement si on clique sur ce titre.');
      };
    </script>
  </body>
</html>
```

1. Soit on accède au DOM dans l'entête du document dans une fonction définie dans la propriété `.onload` de l'objet `window` et on définit la fonction relative à l'événement. Par exemple :

Gestion d'un événement `onclick` à partir du document HTML

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
  </head>
  <body>
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2 onclick="window.alert('Ce message s\'affichera uniquement si on clique sur ce titre.');">Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
  </body>
</html>
```

Gestion du même événement à partir du DOM (2ème possibilité)

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
    <script>
      window.onload = function(){ //Ici on définit la méthode .onload() qui correspond au code qui devra être exécuté en réaction à l'événement onload c'est à dire à la fin du chargement du document HTML en mémoire.
        var objetHTMLh2 = window.document.getElementById('titrePremiereSection'); //Ici on utilise la méthode .getElementById() de l'objet document pour récupérer une référence à l'objet du DOM qui correspond à la balise h2 identifiée dans le document HTML chargé.
        objetHTMLh2.onclick = function(){ //Ici on définit la méthode .onclick() qui correspond au code qui devra être exécuté en réaction à l'événement onclick.
          window.alert('Ce message s\'affichera uniquement si on clique sur ce titre.');
        };
      };
    </script>
  </head>
  <body>
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2 id="titrePremiereSection">Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
  </body>
</html>
```

Les événements sont donc gérés dans le DOM par des méthodes des objets concernés. Si ce sont des méthodes, cela signifie qu'on peut utiliser le mot-clé `this` à l'intérieur de ces méthodes pour faire référence à l'objet concerné par l'événement. On pourra donc écrire :

Utilisation du mot-clé `this` dans un événement géré avec le DOM

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
    <script>
      window.onload = function(){
        this; //fait référence à l'objet window.
        var objetHTMLh2 = window.document.getElementById('titrePremiereSection');
        objetHTMLh2.onclick = function(){
          this; fait référence à l'objet référencé dans objetHTMLh2 à savoir l'objet correspondant à la balise <h2> identifiée par 'titrePremiereSection'.
          this.innerHTML = 'Nouveau Titre'; //Affecte une nouvelle valeur à la propriété .innerHTML de this, c'est à dire l'objet référencé dans objetHTMLh2.
        };
      };
    </script>
  </head>
  <body>
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2 id="titrePremiereSection">Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
  </body>
</html>
```

Utilisation du mot-clé `this` dans un événement géré à partir du document HTML

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
  </head>
  <body>
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2 onclick="this.innerHTML = 'Nouveau Titre';">Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
  </body>
</html>
```

Les méthodes qu'on définit à partir du DOM pour gérer un événement sont appelées « **gestionnaires d'événements** » en français et « **event handlers** » ou « **event listeners** » en anglais. On remarque qu'on ne peut définir qu'un seul gestionnaire d'événement par événement. Si on veut définir plusieurs gestionnaires d'événements pour un seul événement, on peut utiliser la méthode `.addEventListener()` du DOM qui est disponible sur chaque objet correspondant à un élément HTML du document Web.

La méthode `.addEventListener()` prend en argument le *nom d'un événement* et *une fonction*. Cette fonction sera ajoutée à la liste des gestionnaires d'événements de l'événement fourni en argument. Exemple d'utilisation :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
    <script>
      window.addEventListener('load', function(){
        //Ceci est une première fonction qui sera exécutée au chargement du document : gestionnaire d'événement 1.
        window.alert('un');
      });
      window.addEventListener('load', function(){
        //Ceci est une deuxième fonction qui sera exécutée au chargement du document : gestionnaire d'événement 2.
        window.alert('deux');
      });
      window.addEventListener('load', function(){
        //Ceci est une troisième fonction qui sera exécutée au chargement du document : gestionnaire d'événement 3.
        window.alert('deux');
        this; //fait toujours référence à l'objet window dans lequel sera exécuté la fonction.
      });
    </script>
  </head>
  <body>
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2 id="titrePremiereSection">Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
  </body>
</html>
```

La méthode `.addEventListener()` permet également de définir des gestionnaires d'événement pour des événements qui ne sont pas forcément accessible directement sous la forme de propriétés des objets du DOM. Comme, par exemple, l'événement `DOMContentLoaded` (défini **ici** dans le MDN).

