JS

Navigateur : Document, Évènements, Interfaces

3

Les événements

Introduction aux événements du navigateur

Un événement est le signal que quelque chose s'est produit. Tous les nœuds DOM génèrent de tels signaux (mais les événements ne se limitent pas au DOM).

Voici une liste des événements DOM les plus utiles, juste pour y jeter un œil :

Événements de souris :

- **click**– lorsque la souris clique sur un élément (les appareils à écran tactile le génèrent en un clic).
- **contextmenu** – lorsque la souris fait un clic droit sur un élément.
- **mouseover** / **mouseout** – lorsque le curseur de la souris survole/quitte un élément.
- **mousedown / mouseup – lorsque le bouton de la souris est enfoncé/relâché sur un élément.**
- **mousemove** – lorsque la souris est déplacée.

Événements de clavier :

- **keydown et keyup** – lorsqu'une touche du clavier est enfoncée et relâchée.

Les événements

Événements d'élément de formulaire :

- **submit** – lorsque le visiteur soumet un <form>.
- **focus** – lorsque le visiteur se concentre sur un élément, par exemple sur un <input>.

Événements document :

- **DOMContentLoaded**– lorsque le HTML est chargé et traité, le DOM est entièrement construit.

Événements CSS :

- **transitionend**– quand une animation CSS se termine.

Il existe de nombreux autres événements. Nous entrerons plus en détail sur des événements particuliers dans les prochains chapitres.

Les événements

Gestionnaires d'événements

Pour réagir aux événements, nous pouvons attribuer un gestionnaire – une fonction qui s'exécute en cas d'événement.

Les gestionnaires sont un moyen d'exécuter du code JavaScript en cas d'actions de l'utilisateur.

Il existe plusieurs façons d'attribuer un gestionnaire. Voyons-les, en commençant par le plus simple.

Attribut HTML

Un gestionnaire peut être défini en HTML avec un attribut nommé **on<event>**.

Par exemple, pour attribuer un gestionnaire de "click" à un input, nous pouvons utiliser **onclick**, comme ici :

```
<input value="Cliquer" onclick="alert('Click!')" type="button">
```

En cliquant sur la souris, le code à l'intérieur **onclick** s'exécute.

Veuillez noter qu'à l'intérieur d'*onclick* nous utilisons des guillemets simples, car l'attribut lui-même est entre guillemets doubles. Si nous oublions que le code est à l'intérieur de l'attribut et utilisons des guillemets doubles à l'intérieur, comme ceci : **onclick="alert("Click!")"**, alors cela ne fonctionnera pas correctement.

Les événements

Un attribut HTML n'est pas un endroit pratique pour écrire beaucoup de code, nous ferions donc mieux de créer une fonction JavaScript et de l'appeler ici. Ici un clic exécute la fonction **countRabbits**( ) :

```
<script>
  function countRabbits() {
    for(let i=1; i<=3; i++) {
      alert("Nombre de lapins " + i);
    }
  }
</script>
<input value="Compter les lapins" onclick="countRabbits()" type="button">
```

Comme nous le savons, les noms d'attributs HTML ne sont pas sensibles à la casse, donc **ONCLICK** fonctionnent aussi bien que **onClick** et **onCLICK…** Mais généralement les attributs sont en minuscules : **onclick**.

Propriété DOM

Nous pouvons attribuer un gestionnaire en utilisant une propriété DOM **on<event>**. Par exemple, **elem.onclick**:

Les événements

```
<input value="Cliquer" id="exemple" type="button">
<script>
  exemple.onclick = function() {
    alert("Merci !" );
  }
</script>
```

Si le gestionnaire est affecté à l'aide d'un attribut HTML, le navigateur le lit, crée une nouvelle fonction à partir du contenu de l'attribut et l'écrit dans la propriété DOM.

Cette méthode est donc en fait la même que la précédente.

