Dynamisez vos sites web avec JavaScript !

40 heures Moyenne

Mis à jour le 16/04/2019

Les événements

Après l'introduction au DOM, il est temps d'approfondir ce domaine en abordant les événements en JavaScript. Au cours de ce chapitre, nous étudierons l'utilisation des événements sans le DOM, avec le DOM-0 (inventé par Netscape) puis avec le DOM-2. Nous verrons comment mettre en place ces événements, les utiliser, modifier leur comportement, etc.

Après ce chapitre, vous pourrez d'ores-et-déjà commencer à interagir avec l'utilisateur, vous permettant ainsi de créer des pages web interactives capables de réagir à diverses actions effectuées soit par le visiteur, soit par le navigateur.

Que sont les événements ?

Les événements permettent de déclencher une fonction selon qu'une action s'est produite ou non. Par exemple, on peut faire apparaître une fenêtre `alert()` lorsque l'utilisateur survole une zone d'une page Web.

« Zone » est un terme un peu vague, il vaut mieux parler d'élément (HTML dans la plupart des cas). Ainsi, vous pouvez très bien ajouter un événement à un élément de votre page Web (par exemple, une balise `<div>`) pour faire en sorte de déclencher un code JavaScript lorsque l'utilisateur fera une action sur l'élément en question.

La théorie

Voici la liste des événements principaux, ainsi que les actions à effectuer pour qu'ils se déclenchent :

Nom de l'événement	Action pour le déclencher
`click`	Cliquer (appuyer puis relâcher) sur l'élément
`dblclick`	Double-cliquer sur l'élément
`mouseover`	Faire entrer le curseur sur l'élément
`mouseout`	Faire sortir le curseur de l'élément
`mousedown`	Appuyer (sans relâcher) sur le bouton gauche de la souris sur l'élément
`mouseup`	Relâcher le bouton gauche de la souris sur l'élément
`mousemove`	Faire déplacer le curseur sur l'élément
`keydown`	Appuyer (sans relâcher) sur une touche de clavier sur l'élément
`keyup`	Relâcher une touche de clavier sur l'élément
`keypress`	Frapper (appuyer puis relâcher) une touche de clavier sur l'élément
`focus`	« Cibler » l'élément
`blur`	Annuler le « ciblage » de l'élément
`change`	Changer la valeur d'un élément spécifique aux formulaires (`input` , `checkbox` , etc.)
`input`	Taper un caractère dans un champ de texte (son support n'est pas complet sur tous les navigateurs)
`select`	Sélectionner le contenu d'un champ de texte (`input` , `textarea` , etc.)

Il a été dit précédemment que les événements `mousedown` et `mouseup` se déclenchaient

qui a choisi de bloquer cette possibilité. L'utilisation de ces deux événements se limite donc généralement au bouton gauche de la souris.

Toutefois, ce n'est pas tout, il existe aussi deux événements spécifiques à l'élément `<form>`, que voici :

Nom de l'événement	Action pour le déclencher
`submit`	Envoyer le formulaire
`reset`	Réinitialiser le formulaire

Tout cela est pour le moment très théorique, nous ne faisons que vous lister quelques événements existants, mais nous allons rapidement apprendre à les utiliser après un dernier petit passage concernant ce qu'on appelle le **focus**.

Retour sur le focus

Le focus définit ce qui peut être appelé le « ciblage » d'un élément. Lorsqu'un élément est ciblé, il va recevoir tous les événements de votre clavier. Un exemple simple est d'utiliser un `<input>` de type `text` : si vous cliquez dessus alors l'`input` possède le focus. Autrement dit : il est ciblé, et si vous tapez des caractères sur votre clavier vous allez les voir s'afficher *dans* l'`input` en question.

Le focus peut s'appliquer à de nombreux éléments, ainsi, si vous tapez sur la touche `Tabulation` de votre clavier alors que vous êtes sur une page Web, vous allez avoir un élément de ciblé ou de sélectionné qui recevra alors tout ce que vous tapez sur votre clavier. Par exemple, si vous avez un lien de ciblé et que vous tapez sur la touche `Entrée` de votre clavier alors vous serez redirigé vers l'URL contenue dans ce lien.

La pratique

Utiliser les événements

Bien, maintenant que vous avez vu le côté théorique (et barbant) des événements, nous allons pouvoir passer un peu à la pratique. Toutefois, dans un premier temps, il n'est que question de vous faire découvrir à quoi sert tel ou tel événement et comment il réagit, nous allons donc voir comment les utiliser sans le DOM, ce qui est considérablement plus limité.

Bien, commençons par l'événement `click` sur un simple `<span>` :

```
html
```

Comme vous pouvez le constater, il suffit de cliquer sur le texte pour que la boîte de dialogue s'affiche. Afin d'obtenir ce résultat nous avons ajouté à notre `<span>` un attribut contenant les deux lettres « on » et le nom de notre événement « click » ; nous obtenons donc « onclick ».