L'événement `DOMContentLoaded` correspond à la fin du chargement du DOM en mémoire par le navigateur Internet là où l'événement `onload` (ou *load*) correspond à la fin du chargement du document Web et de toutes les ressources associées (feuilles CSS, images, ...). Cet événement se déclenche donc plus en amont lors du chargement d'un document Web et permet de commencer l'exécution de scripts en avance de phase. Son utilisation est **recommandée** par rapport à l'événement *load*.

Exemple d'utilisation :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
    <script>
      window.addEventListener('DOMContentLoaded', function(){
        //Ceci est une fonction qui sera exécutée au chargement du DOM du document.
        window.alert('S\'affiche avant même le chargement des feuilles CSS, images, etc...');
      });
    </script>
  </head>
  <body>
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2 id="titrePremiereSection">Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
  </body>
</html>
```

La liste des événements qu'on peut utiliser à partir du DOM en utilisant la méthode `.addEventListener()` est disponible **ici** sur le MDN. Sur cette page, les événements marqués comme appartenant au DOM L3 sont les événements qui concernent le navigateur Internet, la souris, le clavier ou l'écran; les événement HTML5 sont les événement qui concernent les éléments HTML, les événements SVG concernent les éléments SVG, ...

Dans tous les cas, les gestionnaires d'événements (qui ne sont que des fonctions, il ne faut pas l'oublier) sont toujours exécutés par le moteur JavaScript en réaction à un événement. Les événements qui peuvent survenir ont tous des caractéristiques qui leurs sont propre. Par exemple, pour un clic de souris, l'événement à entre autres caractéristiques les coordonnées du pointeur de souris. Ces caractéristiques sont transmises par le navigateur Internet au gestionnaires d'événements à l'aide d'un objet : l'« objet d'événement ».

5.3 Gestion des événements et Objet d'événement

Lorsque le navigateur Internet exécute une fonction gestionnaire d'événement, **il fournit** en paramètre un objet qui contient toutes les caractéristiques de l'événement. On appelle cette objet un **Objet d'Evènement** (en anglais, « Event Object »).

Le programmeur averti peut prévoir un argument pour accueillir l'objet d'événement fourni par le navigateur Internet lorsqu'il écrit une fonction gestionnaire d'événement. Il peut ainsi utiliser dans la fonction gestionnaire d'événement les caractéristiques associées à l'événement.

Exemple de gestionnaire d'événement utilisant l'objet d'événement :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
    <script>
      window.addEventListener('DOMContentLoaded', function(){
        var elementHTMLh2 = window.document.getElementById('titrePremiereSection');
        elementHTMLh2.addEventListener('mouseover', function(argumentFourniParLeNavigateur){
          argumentFourniParLeNavigateur; //EventObject, contient un objet d'événement fourni par le navigateur Internet.
        });
      });
    </script>
  </head>
  <body>
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2 id="titrePremiereSection">Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
  </body>
</html>
```

Dans cet exemple, si l'utilisateur survole le titre `<h2>` identifié, le navigateur Internet exécute la fonction gestionnaire d'événement et lui fourni en paramètre un objet d'événement. Le programmeur peut donc prévoir d'utiliser cet objet d'événement à l'intérieur de se fonction gestionnaire d'événement.

Les objets d'événements fournis par le navigateur Internet sont des objets de type `Event`. Ce type d'objet est **normalisé** par le W3C. La liste des objets qui peuvent être fournis par typologie d'événement est disponible **ici** dans le MDN.

Les objets de type `Event` fournis par le navigateur Internet dépendent de l'événement qui est survenu. Dans l'exemple qui précédait, nous avions défini un gestionnaire d'événement pour l'événement `onmouseover` qui correspond au survol de l'élément par la souris. Pour ce type d'événement, le navigateur Internet nous fourni un objet de type `MouseEvent` qui hérite des caractéristiques d'un objet de type `Event`. Les objets de type `MouseEvent` sont définis **ici** dans le MDN.