Ces deux éléments de code fonctionnent de la même manière :

1. **HTML uniquement** :

```
<input value="Cliquer" onclick="alert('Clic !')" type="button">
```

2. **HTML + JS** :

```
<input value="button" id="button" type="button">
<script>button.onclick=function(){ alert("Clic !" );}</script>
```

Les événements

Dans le premier exemple, l'attribut HTML est utilisé pour initialiser le **button.onclick**, tandis que dans le deuxième exemple, c'est le script, *...c'est toute la différence*.

Comme il n'y a qu'une seule propriété **onclick**, nous ne pouvons pas attribuer plus d'un gestionnaire d'événements.

Dans l'exemple ci-dessous, l'ajout d'un gestionnaire avec JavaScript écrase le gestionnaire existant :

```
<input value="Cliquer" id="exemple" type="button" onclick="alert('Avant!')">
<script>
  exemple.onclick = function() {
    alert("Après !" );
  }
</script>
```

Pour supprimer un gestionnaire – attribuez **elem.onclick = null**.

Les événements

Accéder à l'élément : this

La valeur de **this** à l'intérieur d'un gestionnaire est l'élément. Celui sur lequel se trouve le gestionnaire.

Dans le code ci-dessous **button** montre son contenu en utilisant **this.innerHTML**

```
<button type="button" onclick="alert( this.innerHTML )">
Cliquer moi
</button>
```

Erreurs possibles

Si vous commencez à travailler avec des événements, veuillez noter quelques subtilités.

Nous pouvons définir une fonction existante comme gestionnaire :

```
function sayThanks() {
  alert( 'Merci !' );
}
elem.onclick = sayThanks;
```

Mais attention : la fonction doit être appelée comme **sayThanks**, et non comme *sayThanks()*.

Les événements

```
// Bon
button.onclick = sayThanks;
// Mauvais
button.onclick = sayThanks();
```

Si nous ajoutons des parenthèses, **sayThanks()** devient un appel de fonction. Ainsi, la dernière ligne prend en fait le résultat de l'exécution de la fonction, c'est-à-dire *undefined* (car la fonction ne renvoie rien), et l'affecte à onclick. Cela ne fonctionne pas.

...Par contre, dans le balisage nous avons besoin des parenthèses :

```
<input id="button" onclick="sayThanks()" type="button">
```

La différence est facile à expliquer. Lorsque le navigateur lit l'attribut, il crée une fonction de gestionnaire avec un corps à partir du contenu de l'attribut. Le balisage génère donc cette propriété :

```
button.onclick = function(){
   sayThanks();
}
```

Les événements

La casse de la propriété DOM est importante.

Attribuez un gestionnaire à elem.onclick, et non à elem.ONCLICK, car les **propriétés DOM sont sensibles à la casse**.

addEventListener

Le problème fondamental des méthodes d'attribution de gestionnaires mentionnées ci-dessus est que nous ne pouvons pas attribuer plusieurs gestionnaires à un même événement.

Disons qu'une partie de notre code veut mettre en évidence un bouton lors d'un clic, et une autre veut afficher un message lors du même clic.

Nous aimerions attribuer deux gestionnaires d'événements pour cela.

Mais une nouvelle propriété DOM écrasera celle existante :

```
input.onclick = function() { alert(1); }
// ...
input.onclick = function() { alert(2); }
// Remplace le gestionnaire précédent
```

Les événements

Les développeurs de standards Web l'ont compris depuis longtemps et ont suggéré une manière alternative de gérer les gestionnaires en utilisant des méthodes spéciales **addEventListener** et **removeEventListener** qui ne sont pas liés par une telle contrainte.

La syntaxe pour ajouter un gestionnaire :

```
element.addEventListener(event, handler, [options]);
```

- **event :** Nom de l'événement, par exemple "click".
- **handler** : La fonction de gestionnaire.
- **options** : Un objet facultatif supplémentaire avec des propriétés :
  - **once**: si true, alors l'écouteur est automatiquement supprimé après son déclenchement.
  - **capture**: la phase où gérer l'événement, qui sera abordée plus tard dans le chapitre Bouillonnement et capture.
  - **passive**: if true, alors le gestionnaire n'appellera pas **preventDefault**(), nous expliquerons cela plus tard dans Actions par défaut du navigateur.