Cet attribut possède une valeur qui est un code JavaScript, vous pouvez y écrire quasiment tout ce que vous souhaitez, mais tout doit tenir entre les guillemets de l'attribut.

Le mot-clé `this`

Ce mot-clé n'est, normalement, pas censé vous servir dès maintenant, cependant il est toujours bon de le connaître pour les événements. Il s'agit d'une propriété pointant sur l'objet actuellement en cours d'utilisation. Donc, si vous faites appel à ce mot-clé lorsqu'un événement est déclenché, l'objet pointé sera l'élément qui a déclenché l'événement. Exemple :

html

```
<span onclick="alert('Voici le contenu de l\'élément que vous avez cliqué :\n\n' + this.innerHTML);">Cliquez-moi !</span>
```

Essayer le code

Retour sur le focus

Afin de bien vous montrer ce qu'est le focus, voici un exemple qui vous montrera ce que ça donne sur un `input` classique et un lien :

html

```
<input id="input" type="text" size="50" value="Cliquez ici !" onfocus="this.value='Appuyez maintenant sur votre touche de tabulation.';" onblur="this.value='Cliquez ici !';"/>
<br /><br/>
<a href="#" onfocus="document.getElementById('input').value = 'Vous avez maintenant le focus sur le lien, bravo !';">Un lien bidon</a>
```

Essayer le code

Comme vous pouvez le constater, lorsque vous cliquez sur l'`input`, celui-ci « possède » le focus : il exécute donc l'événement et affiche alors un texte différent vous demandant d'appuyer sur votre touche de tabulation. L'appui sur la touche de tabulation permet de faire passer le focus à l'élément suivant. En clair, en appuyant sur cette touche vous faites perdre le focus à l'`input`, ce qui déclenche l'événement `blur` (qui désigne la perte du focus) et fait passer le focus sur le lien qui affiche alors son message grâce à son événement `focus`.

Bloquer l'action par défaut de certains événements

Passons maintenant à un petit problème : quand vous souhaitez appliquer un événement `click`

Si vous avez essayé le code, vous avez sûrement remarqué que la fonction `alert()` a bien fonctionné, mais qu'après vous avez été redirigé vers le Site du Zéro, or on souhaite bloquer cette redirection. Pour cela, il suffit juste d'ajouter le code `return false;` dans notre événement `click` :

```html
<a href="http://www.siteduzero.com" onclick="alert('Vous avez cliqué !'); return false;">Cliquez-moi !</a>
```

Essayer le code

Ici, le `return false;` sert juste à bloquer l'action par défaut de l'événement qui le déclenche.

> À noter que l'utilisation de `return true;` permet de faire fonctionner l'événement comme si de rien n'était. En clair, comme si on n'utilisait pas de `return false;`
> Cela peut avoir son utilité si vous utilisez, par exemple, la fonction `confirm()` dans votre événement.

L'utilisation de « javascript: » dans les liens : une technique prohibée

Dans certains cas, vous allez devoir créer des liens juste pour leur attribuer un événement `click` et non pas pour leur fournir un lien vers lequel rediriger. Dans ce genre de cas, il est courant de voir ce type de code :

```html
<a href="javascript: alert('Vous avez cliqué !');">Cliquez-moi !</a>
```

> Nous vous déconseillons fortement de faire cela ! Si vous le faites quand même, ne venez pas dire que vous avez appris le JavaScript grâce à ce cours, ce serait la honte pour nous ! 

Plus sérieusement, il s'agit d'une vieille méthode qui permet d'insérer du JavaScript directement dans l'attribut `href` de votre lien juste en ajoutant `javascript:` au début de l'attribut. Cette technique est maintenant obsolète et nous serions déçus de vous voir l'utiliser, nous vous en déconseillons donc très fortement l'utilisation et vous proposons même une méthode alternative :

```html
<a href="#" onclick="alert('Vous avez cliqué !'); return false;">Cliquez-moi !</a>
```

Essayer le code

Concrètement, qu'est-ce qui change ? On a tout d'abord remplacé l'immonde `javascript:` par un dièse (#) puis on a mis notre code JavaScript dans l'événement approprié (`click`). Par ailleurs, on

> OK, j'ai compris, mais pourquoi un `return false;` ?

Tout simplement parce que le dièse redirige tout en haut de la page Web, ce qui n'est pas ce que l'on souhaite. On bloque donc cette redirection avec notre petit bout de code.

> Vous savez maintenant que l'utilisation de `javascript:` dans les liens est prohibée et c'est déjà une bonne chose. Cependant, gardez bien à l'esprit que l'utilisation d'un lien uniquement pour le déclenchement d'un événement `click` n'est pas une bonne chose, préférez plutôt l'utilisation d'une balise `<button>` à laquelle vous aurez retiré le style CSS. La balise `<a>`, elle, est conçue pour rediriger vers une page Web et non pas pour servir exclusivement de déclencheur !