Dans les fonctions gestionnaires d'événement, on peut utiliser les propriétés et les méthodes des objets d'événement fournis en paramètre par le navigateur Internet. On peut ainsi écrire :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
    <script>
      window.addEventListener('DOMContentLoaded', function(){
        var elementHTMLh2 = window.document.getElementById('titrePremiereSection');
        elementHTMLh2.addEventListener('click', function(argumentFourniParLeNavigateur){
          argumentFourniParLeNavigateur; //EventObject, contient un objet d'événement de type MouseEvent fourni par le navigateur Internet.
          argumentFourniParLeNavigateur.clientX; //Contient un nombre exprimant la coordonnée horizontale en pixels à laquelle l'événement est survenu.
          argumentFourniParLeNavigateur.clientY; //Contient un nombre exprimant la coordonnée verticale en pixels à laquelle l'événement est survenu.
          /*
          On peut imaginer d'utiliser ces deux informations pour effectuer des modifications de mise en forme du document. Par exemple, changer la position d'un élément pour le caler aux coordonnées de la souris...
          */
        });
      });
    </script>
  </head>
  <body>
    <h1>Document HTML5 quelconque</h1>
    <div>
      <h2 id="titrePremiereSection">Première section</h2>
      <p>Premier paragraphe</p>
    </div>
    <div>
      <h2>Deuxième section</h2>
      <p>Premier paragraphe et un <a href="#">lien vers un site</a></p>
    </div>
  </body>
</html>
```

Le MDN propose un chapitre sur les évènements **ici**. JavaScript est donc un langage événementiel orienté objet à prototypes qui propose d'interagir avec l'affichage par l'intermédiaire du DOM. Le langage nous offre la possibilité d'introduire de nouvelles fonctionnalités d'ergonomie sur un document Web. Un des apports majeurs du langage réside dans la possibilité de créer des algorithmes qui vont animer certaines portions de l'affichage.

6. Processus et Animations

1. **Notion de Processus**
2. **Créer des Processus**
3. **Le principe des animations**

6.1 Notion de Processus

En informatique, un processus (en anglais, « Thread ») est le nom donné à un ensemble d'instructions au moment elles sont exécutées par la machine. Le temps d'exécution d'un processus dépend du nombre d'instructions à exécuter. On peut dire que les instructions sont exécutées dans l'ordre (de haut en bas) dans lequel elles ont été écrites par le programmeur.

En JavaScript, on pourrait mesurer l'exécution d'un processus comme suit :

```
window.console.time('Durée d\'exécution'); //On exécute la méthode .time() de l'objet console. Elle prend en paramètre une chaîne
de caractère qui servira d'identifiant pour le chronomètre (Timer, en anglais)

for (i = 0; i <= 1000000; i++) { //On lance une première boucle qui va faire 1000000 de tours.
  //Dans la boucle, on ne fait rien.
};


for (j = 0; j <= 1000000; j++) { //On lance une deuxième boucle qui va faire 1000000 de tours.
  //Dans la boucle, on ne fait rien.
};

window.console.timeEnd('Durée d\'exécution'); //Affiche Durée d'exécution: 893.65ms sur mon ordinateur.
```

Dans l'exemple qui précède, la durée d'exécution du processus (ensemble des instructions) est de 893.65ms sur mon navigateur Internet sur mon ordinateur. En informatique, certains langages offrent la possibilité de créer des sous processus. C'est à dire des « morceaux » de programme qui pourraient être exécutés simultanément et sans bloquer le déroulement du programme principal.

Nous verrons qu'en JavaScript nous pouvons simuler ce comportement de sous processus ou créer de vrais sous processus.

6.2 Créer des Processus

En JavaScript, on peut **simuler** des sous processus en utilisant les méthodes `window.setTimeout()` documentée **ici**, `window.setInterval()` documentée **ici** ou `window.requestAnimationFrame()` documentée **ici**. Par exemple :

```
window.console.time('Durée d\'exécution'); //On exécute la méthode .time() de l'objet console. Elle prend en paramètre une chaîne
de caractère qui servira d'identifiant pour le chronomètre (Timer, en anglais)

window.setTimeout(function(){//On utilise la méthode .setTimeout() pour déclencher une fonction au bout de 0ms.
  for (i = 0; i <= 1000000; i++) { //Ici, on lance une première boucle qui va faire 1000000 de tours.
    //Dans la boucle, on ne fait rien.
  };
}, 0);


window.setTimeout(function(){//On utilise la méthode .setTimeout() pour déclencher une fonction au bout de 0ms.
  for (j = 0; j <= 1000000; j++) { //Ici, on lance une deuxième boucle qui va faire 1000000 de tours.
    //Dans la boucle, on ne fait rien.
  };
}, 0);

window.console.timeEnd('Durée d\'exécution'); //Affiche Durée d'exécution: 0.08ms sur mon ordinateur.
```

On constate que la durée d'exécution du processus est 0.08ms sur mon navigateur Internet sur mon ordinateur. Pourtant les 2 fonctions qui effectuent des boucles de 0 à 1000000 sont quand même exécutées par la méthode `.setTimeout()` à l'issue d'un délai de 0ms. Combien de temps prend l'exécution de ces fonctions ?