Les événements

Pour supprimer le gestionnaire, utilisez removeEventListener :

```
element.removeEventListener(event, handler, [options]);
```

Pour certains événements, les gestionnaires travaillent uniquement avec addEventListener

Il existe des événements qui ne peuvent pas être attribués via une propriété DOM, mais seulement avec **addEventListener**.

Par exemple, l'événement **DOMContentLoaded** qui se déclenche lorsque le document est chargé et que le DOM a été construit.

```
// Ne fonctionne pas
document.onDOMContentLoaded=function(){
  alert("DOM chargé");
};
```

```
// Fonctionne
document.addEventListener("DOMContentLoaded",function(){
  alert("DOM chargé");
};
```

Les événements

Plusieurs appels à **addEventListener** permettent d'ajouter plusieurs gestionnaires, comme ceci :

```
<input type="button" value="Cliquer" id="elem">
<script>
 function handler1() {
    alert('Merci !');
 };
 function handler2() {
   alert('Merci encore !');
 }
 elem.onclick = () => alert("Hello");
 elem.addEventListener("click", handler1); // Merci!
 elem.addEventListener("click", handler2); // Merci encore !
</script>
```

Comme nous pouvons le voir dans l'exemple ci-dessus, nous pouvons définir des gestionnaires à la fois en utilisant une propriété DOM et addEventListener. Mais en général, nous n'utilisons qu'une seule de ces méthodes.

Les événements

Objet événement

Pour gérer correctement un événement, nous voudrions en savoir plus sur ce qui s'est passé. Pas seulement un « clic » ou un « appui sur la touche », mais quelles étaient les coordonnées du pointeur ? Quelle touche a été appuyée ? Et ainsi de suite.

Lorsqu'un événement se produit, le navigateur crée un objet *événement* - *event*-, y met des détails et le transmet comme argument au gestionnaire.

Voici un exemple d'obtention des coordonnées d'un pointeur à partir de l'objet événement :

```
<input type="button" value="Cliquer" id="elem">
<script>
  elem.onclick = function(event) {
    // Montre “event type”, element et les coordonnées du click
    alert(event.type + " sur " + event.currentTarget);
    alert("Coordonnées: " + event.clientX + ":" + event.clientY);
  };
</script>
```

Les événements

Quelques propriétés de l'objet "event" :

- **event.type :** Type d'événement, le voici "click".
- **event.currentTarget** : Élément qui a géré l'événement. C'est exactement la même chose que this, à moins que le gestionnaire ne soit une fonction fléchée ou qu'il thissoit lié à quelque chose d'autre, alors nous pouvons obtenir l'élément à partir de event.currentTarget.
- **event.clientX/event.clientY :** Coordonnées relatives à la fenêtre du curseur, pour les événements de pointeur.

Il y a plus de propriétés. Beaucoup d'entre eux dépendent du type d'événement : les événements de clavier ont un ensemble de propriétés, les événements de pointeur – un autre, nous les étudierons plus tard lorsque nous passerons aux détails des différents événements.

Gestionnaires d'objets : handleEvent

Nous pouvons attribuer non seulement une fonction, mais un objet comme gestionnaire d'événements en utilisant **addEventListener**. Lorsqu'un événement se produit, sa méthode **handleEvent** est appelée.

Les événements

```
<button id="elem">Cliquer</button>
<script>
 let obj = {
   handleEvent(event) {
    alert( event.type + " sur " + event.currentTarget );
   }
 elem.addEventListener("click", obj )
</script>
```

Comme nous pouvons le voir, lorsque **addEventListener** reçoit un objet en tant que gestionnaire, il appelle **obj.handleEvent(event)** en cas d'événement.

Le bouillement des évènements

Commençons par un exemple.

Un gestionnaire "handler" est affecté à ** **, mais s'exécute également si vous cliquez sur une balise imbriquée comme **** ou **<code>** :

```
<div onclick="alert('Le handler !')">
  <em>Si vous cliquer sur la balise <code>EM</code>, le gestionnaire sur la balise <code>DIV</code> fonctionne.</em>
</div>
```

N'est-ce pas un peu étrange ? Pourquoi le gestionnaire sur la balise ** ** s'exécute-t-il si le clic réel s'est fait sur la balise **** ?