Les événements au travers du DOM

Bien, maintenant que nous avons vu l'utilisation des événements sans le DOM, nous allons passer à leur utilisation au travers de l'interface implémentée par Netscape que l'on appelle le DOM-0 puis au standard de base actuel : le DOM-2.

Le DOM-0

Cette interface est vieille mais n'est pas forcément dénuée d'intérêt. Elle reste très pratique pour créer des événements et peut parfois être préférée au DOM-2.

Commençons par créer un simple code avec un événement `click` :

html

```
<span id="clickme">Cliquez-moi !</span>

<script>

    var element = document.getElementById('clickme');

    element.onclick = function() {
        alert("Vous m'avez cliqué !");
    };

</script>
```

Essayer le code

Alors, voyons par étapes ce que nous avons fait dans ce code :

Comme vous le voyez, on définit maintenant les événements non plus dans le code HTML mais directement en JavaScript. Chaque événement standard possède donc une propriété dont le nom est, à nouveau, précédé par les deux lettres « on ». Cette propriété ne prend plus pour valeur un code JavaScript brut, mais soit le nom d'une fonction, soit une fonction anonyme. Bref, dans tous les cas, il faut lui fournir une fonction qui contiendra le code à exécuter en cas de déclenchement de l'événement.

Concernant la suppression d'un événement avec le DOM-0, il suffit tout simplement de lui attribuer une fonction anonyme vide :

javascript

```
element.onclick = function() {};
```

Voilà tout pour les événements DOM-0, nous pouvons maintenant passer au cœur des événements : le DOM-2 et l'objet `Event` .

Le DOM-2

Nous y voici enfin ! Alors, tout d'abord, pourquoi le DOM-2 et non pas le DOM-0 voire pas de DOM du tout ? Concernant la méthode sans le DOM, c'est simple : on ne peut pas y utiliser l'objet `Event` qui est pourtant une mine d'informations sur l'événement déclenché. Il est donc conseillé de mettre cette méthode de côté dès maintenant (nous l'avons enseignée juste pour que vous sachiez la reconnaître).
En ce qui concerne le DOM-0, il a deux problèmes majeurs : il est vieux, et il ne permet pas de créer plusieurs fois le même événement.

Le DOM-2, lui, permet la création multiple d'un même événement et gère aussi l'objet `Event` .
Autrement dit, **le DOM-2 c'est bien, mangez-en !**

> En clair, il faut constamment utiliser le DOM-2 ?

Cela est très fortement conseillé surtout lorsque vous en viendrez à utiliser des librairies au sein de vos sites web. Si ces librairies ajoutent des évènements DOM-0 à vos éléments HTML, vous aurez alors deux problèmes :

- Ce sont de mauvaises librairies, elles n'ont pas à faire ça. Nous vous conseillons d'en trouver d'autres, le web n'en manque pas.
- Les évènements de votre propre code seront écrasés ou bien vous écraserez ceux des librairies. Dans les deux cas, cela s'avère assez gênant.

Le DOM-2

```
<script>
    var element = document.getElementById('clickme');

    element.addEventListener('click', function() {
        alert("Vous m'avez cliqué !");
    });
</script>
```

Essayer le code

Concrètement, qu'est-ce qui change par rapport au DOM-0 ? Alors tout d'abord nous n'utilisons plus une propriété mais une méthode nommée `addEventListener()`, et qui prend trois paramètres (bien que nous n'en ayons spécifié que deux) :

- Le nom de l'événement (sans les lettres « on ») ;
- La fonction à exécuter ;
- Un booléen **optionnel** pour spécifier si l'on souhaite utiliser la phase de capture ou bien celle de bouillonnement. Nous expliquerons ce concept un peu plus tard dans ce chapitre.

Une petite explication s'impose pour ceux qui n'arriveraient éventuellement pas à comprendre le code précédent : nous avons bel et bien utilisé la méthode `addEventListener()`, elle est simplement écrite sur trois lignes :

- La première ligne contient l'appel à la méthode `addEventListener()`, le premier paramètre, et l'initialisation de la fonction anonyme pour le deuxième paramètre ;
- La deuxième ligne contient le code de la fonction anonyme ;
- La troisième ligne contient l'accolade fermante de la fonction anonyme, puis le troisième paramètre.

Ce code revient à écrire le code suivant, de façon plus rapide :

javascript

```
var element = document.getElementById('clickme');

var myFunction = function() {
    alert("Vous m'avez cliqué !");
};

element.addEventListener('click', myFunction);
```

Comme indiqué plus haut, le DOM-2 permet la création multiple d'événements identiques pour un même élément, ainsi, vous pouvez très bien faire ceci :

html

```
<span id="clickme">Cliquez-moi !</span>
```

```
        alert("Et de un !");
    });

    // Deuxième événement click
    element.addEventListener('click', function() {
        alert("Et de deux !");
    });
</script>
```

Essayer le code

Si vous avez exécuté ce code, vous avez peut-être eu les événements déclenchés dans l'ordre de création, cependant ce ne sera pas forcément le cas à chaque essai. En effet, l'ordre de déclenchement est un peu aléatoire...