```
window.console.time('Durée d\'exécution du programme principal'); //On exécute la méthode .time() de l'objet console. Elle prend en paramètre une chaîne de caractère qui servira d'identifiant pour le chronomètre (Timer, en anglais)

window.setTimeout(function(){//On utilise la méthode .setTimeout() pour déclencher une fonction au bout de 0ms.
  window.console.time('Durée d\'exécution de la première fonction');
  for (i = 0; i <= 1000000; i++) { //Ici, on lance une première boucle qui va faire 1000000 de tours.
    //Dans la boucle, on ne fait rien.
  };
  window.console.timeEnd('Durée d\'exécution de la première fonction'); //Affiche Durée d\'exécution de la première fonction: 457.28ms sur mon ordinateur.
}, 0);


window.setTimeout(function(){//On utilise la méthode .setTimeout() pour déclencher une fonction au bout de 0ms.
  window.console.time('Durée d\'exécution de la seconde fonction');
  for (j = 0; j <= 1000000; j++) { //Ici, on lance une deuxième boucle qui va faire 1000000 de tours.
    //Dans la boucle, on ne fait rien.
  };
  window.console.timeEnd('Durée d\'exécution de la seconde fonction'); //Affiche Durée d\'exécution de la seconde fonction: 439.01ms sur mon ordinateur.
}, 0);

window.console.timeEnd('Durée d\'exécution du programme principal'); //Affiche Durée d\'exécution du programme principal: 0.05ms sur mon ordinateur.
```

A l'exécution, On constate d'abord que le processus principal, celui qui fait appel à la méthode `.setTimeout()`, est le premier à se terminer. La première durée à s'afficher est 0.05ms qui correspond à la durée d'exécution du processus principal. Puis on voit apparaître sur la console les durées prises par l'exécution des 2 fonctions exécutées par les `.setTimeout()`. Ces 2 fonctions prennent respectivement 457.28ms et 439.01ms de temps d'exécution sur mon ordinateur.

Que peut-on en conclure ?

L'utilisation de la méthode `.setTimeout()` est un moyen pour le programmeur de différer l'exécution de parties de programme **sans bloquer** l'exécution de son programme principal. En d'autres termes, il s'agit de simuler des « **sous processus** » indépendants d'un **processus principal**. Est-ce que les sous processus simulés s'exécutent simultanément ?

```
window.console.time('Durée d\'exécution du programme principal'); //On exécute la méthode .time() de l'objet console. Elle prend en paramètre une chaîne de caractère qui servira d'identifiant pour le chronomètre (Timer, en anglais)

window.setTimeout(function(){//On utilise la méthode .setTimeout() pour déclencher une fonction au bout de 0ms.
  window.console.time('Durée d\'exécution de la première fonction');
  for (i = 0; i <= 100; i++) { //Ici, on lance une première boucle qui va faire 1000000 de tours.
    window.console.log('i vaut :' + i);//On écrit dans la console la valeur courante de i.
  };
  window.console.timeEnd('Durée d\'exécution de la première fonction'); //Affiche Durée d\'exécution de la première fonction: 457.28ms sur mon ordinateur.
}, 0);


window.setTimeout(function(){//On utilise la méthode .setTimeout() pour déclencher une fonction au bout de 0ms.
  window.console.time('Durée d\'exécution de la seconde fonction');
  for (j = 0; j <= 100; j++) { //Ici, on lance une deuxième boucle qui va faire 1000000 de tours.
    window.console.log('j vaut :' + j);//On écrit dans la console la valeur courante de j.
  };
  window.console.timeEnd('Durée d\'exécution de la seconde fonction'); //Affiche Durée d\'exécution de la seconde fonction: 439.01ms sur mon ordinateur.
}, 0);