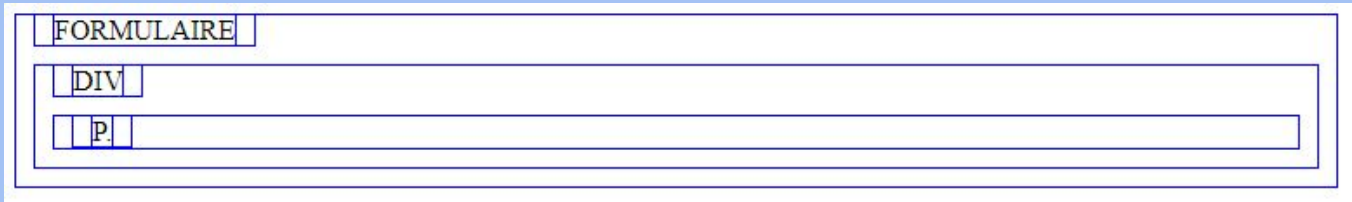
Bouillonnement - bubling -

Le principe du bouillonnement est simple. Lorsqu'un événement se produit sur un élément, il exécute d'abord les gestionnaires sur celui-ci, puis sur son parent, puis jusqu'aux autres ancêtres.

Disons que nous avons 3 éléments imbriqués **FORM** > **DIV** > **P** avec un gestionnaire sur chacun d'eux :

Le bouillement des évènements

```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>
<form onclick="alert('form')">FORM
  <div onclick="alert('div')">DIV
    <p onclick="alert('p')">P</p>
  </div>
</form>
```

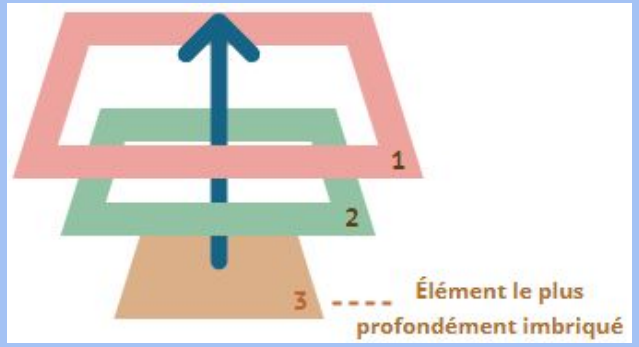


Le bouillement des évènements

Un clic sur le **<p>** premiers lancers intérieurs onclick:

- Sur le **<p>**
- Puis sur la ** **
- Puis sur **<form>**

Et ainsi de suite jusqu'à l'objet document.



Ce processus est appelé « bouillonnement », car les événements « bouillonnent » de l'élément interne vers les parents, comme une bulle dans l'eau.

Presque tous les événements "font des bulles".

Le mot clé de cette phrase est « presque ».

*Par exemple, un événement « **focus** » ne fait pas de bouillonnement. Il y a d'autres exemples, nous les rencontrerons. Mais il s'agit toujours d'une exception, plutôt que d'une règle, la plupart des événements "font des bulles".*

Le bouillement des évènements

event.target

Un gestionnaire sur un élément parent peut toujours obtenir des détails sur l'endroit où l'événement s'est produit.

L'élément le plus profondément imbriqué à l'origine de l'événement est appelé élément cible, accessible sous la forme **event.target.**

Notez les différences par rapport à ceci (**=event.currentTarget**) :

- **event.target** - est l'élément « cible » qui a déclenché l'événement, il ne change pas au cours du processus de bouillonnement.
- **this** (**=event.currentTarget**) - est l'élément « courant », celui sur lequel un gestionnaire est en cours d'exécution.

Par exemple, si nous avons un seul gestionnaire **form.onclick**, il peut « attraper » tous les clics à l'intérieur du formulaire. Quel que soit l'endroit où le clic s'est produit, il remonte jusqu'à <form> et exécute le gestionnaire.