Venons-en maintenant à la suppression des événements ! Celle-ci s'opère avec la méthode `removeEventListener()` et se fait de manière très simple :

javascript

```
element.addEventListener('click', myFunction); // On crée l'événement

element.removeEventListener('click', myFunction); // On supprime l'événement en lui repassant les mêmes paramètres
```

Toute suppression d'événement avec le DOM-2 se fait avec les mêmes paramètres utilisés lors de sa création ! Cependant, cela ne fonctionne pas aussi facilement avec les fonctions anonymes ! Tout événement DOM-2 créé avec une fonction anonyme est particulièrement complexe à supprimer, car il faut posséder une référence vers la fonction concernée, ce qui n'est généralement pas le cas avec une fonction anonyme.

> Attention ! Les versions d'Internet Explorer antérieures à la version 9 ne supportent pas l'ajout d'évènements DOM-2, tout du moins pas de la même manière que la version standardisée.

Les phases de capture et de bouillonnement

Du côté de la théorie

Vous souvenez-vous que notre méthode `addEventListener()` prend trois paramètres ? Nous vous avions dit que nous allions revenir sur l'utilisation de son troisième paramètre plus tard. Eh bien ce « plus tard » est arrivé !

> Capture ? Bouillonnement ? De quoi parle-t-on ?

deuxième, le **bouillonnement** (*bubbling* en anglais), s'exécute après que l'événement a été déclenché. Toutes deux permettent de définir le sens de propagation des événements.

Mais qu'est-ce que la propagation d'un événement ? Pour expliquer cela, prenons un exemple avec ces deux éléments HTML :

```html
<div>
    <span>Du texte !</span>
</div>
```

Si nous attribuons une fonction à l'événement `click` de chacun de ces deux éléments et que l'on clique sur le texte, quel événement va se déclencher en premier à votre avis ? Bonne question n'est-ce pas ?

Notre réponse se trouve dans les phases de capture et de bouillonnement. Si vous décidez d'utiliser la capture, alors l'événement du `<div>` se déclenchera en premier puis viendra ensuite l'événement du `<span>`. En revanche, si vous utilisez le bouillonnement, ce sera d'abord l'événement du `<span>` qui se déclenchera, puis viendra par la suite celui du `<div>`.

> La phase de bouillonnement est celle définie par défaut et sera probablement celle que vous utiliserez la majorité du temps;

Voici un petit code qui met en pratique l'utilisation de ces deux phases :

```html
<div id="capt1">
    <span id="capt2">Cliquez-moi pour la phase de capture.</span>
</div>

<div id="boul1">
    <span id="boul2">Cliquez-moi pour la phase de bouillonnement.</span>
</div>

<script>
    var capt1 = document.getElementById('capt1'),
        capt2 = document.getElementById('capt2'),
        boul1 = document.getElementById('boul1'),
        boul2 = document.getElementById('boul2');

    capt1.addEventListener('click', function() {
        alert("L'événement du div vient de se déclencher.");
    }, true);

    capt2.addEventListener('click', function() {
```

```
26
27      boul2.addEventListener('click', function() {
28          alert("L'événement du span vient de se déclencher.");
29      }, false);
30  </script>
```

Essayer le code

Et pour finir, un lien vers la spécification du W3C concernant ces phases si vous avez envie d'aller plus loin. Il est conseillé de regarder ce lien ne serait-ce que pour voir le schéma fourni qui explique bien le concept de ces phases.

L'objet Event

Maintenant que nous avons vu comment créer et supprimer des événements, nous pouvons passer à l'objet Event !

Généralités sur l'objet `Event`

Tout d'abord, à quoi sert cet objet ? À vous fournir une multitude d'informations sur l'événement actuellement déclenché. Par exemple, vous pouvez récupérer quelles sont les touches actuellement enfoncées, les coordonnées du curseur, l'élément qui a déclenché l'événement... Les possibilités sont nombreuses !