window.console.timeEnd('Durée d\'exécution du programme principal'); //Affiche Durée d\'exécution du programme principal: 0.05ms sur mon ordinateur.
```

On peut constater, à l'exécution de ce programme, qu'on a d'abord toutes les valeurs de *i* qui s'affichent puis celle de *j*. L'exécution des « **sous processus** » n'est donc **pas** simultanée. En JavaScript, l'utilisation la méthode `.setTimeout()` nous permet de **simuler** le comportement de sous processus qui seraient non bloquant mais pas de créer de vrais sous processus qui seraient non bloquants et exécutés simultanément. Pour obtenir ce type de comportement, nous pourrions utiliser l'objet fondamental `Worker` du langage, non standard, défini par le MDN **ici**. On trouve des exemples d'utilisation de cet objet **ici**.

Les méthodes `window.setTimeout()`, `window.setInterval()` ou `window.requestAnimationFrame()` nous permettent de créer des pseudo sous-processus que nous pourrons mettre à profit pour la réalisation d'animations.

6.3 Le principe des animations

Les animations reposent sur deux notions : les images clé (en anglais, « **keyframes** ») et les interpolations (en anglais, « **tweens** »).

Pour illustrer la notion d'images clé on peut imaginer un carré qui aurait une position initiale et une position finale comme ci-après :

1. Position initiale : coordonnée x : 10px, coordonnée y : 20px
1. Position finale : coordonnée x : 20px, coordonnée y : 30px

Nous avons défini ici 2 images clé. Maintenant, intéressons nous aux interpolations. Il s'agit de calculer toutes les positions intermédiaires que devra prendre le carré pour aller de la position initiale aux coordonnées (10,20) vers la position finale aux coordonnées (20,30). On aurait dans le cas d'une interpolation linéaire :

1. (10,20) : image clé, position initiale.
2. (11,21)
3. (12,22)
4. (13,23)
5. (14,24)
6. (15,25)
7. (16,26)
8. (17,27)
9. (18,28)
10. (19,29)
11. (20,30) : image clé, position finale.

Maintenant que nous avons défini les caractéristiques d'une animation intéressons nous à son implémentation en JavaScript dans un document Web. Sur le document Web suivant nous avons un élément HTML de type `<div>` qui reprend les caractéristiques du carré définit précédemment :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
  </head>
  <body>
    <div id="carre" style="position:absolute;width:10px;height:10px;top:20px;left:10px;background-color:green;">
      <!-- Cette <div>, mise en forme comme un carré, à pour position initiale (10,20), ce qui correspond à l'image clé définie
plus tôt. -->
    </div>
  </body>
</html>
```

En utilisant JavaScript et le DOM, nous pouvons accéder aux propriétés de positionnement du style de la représentant le carré et les modifier. On pourrait par exemple écrire :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
    <script>
      window.addEventListener('DOMContentLoaded', function(){
        var elementHTMLDiv = window.document.getElementById('carre');//On récupère une référence vers l'objet correspondant à la
<div> identifiée par la chaîne de caractère 'carre'.
        elementHTMLDiv.style.top = '30px'; //Ici j'affecte la coordonnée finale sur l'axe des y à la propriété .top du sous objet
style de l'objet correspondant à la <div> identifiée.
        elementHTMLDiv.style.left = '20px'; //Ici j'affecte la coordonnée finale sur l'axe des x à la propriété .left du sous objet
style de l'objet correspondant à la <div> identifiée.
      });
    </script>
  </head>
  <body>
    <div id="carre" style="position:absolute;width:10px;height:10px;top:20px;left:10px;background-color:green;">
      <!-- Cette <div> mise en forme comme un carré à pour position initiale (10,20), ce qui correspond à l'image clé définie plus
tôt. -->
    </div>
  </body>
</html>
```

Dans l'exemple qui précède, au chargement du document, les propriétés de style de l'élément HTML correspondant à la représentant le carré sont modifiées de telles sorte que le carré se retrouve positionné au niveau des coordonnées l'image clé finale. On voit donc que la notion d'image clé est facilement transposable en JavaScript. La difficulté réside maintenant dans la notion d'interpolation. Comment faire pour que, entre l'image clé initiale et l'image clé finale, la la prenne toutes les valeurs intermédiaires. **Intuitivement**, on écrirait :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
    <script>
      window.addEventListener('DOMContentLoaded', function(){
        var elementHTMLDiv = window.document.getElementById('carre');
        var y = parseFloat(elementHTMLDiv.style.top); //Ici on converti la chaîne de caractère 20px en le nombre 20.
        var x = parseFloat(elementHTMLDiv.style.left); //Ici on converti la chaîne de caractère 10px en le nombre 10.
        for (i = 0; i <= 10; i++) { //Ici, on écrit une boucle qui va incrémenter une variable i.
          elementHTMLDiv.style.top = y + i + 'px'; //Ici, on additionne i à la position initiale sur l'axe y à chaque tour de
boucle.
          elementHTMLDiv.style.left = x + i + 'px'; //Ici, on additionne i à la position initiale sur l'axe x à chaque tour de
boucle.
        };
      });
    </script>
  </head>
  <body>
    <div id="carre" style="position:absolute;width:10px;height:10px;top:20px;left:10px;background-color:green;">
      <!-- Cette <div> mise en forme comme un carré à pour position initiale (10,20), ce qui correspond à l'image clé définie plus
tôt. -->
    </div>
  </body>
</html>
```