Dans le gestionnaire **form.onclick** :

- **this (=event.currentTarget)** est l'élément **<form>**, car le gestionnaire s'exécute sur lui.
- **event.target** est l'élément sur lequel on a cliqué à l'intérieur du formulaire.

Le bouillement des évènements

```html
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="style.css">
</head>
<body>
  Un clic montre à la fois event.target et this pour les comparer :
  <form id="form">FORM
    <div>DIV
      <p>P</p>
    </div>
  </form>
<script>
form.onclick = function(event) {
  event.target.style.backgroundColor = 'yellow';
  // chrome needs some time to paint yellow
  setTimeout(() => {
    alert("target = " + event.target.tagName + ", this=" + this.tagName);
    event.target.style.backgroundColor = ''
  }, 0);
};
</script>
</body>
</html>
```

```css
form {
  background-color: green;
  position: relative;
  width: 150px;
  height: 150px;
  text-align: center;
  cursor: pointer;
}

div {
  background-color: blue;
  position: absolute;
  top: 25px;
  left: 25px;
  width: 100px;
  height: 100px;
}

p {
  background-color: red;
  position: absolute;
  top: 25px;
  left: 25px;
  width: 50px;
  height: 50px;
  line-height: 50px;
  margin: 0;
}
```

Il est possible que **event.target** corresponde à ceci – cela se produit lorsque le clic est effectué directement sur l'élément **<form>**.

Le bouillement des évènements

Arrêter le bouillonnement

Un événement bouillonnant part de l'élément cible vers le haut.
Normalement, il monte jusqu'à **<html>**, puis l'objet **document**, et certains événements atteignent même **window**, appelant tous les gestionnaires sur le chemin.

Mais n'importe quel gestionnaire peut décider que l'événement a été entièrement traité et arrêter le bouillonnement. La méthode pour cela est **event.stopPropagation()**.

Par exemple, ici **body.onclick** ne fonctionne pas si vous cliquez sur **<button>** :

```
<body onclick="alert(`the bubbling doesn't reach here`)">
  <button onclick="event.stopPropagation()">Cliquer !</button>
</body>
```

Les évènements

Actions par défaut du navigateur

Beaucoup d'évènements mènent à l'exécution d'actions par le navigateur.

Par exemple:

- *Un clic sur un lien* – initie la navigation vers son URL.
- *Un clic sur un bouton d'envoi de formulaire* – initie son envoie vers le serveur.
- *Appuyer sur un bouton de la souris* au-dessus d'un texte et la déplacer – sélectionne le texte.

Si nous gérons un évènement avec JavaScript, nous pouvons ne pas avoir envie de déclencher l'action de navigateur associée, et déclencher un autre comportement à la place.

Empêcher les actions du navigateur

Il y a deux manières de dire au navigateur que nous ne souhaitons pas qu'il agisse:

Les évènements

La manière principale est d'utiliser l'objet **event**. Il y a une méthode **event.preventDefault()**.

Si le gestionnaire d'évènement a été assigné en utilisant **on<event>** (pas par **addEventListener**), alors renvoyer **false** fonctionne de la même manière.

Dans cet HTML un clic sur un lien n'entraine pas une navigation, le navigateur ne fait rien :

```
<a href="/" onclick="return false">Cliquez ici</a>
ou
<a href="/" onclick="event.preventDefault()">ici</a>
```

Exemple: le menu

Considérez le menu d'un site, comme ceci:

```
<ul id="menu" class="menu">
  <li><a href="/html">HTML</a></li>
  <li><a href="/javascript">JavaScript</a></li>
  <li><a href="/css">CSS</a></li>
</ul>
```

Les évènements

Les objets du menu sont implémentés comme des liens HTML <a>, pas des boutons <button>.
Il y a plusieurs raisons pour ceci, par exemple:

- Beaucoup de gens aiment utiliser "clic droit" – "ouvrir dans une nouvelle fenêtre". Si nous utilisons <button> ou , cela ne fonctionne pas.
- Les moteurs de recherche suivent les liens <a href="..."> lors de l'indexation.