Cet objet est bien particulier dans le sens où il n'est accessible que lorsqu'un événement est déclenché. Son accès ne peut se faire que dans une fonction exécutée par un événement, cela se fait de la manière suivante avec le DOM-0 :

javascript

```
1  element.onclick = function(e) { // L'argument « e » va récupérer une référence vers l'objet « Event »
2      alert(e.type); // Ceci affiche le type de l'événement (click, mouseover, etc.)
3  };
```

Et de cette façon là avec le DOM-2 :

javascript

```
1  element.addEventListener('click', function(e) { // L'argument « e » va récupérer une référence vers l'objet « Event »
2      alert(e.type); // Ceci affiche le type de l'événement (click, mouseover, etc.)
3  });
```

Il est important de préciser que l'objet `Event` peut se récupérer dans un argument autre que `e` ! Vous pouvez très bien le récupérer dans un argument nommé `test`, `hello`, ou autre... Après tout, l'objet `Event` est tout simplement passé en référence à l'argument de votre fonction, ce qui vous permet de choisir le nom que vous souhaitez.

objet (attention, tout n'est pas présenté, seulement l'essentiel) :

Récupérer l'élément de l'événement actuellement déclenché

Une des plus importantes propriétés de notre objet se nomme `target`. Celle-ci permet de récupérer une référence vers l'élément dont l'événement a été déclenché (exactement comme le `this` pour les événements sans le DOM ou avec DOM-0), ainsi vous pouvez très bien modifier le contenu d'un élément qui a été cliqué :

html

```
<span id="clickme">Cliquez-moi !</span>

<script>
    var clickme = document.getElementById('clickme');

    clickme.addEventListener('click', function(e) {
        e.target.innerHTML = 'Vous avez cliqué !';
    });
</script>
```

Essayer le code

Récupérer l'élément à l'origine du déclenchement de l'événement

> Hum, ce n'est pas un peu la même chose ?

Eh bien non ! Pour expliquer cela de façon simple, certains événements appliqués à un élément parent peuvent se propager d'eux-mêmes aux éléments enfants ; c'est le cas des événements `mouseover`, `mouseout`, `mousemove`, `click`... ainsi que d'autres événements moins utilisés. Regardez donc cet exemple pour mieux comprendre :

html

```
<p id="result"></p>

<div id="parent1">
    Parent
    <div id="child1">Enfant N°1</div>
    <div id="child2">Enfant N°2</div>
</div>

<script>
    var parent1 = document.getElementById('parent1'),
        result = document.getElementById('result');

    parent1.addEventListener('mouseover', function(e) {
        result.innerHTML = "L'élément déclencheur de l'événement \"mouseover\" possède l'ID : " + e.target.id;
```

En testant cet exemple, vous avez sûrement remarqué que la propriété `target` renvoyait toujours l'élément déclencheur de l'événement, or nous souhaitons obtenir l'élément sur lequel a été appliqué l'événement. Autrement dit, on veut connaître l'élément à l'origine de cet événement, et non pas ses enfants.

La solution est simple : utiliser la propriété `currentTarget` au lieu de `target`. Essayez donc par vous-mêmes après modification de cette seule ligne, l'ID affiché ne changera jamais :

javascript
```
result.innerHTML = "L'élément déclencheur de l'événement \"mouseover\" possède l'ID : " + e.currentTarget.id;
```

Essayer le code complet

> Attention ! Cette propriété n'est pas supportée par les versions d'Internet Explorer antérieures à la 9.

Récupérer la position du curseur

La position du curseur est une information très importante, beaucoup de monde s'en sert pour de nombreux scripts comme le *drag & drop*. Généralement, on récupère la position du curseur par rapport au coin supérieur gauche de la page Web, cela dit il est aussi possible de récupérer sa position par rapport au coin supérieur gauche de l'écran. Toutefois, dans ce tutoriel, nous allons nous limiter à la page Web. Regardez la documentation de l'objet `Event` si vous souhaitez en apprendre plus.

Pour récupérer la position de notre curseur, il existe deux propriétés : `clientX` pour la position horizontale et `clientY` pour la position verticale. Étant donné que la position du curseur change à chaque déplacement de la souris, il est donc logique de dire que l'événement le plus adapté à la majorité des cas est `mousemove`.

> Il est très fortement déconseillé d'essayer d'exécuter la fonction `alert()` dans un événement `mousemove` ou bien vous allez rapidement être submergés de fenêtres !

Comme d'habitude, voici un petit exemple pour que vous compreniez bien :

html
```
<div id="position"></div>

<script>
    var position = document.getElementById('position');
```

Essayer le code

Pas très compliqué, n'est-ce pas ? Bon, il est possible que vous trouviez l'intérêt de ce code assez limité, mais quand vous saurez manipuler les propriétés CSS des éléments vous pourrez, par exemple, faire en sorte que des éléments HTML suivent votre curseur. Ce sera déjà bien plus sympathique ! 

> Dans le cadre de cet exemple qui ne permet pas l'utilisation de la fonction `alert()` , vous pourriez utiliser la méthode `console.log()` , vous êtes invités à essayer cela par vous-mêmes, vous verrez que cela est bien pratique.
>
> Si vous vous demandez pourquoi nous nous en servons pas dans nos exercices, c'est tout simplement parce que cela vous entraîne à utiliser le DOM et vous évite d'ouvrir votre console à-tout-va.