Mais ce code ne produit pas le résultat attendu. L'utilisateur constatera que carré se retrouve immédiatement positionné sur l'image clé finale (aux coordonnées (20,30)). Comme on l'a vu au chapitre précédent, bien que toutes valeurs d'interpolation aient été successivement affectées, le programme s'exécutera trop rapidement pour que visuellement on puisse s'apercevoir d'un quelconque changement.

Il faut donc écrire un programme qui puisse afficher successivement chaque interpolation pendant une durée suffisante pour qu'elle soit perceptible à l'oeil nu. En utilisant la notion de **processus simulés**, nous pouvons concevoir un algorithme qui va déclencher un nouveau processus pour afficher chaque interpolation successivement, avec un délai, sans bloquer le déroulement général du programme principal. Concrètement, on pourra proposer en utilisant la méthode `window.setTimeout()` :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Document HTML5 quelconque</title>
    <script>
      window.addEventListener('DOMContentLoaded', function(){
        var elementHTMLDiv = window.document.getElementById('carre');
        //On déclare une fonction qui applique les coordonnées x,y d'un interpolation.
        var interpolation = function(){
          var yActuel = parseFloat(elementHTMLDiv.style.top); //Ici on converti la chaîne de caractère courante en nombre.
          var xActuel = parseFloat(elementHTMLDiv.style.left); //Ici on converti la chaîne de caractère courante en nombre.

          if (xActuel < 20 && yActuel < 30){ //Si les coordonnées actuelles sont différentes de celle de l'image clé finale.

            elementHTMLDiv.style.top = (yActuel + 1) + 'px'; //Ici, on affecte la nouvelle coordonnée sur l'axe des x.
            elementHTMLDiv.style.left = (xActuel + 1) + 'px'; //Ici, on affecte la nouvelle coordonnée sur l'axe des y.

            window.setTimeout(interpolation, 1000); //Ici on démarre un nouveau Timer qui va affecter l'interpolation suivante dans
un délai de 1000ms.
          };
        };

        window.setTimeout(interpolation, 0); //On démarre le pseudo sous processus qui prendra en charge chaque interpolation avec
un délai de 1000ms entre chacune d'entre elles.

      });
    </script>
  </head>
  <body>
    <div id="carre" style="position:absolute;width:10px;height:10px;top:20px;left:10px;background-color:green;">
      <!-- Cette <div> mise en forme comme un carré à pour position initiale (10,20), ce qui correspond à l'image clé définie plus
tôt. -->
    </div>
  </body>
</html>
```

On aurait pu utiliser de façon comparable la méthode `window.requestAnimationFrame()` ou bien encore la méthode `window.setInterval()`.

Au fil d'exercices pratiques nous découvrirons les subtilités du langage ainsi qu'un ensemble d'algorithmes fréquemment utilisés pour le développement dans le cadre du navigateur Internet. Vous pouvez retrouvez un ensemble de bonnes pratiques syntaxiques recommandées par Google **ici** et par Opera Software **ici**. Pour vous perfectionner, le MDN recense plusieurs ressources concernant le langage **ici**