Donc nous utilisons <a>. Mais normalement nous avons l'intention de gérer les clics avec JavaScript.
Donc nous devrions empêcher les actions par défaut du navigateur. Comme ci-dessous:

```
menu.onclick = function(event) {
  if (event.target.nodeName != 'A') return;
  let href = event.target.getAttribute('href');
  alert( href ); // ...peut être en chargement depuis le serveur, génération d'UI etc
  return false; // empêche l'action du navigateur (ne va pas sur l'URL)
};
```

Les évènements

Distribution d'événements personnalisés

Nous pouvons non seulement affecter des gestionnaires, mais également générer des événements à partir de JavaScript.

Les événements personnalisés peuvent être utilisés pour créer des "composants graphiques". Par exemple, un élément racine de notre propre menu basé sur JS peut déclencher des événements indiquant ce qui se passe avec le menu: *open* (menu ouvert), *select* (un élément est sélectionné) et ainsi de suite... Un autre code peut écouter les événements et observer ce qui se passe avec le menu.

Nous pouvons générer non seulement des événements complètement nouveaux, que nous inventons pour nos propres besoins, mais aussi des événements intégrés, tels que **click**, **mousedown**, etc.

Cela peut être utile pour les tests automatisés.

Constructeur d'événements

Les classes d'événements intégrées forment une hiérarchie, similaire aux classes d'éléments DOM. La racine est la classe intégrée **Event**.

Les évènements

Nous pouvons créer des objets Event comme ceci:

```
let event = new Event(type[, options]);
```

Arguments:

- **type** – type d'événement, soit une chaîne comme "click" ou la nôtre comme "my-event".
- **options** – l'objet avec deux propriétés facultatives:
  - **bubbles**: true/false – si true, alors l'événement bouillonne.
  - **cancelable**: true/false – si true, alors "l'action par défaut" peut être empêchée.

Par défaut, les deux sont "false": {bubbles: false, cancelable: false}.

dispatchEvent

Après la création d'un objet événement, nous devons le "lancer" sur un élément en utilisant l'appel elem.**dispatchEvent**(event).

Les évènements

Dans l'exemple ci-dessous l'événement **click** est initié avec JavaScript.
Le gestionnaire fonctionne de la même manière que si le bouton était cliqué:

```
<button id="elem" onclick="alert('Click!');">Autoclick</button>
<script>
  let event = new Event("click");
  elem.dispatchEvent(event);
</script>
```

event.isTrusted

*Il existe un moyen de distinguer un événement utilisateur "réel" d'un événement généré par script. La propriété **event.isTrusted** est **true** pour les événements qui proviennent d'actions réelles de l'utilisateur et **false** pour les événements générés par un script.*

Exemple de bouillonnement

Nous pouvons créer un événement qui bouillonne avec le nom "hello" et l'attraper sur document.
Tout ce dont nous avons besoin de faire est de définir l'option **bubles** en tant que **true**:

Les évènements

```
<h1 id="elem">Hello depuis le script!</h1>
<script>
  // attraper sur document...
  document.addEventListener("hello", function(event) { // (1)
    alert("Hello de " + event.target.tagName); // Hello de H1
  });
  // ...distribué sur elem!
  let event = new Event("hello", {bubbles: true}); // (2)
  elem.dispatchEvent(event);
  // le gestionnaire sur le document activera et affichera le message.
</script>
```

Les évènements

Remarques:

Nous devrions utiliser **addEventListener** pour nos événements personnalisés, car **on<event>** n'existe que pour les événements intégrés, **document.onhello** ne fonctionne pas.

Il est essentiel de définir **bubbles: true**, sinon l'événement ne bouillonnera pas.

Le mécanisme de bouillonnement est le même pour les événements intégrés (click) et personnalisés (hello). Il existe également des étapes de capture et de bouillonnement.

MouseEvent, KeyboardEvent et autres

Voici une courte liste de classes pour les événements UI de la spécification:

- UIEvent
- FocusEvent
- MouseEvent
- WheelEvent
- KeyboardEvent
- ...