Récupérer l'élément en relation avec un événement de souris

Cette fois nous allons étudier une propriété un peu plus « exotique » assez peu utilisée mais qui peut pourtant se révéler très utile ! Il s'agit de `relatedTarget` et elle ne s'utilise qu'avec les événements `mouseover` et `mouseout` .
Cette propriété remplit deux fonctions différentes selon l'événement utilisé. Avec l'événement `mouseout` , elle vous fournira l'objet de l'élément sur lequel le curseur vient d'entrer ; avec l'événement `mouseover` , elle vous fournira l'objet de l'élément dont le curseur vient de sortir.

Voici un exemple qui illustre son fonctionnement :

html

```
<p id="result"></p>

<div id="parent1">
    Parent N°1<br /> Mouseover sur l'enfant
    <div id="child1">Enfant N°1</div>
</div>

<div id="parent2">
    Parent N°2<br /> Mouseout sur l'enfant
    <div id="child2">Enfant N°2</div>
</div>

<script>
    var child1 = document.getElementById('child1'),
        child2 = document.getElementById('child2'),
        result = document.getElementById('result');
```

```
    child2.addEventListener('mouseout', function(e) {
        result.innerHTML = "L'élément survolé juste après que le curseur ait quitté l'enfant n°2 est : " + e.relatedTarget.id;
    });
</script>
```

Essayer le code

Récupérer les touches frappées par l'utilisateur

La récupération des touches frappées se fait par le biais de trois événements différents. Dit comme ça, cela laisse sur un sentiment de complexité, mais vous allez voir qu'au final tout est beaucoup plus simple qu'il n'y paraît.

Les événements `keyup` et `keydown` sont conçus pour capter toutes les frappes de touches. Ainsi, il est parfaitement possible de détecter l'appui sur la touche `A` voire même sur la touche `Ctrl`.
La différence entre ces deux événements se situe dans l'ordre de déclenchement : `keyup` se déclenche lorsque vous relâchez une touche, tandis que `keydown` se déclenche au moment de l'appui sur la touche (comme `mousedown`).
Cependant, faites bien attention avec ces deux événements : toutes les touches retournant un caractère retourneront un caractère *majuscule*, que la touche `Maj` soit pressée ou non.

L'événement `keypress`, lui, est d'une toute autre utilité : il sert uniquement à capter les touches qui écrivent un caractère, oubliez donc les `Ctrl`, `Alt` et autres touches de ce genre qui n'affichent pas de caractère. Alors, forcément, vous vous demandez probablement à quoi peut bien servir cet événement au final ? Eh bien son avantage réside dans sa capacité à détecter les combinaisons de touches ! Ainsi, si vous faites la combinaison `Maj` + `A`, l'événement `keypress` détectera bien un A majuscule là où les événements `keyup` et `keydown` se déclencheront deux fois, une fois pour la touche `Maj` et une deuxième fois pour la touche `A`.

> Et j'utilise quelle propriété pour récupérer mon caractère, du coup ?

Si nous devions énumérer toutes les propriétés capables de vous fournir une valeur, il y en aurait trois : `keyCode`, `charCode` et `which`. Ces propriétés renvoient chacune un code ASCII correspondant à la touche pressée.
Cependant, la propriété `keyCode` est amplement suffisante dans tous les cas, comme vous pouvez le constater dans l'exemple qui suit :

html

```
<p>
    <input id="field" type="text" />
```

```
    </tr>
    <tr>
        <td>keypress</td>
        <td id="press"></td>
    </tr>
    <tr>
        <td>keyup</td>
        <td id="up"></td>
    </tr>
</table>

<script>
    var field = document.getElementById('field'),
        down = document.getElementById('down'),
        press = document.getElementById('press'),
        up = document.getElementById('up');

    document.addEventListener('keydown', function(e) {
        down.innerHTML = e.keyCode;
    });

    document.addEventListener('keypress', function(e) {
        press.innerHTML = e.keyCode;
    });

    document.addEventListener('keyup', function(e) {
        up.innerHTML = e.keyCode;
    });
</script>
```

Essayer le code

> Je ne veux pas obtenir un code, mais le caractère !

Dans ce cas, il n'existe qu'une seule solution : la méthode `fromCharCode()`. Elle prend en paramètre une infinité d'arguments. Cependant, pour des raisons un peu particulières qui ne seront abordées que plus tard dans ce cours, sachez que cette méthode s'utilise avec le préfixe `String.`, comme suit :

javascript

```
String.fromCharCode(/* valeur */);
```

Cette méthode est donc conçue pour convertir les valeurs ASCII vers des caractères lisibles. Faites donc bien attention à n'utiliser cette méthode qu'avec un événement `keypress` afin d'éviter d'afficher, par exemple, le caractère d'un code correspondant à la touche `Ctrl`, cela ne fonctionnera pas !