Les évènements

Le bon constructeur permet de spécifier des propriétés standard pour ce type d'événement.
Comme clientX/clientY pour un événement de souris:

```
let event = new MouseEvent("click", {
  bubbles: true,
  cancelable: true,
  clientX: 100,
  clientY: 100
});
alert(event.clientX); // 100
```

Remarque: le constructeur générique **Event** ne le permet pas. Essayons:

```
let event = new Event("click", {
  bubbles: true,
  cancelable: true,
  clientX: 100,
  clientY: 100
});
alert(event.clientX); // undefined, la propriété inconnue est ignorée!
```

Les évènements

Techniquement, nous pouvons contourner cela en attribuant directement event.clientX=100 après la création. C'est donc une question de commodité et de respect des règles. Les événements générés par le navigateur ont toujours le bon type.

La liste complète des propriétés des différents événements UI se trouve dans la spécification, par exemple, MouseEvent.

Événements personnalisés

Pour nos propres types d'événements comme "hello", nous devrions utiliser new CustomEvent. Techniquement, CustomEvent est identique à Event, à une exception près.

Dans le deuxième argument (objet), nous pouvons ajouter une propriété supplémentaire detail pour toute information personnalisée que nous voulons transmettre avec l'événement.

Les évènements

Par exemple:

```
<h1 id="elem">Hello Jean!</h1>
<script>
  // des détails supplémentaires sont fournis avec l'événement au gestionnaire
  elem.addEventListener("hello", function(event) {
    alert(event.detail.name);
  });
  elem.dispatchEvent( new CustomEvent("hello", {
    detail: { name: "Jean" }
  }));
</script>
```

La propriété **detail** peut avoir n'importe quelle donnée. Techniquement, nous pourrions vivre sans, car nous pouvons attribuer n'importe quelle propriété à un objet *new Event* après sa création.
Mais *CustomEvent* fournit le champ spécial detail pour éviter les conflits avec d'autres propriétés d'événement. De plus, la classe *event* décrit "quel genre d'événement" il s'agit, et si l'événement est personnalisé, alors nous devrions utiliser *CustomEvent* juste pour être clair sur ce que c'est.

Les évènements

event.preventDefault()

De nombreux événements de navigateur ont une "action par défaut", telle que la navigation vers un lien, le démarrage d'une sélection, etc.

Pour les nouveaux événements personnalisés, il n'y a certainement pas d'actions de navigateur par défaut, mais un code qui distribue un tel événement peut avoir ses propres plans pour ce qu'il faut faire après le déclenchement de l'événement.

En appelant **event.preventDefault()**, un gestionnaire d'événements peut envoyer un signal indiquant que ces actions doivent être annulées.

Dans ce cas, l'appel à **elem.dispatchEvent(event)** retourne **false**. Et le code qui l'a envoyé sait qu'il ne devrait pas continuer.

Voyons un exemple pratique – un lapin qui se cache (peut être un menu ou autre chose).

Les évènements

Ci-contre, vous pouvez voir une balise #rabbit et une fonction hide() qui var distribuer l'événement "hide", pour faire savoir à toutes les parties intéressées que le lapin va se cacher.

N'importe quel gestionnaire peut écouter cet événement avec **rabbit.addEventListener**('hide',...) et, si nécessaire, annuler l'action en utilisant **event.preventDefault()**.

Alors le lapin ne disparaîtra pas !

```
<pre id="rabbit">
  |\   /|
   \|_|/
   /. .\
  =\_Y_/=
   {>o<}
</pre>
<button onclick="hide()">Cacher</button>
<script>
  function hide() {
    let event = new CustomEvent("hide", {
      cancelable: true // sans ce marqueur, preventDefault ne fonctionne pas
    });
    if (!rabbit.dispatchEvent(event)) {
      alert('Action empêchée par un handler');
    } else {
      rabbit.hidden = true;
    }
  }
  rabbit.addEventListener('hide', function(event) {
    if (confirm("Appeler preventDefault?")) {
      event.preventDefault();
    }
  });
</script>
```

Les évènements de la souris

Dans ce chapitre, nous verrons plus en détails les événements de la souris et leurs propriétés.