Bloquer l'action par défaut de certains événements

Eh oui, nous y revenons ! Nous avons vu qu'il est possible de bloquer l'action par défaut de certains événements, comme la redirection d'un lien vers une page Web. Sans le DOM-2, cette opération était très simple vu qu'il suffisait d'écrire `return false;`. Avec l'objet `Event`, c'est quasiment tout aussi simple vu qu'il suffit juste d'appeler la méthode `preventDefault()` !

Reprenons l'exemple que nous avions utilisé pour les événements sans le DOM et utilisons donc cette méthode :

html

```
<a id="link" href="http://www.siteduzero.com">Cliquez-moi !</a>

<script>
    var link = document.getElementById('link');

    link.addEventListener('click', function(e) {
        e.preventDefault(); // On bloque l'action par défaut de cet événement
        alert('Vous avez cliqué !');
    });
</script>
```

Essayer le code

C'est simple comme bonjour n'est-ce pas ?

> Attention ! Le blocage de l'action par défaut ne fonctionne pas pour les versions d'Internet Explorer antérieures à la 9. I lest cependant possible de résoudre cela simplement en faisant un simple `e.returnValue = false;` au sein du code de votre évènement.

Résoudre les problèmes d'héritage des événements

En JavaScript, il existe un problème fréquent que nous vous proposons d'étudier et de résoudre afin de vous éviter bien des peines lorsque cela vous arrivera.

Le problème

Plutôt que de vous expliquer le problème, nous allons vous le faire constater. Prenez donc ce code HTML ainsi que ce code CSS :

html

```
<div id="myDiv">
    <div>Texte 1</div>
    <div>Texte 2</div>
    <div>Texte 3</div>
```

```
#myDiv, #results {
    margin: 50px;
}

#myDiv {
    padding: 10px;
    width: 200px;
    text-align: center;
    background-color: #000;
}

#myDiv div {
    margin: 10px;
    background-color: #555;
}
```

Maintenant, voyons ce que nous souhaitons obtenir. Notre but ici est de faire en sorte de détecter quand le curseur entre sur notre élément `#myDiv` et quand il en ressort. Vous allez donc penser qu'il n'y a rien de plus facile et vous lancer dans un code de ce genre :

javascript

```
var myDiv = document.getElementById('myDiv'),
    results = document.getElementById('results');

myDiv.addEventListener('mouseover', function() {
    results.innerHTML += "Le curseur vient d'entrer.<br />";
});

myDiv.addEventListener('mouseout', function() {
    results.innerHTML += "Le curseur vient de sortir.<br />";
});
```

Eh bien soit ! Pourquoi ne pas l'essayer ?

Alors ? Avez-vous essayé de faire passer votre curseur sur toute la surface du `<div>` `#myDiv` ? Il y a effectivement quelques lignes en trop qui s'affichent dans nos résultats...

> Je ne vois absolument pas d'où ça peut venir...

Si cela peut vous rassurer, personne ne voit bien d'où cela peut venir au premier coup d'œil. En fait, le souci est tout bête et a déjà été fortement évoqué au travers de ce chapitre, relisez donc ceci :

Citation

> Certains événements appliqués à un élément parent peuvent se propager d'eux-mêmes aux éléments enfants, c'est le cas des événements `mouseover` `mouseout` `mousemove` `click`

un `<div>` enfant, vous allez déclencher l'événement `mouseout` sur `#myDiv` et l'événement `mouseover` sur le `<div>` enfant.

La solution

Afin de pallier ce problème, il existe une solution assez tordue. Vous souvenez-vous de la propriété `relatedTarget` abordée dans ce chapitre ? Son but va être de détecter quel est l'élément vers lequel le curseur se dirige ou de quel élément il provient.

Ainsi, nous avons deux cas de figure :

- Dans le cas de l'événement `mouseover`, nous devons détecter la *provenance* du curseur. Si le curseur vient d'un enfant de `#myDiv` alors le code de l'événement ne devra pas être exécuté. S'il provient d'un élément extérieur à `#myDiv` alors l'exécution du code peut s'effectuer.
- Dans le cas de `mouseout`, le principe est similaire, si ce n'est que là nous devons détecter la *destination* du curseur. Dans le cas où la destination du curseur est un enfant de `#myDiv` alors le code de l'événement n'est pas exécuté, sinon il s'exécutera sans problème.

Mettons cela en pratique avec l'événement `mouseover` pour commencer. Voici le code d'origine :

javascript

```
myDiv.addEventListener('mouseover', function() {
    results.innerHTML += "Le curseur vient d'entrer.";
});
```

Maintenant, il nous faut savoir si l'élément en question est un enfant direct de `myDiv` ou non. La solution consiste à remonter tout le long de ses éléments parents jusqu'à tomber soit sur `myDiv`, soit sur l'élément `<body>` qui désigne l'élément HTML le plus haut dans notre document. Il va donc nous falloir une boucle `while` :

javascript

```
myDiv.addEventListener('mouseover', function(e) {

    var relatedTarget = e.relatedTarget;

    while (relatedTarget != myDiv && relatedTarget.nodeName != 'BODY') {
        relatedTarget = relatedTarget.parentNode;
    }

    results.innerHTML += "Le curseur vient d'entrer.";

});
```

```
myDiv.addEventListener('mouseover', function(e) {

    var relatedTarget = e.relatedTarget;

    while (relatedTarget != myDiv && relatedTarget.nodeName != 'BODY') {
        relatedTarget = relatedTarget.parentNode;
    }

    if (relatedTarget != myDiv) {
        results.innerHTML += "Le curseur vient d'entrer.";
    }

});
```

Cependant, il reste encore un petit cas de figure qui n'a pas été géré et qui peut être source de problèmes ! Comme vous le savez, la balise `<body>` ne couvre pas forcément la page Web complète de votre navigateur, ce qui fait que votre curseur peut provenir d'un élément situé encore plus haut que la balise `<body>`. Cet élément correspond à la balise `<html>` — soit l'élément `document` en JavaScript —, il nous faut donc faire une petite modification afin de bien préciser que si le curseur provient de `document` il ne peut forcément pas provenir de `myDiv` :

javascript

```
myDiv.addEventListener('mouseover', function(e) {

    var relatedTarget = e.relatedTarget;

    while (relatedTarget != myDiv && relatedTarget.nodeName != 'BODY' && relatedTarget != document) {
        relatedTarget = relatedTarget.parentNode;
    }

    if (relatedTarget != myDiv) {
        results.innerHTML += "Le curseur vient d'entrer.";
    }

});
```

Voilà ! Maintenant, notre événement `mouseover` fonctionne comme nous le souhaitions ! Rassurez-vous, vous avez fait le plus gros, il ne nous reste plus qu'à adapter un peu le code pour l'événement `mouseout`. Cet événement va utiliser le même code que celui de `mouseover` à deux choses près :

- Le texte à afficher n'est pas le même ;
- Nous n'utilisons plus `mouseover` mais `mouseout`, car nous souhaitons l'élément de destination.

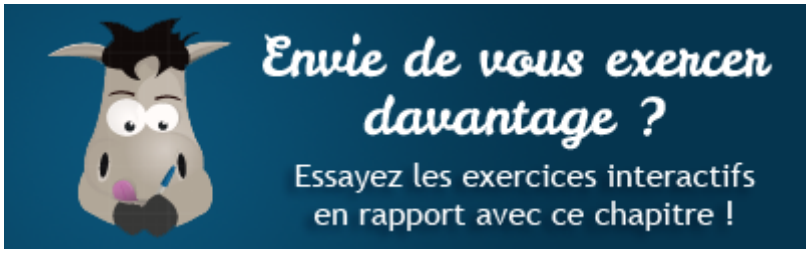
```
4
5       while (relatedTarget != myDiv && relatedTarget.nodeName != 'BODY' && relatedTarget != document) {
6           relatedTarget = relatedTarget.parentNode;
7       }
8
9       if (relatedTarget != myDiv) {
10          results.innerHTML += "Le curseur vient de sortir.<br />";
11      }
12
13  });
```

Enfin, nous avons terminé ! Il est grand temps d'essayer le code complet !

L'étude de ce problème était quelque peu avancée par rapport à vos connaissances actuelles, sachez que vous n'êtes pas obligés de retenir la solution. Retenez cependant qu'elle existe et que vous pouvez la trouver ici, dans ce chapitre, car ce genre de soucis peut être très embêtant dans certains cas, notamment quand il s'agit de faire des animations.

En résumé

- Les événements sont utilisés pour appeler une fonction à partir d'une action produite ou non par l'utilisateur.
- Différents événements existent pour détecter certaines actions comme le clic, le survol, la frappe au clavier et le contrôle des champs de formulaires.
- Le DOM-0 est l'ancienne manière de capturer des événements. Le DOM-2 introduit l'objet `Event` et la fameuse méthode `addEventListener()`.
- L'objet `Event` permet de récolter toutes sortes d'informations se rapportant à l'événement déclenché : son type, depuis quel élément il a été déclenché, la position du curseur, les touches frappées... Il est aussi possible de bloquer l'action d'un événement avec `preventDefault()`.
- Parfois, un événement appliqué sur un parent se propage à ses enfants. Cet héritage des événements peut provoquer des comportements inattendus.



- Q.C.M.
- Utilisation d'addEventListener()

J'AI TERMINÉ CE CHAPITRE ET JE PASSE AU SUIVANT

MANIPULER LE CODE HTML (PARTIE 2/2)

LES FORMULAIRES

Les professeurs

Sébastien

Johann Pardanaud

Découvrez aussi ce cours en...

Livre PDF

OpenClassrooms

L'entreprise

Alternance

Forum

Blog

Nous rejoindre

Entreprises