Remarque: Ces événements peuvent provenir non seulement de "périphériques de souris", mais également de périphériques, tels que les téléphones et les tablettes, où ils sont émulés pour des raisons de compatibilité.

Les types d'évènements de Souris

Nous avons déjà vu certains de ces événements :

- **mousedown/mouseup** : Le bouton de la souris est appuyé puis relâché sur un élément.
- **mouseover/mouseout** : Le pointeur de la souris entre ou sort d'un élément.
- **mousemove** : Chaque déplacement de la souris sur un élément déclenche cet évènement.
- **click** : est déclenché après un évènement mousedown et suite à un mouseup sur le même élément, si le bouton gauche de la souris a été utilisé
- **dblclick** : Se déclenche après deux clics sur le même élément dans un court laps de temps. Rarement utilisé de nos jours.

contextmenu

Se déclenche lorsque le bouton droit de la souris est enfoncé. Il existe d'autres façons d'ouvrir un

Les évènements de la souris

- **contextmenu** : Se déclenche lorsque le bouton droit de la souris est enfoncé. Il existe d'autres façons d'ouvrir un menu contextuel, par ex. en utilisant une touche spéciale du clavier, il se déclenche dans ce cas également, donc ce n'est pas exactement l'événement de la souris.

Ordre des événements

Comme vous pouvez le voir dans la liste ci-dessus, une action utilisateur peut déclencher plusieurs événements.

Par exemple, un clic gauche déclenche d'abord **mousedown**, lorsque le bouton est enfoncé, puis **mouseup** et **click** lorsqu'il est relâché.

Au cas où une action unique initialise plusieurs évènements, leur ordre est fixé. C'est-à-dire que les gestionnaires sont appelés dans l'ordre **mousedown** → **mouseup** → **click**.

Bouton de la souris

Les événements liés aux clics ont toujours la propriété **button**, qui permet d'obtenir le bouton exact de la souris.

Les évènements de la souris

D'un autre côté, les gestionnaires mousedown et mouseup peuvent avoir besoin de event.button, car ces événements se déclenchent sur n'importe quel bouton, donc button permet de faire la distinction entre "right-mousedown" et "left-mousedown".

Les valeurs possibles de **event.button** sont :

État du bouton	**event.button**
Bouton gauche (principal)	0
Bouton central (auxiliaire)	1
Bouton droit (secondaire)	2
X1 bouton (arrière)	3
X2 bouton (avant)	4

La plupart des souris n'ont que les boutons gauche et droit, donc les valeurs possibles sont 0 ou 2. Les appareils tactiles génèrent également des événements similaires lorsque l'on appuie dessus.

Il existe également la propriété event.buttons qui a tous les boutons actuellement pressés sous forme d'entier, un bit par bouton. En pratique cette propriété est très rarement utilisée, vous pouvez trouver des détails sur MDN si jamais vous en avez besoin.

Les évènements de la souris

Les Touches de Modifications: shift, alt, ctrl and meta

Tous les évènements de la souris contiennent des informations à propos des touches de modifications qui sont appuyées.

Propriétés d'événement :

- **shiftKey**: Shift
- **altKey**: Alt (or Opt for Mac)
- **ctrlKey**: Ctrl
- **metaKey**: Cmd for Mac

Ils sont true si la touche correspondante fut appuyée durant l'évènement. Par exemple le bouton ci-contre fonctionne seulement avec **Alt+Shift+click**:

```
<button id="button">Alt+Shift+Clique moi !</button>
<script>
  button.onclick = function(event) {
    if (event.altKey && event.shiftKey) {
      alert('Hello click !');
    }
  };
</script>
```

Les évènements de la souris

Attention : Sur Mac c'est souvent Cmd au lieu de Ctrl

Sous Windows et Linux, il y a des touches modificatrices Alt, Shift et Ctrl. Sur Mac, il y en a une en plus : **Cmd**, correspondant à la propriété **metaKey**.

Dans la plupart des applications, lorsque Windows / Linux utilise Ctrl, sur Mac Cmd est utilisée. C'est-à-dire : lorsqu'un utilisateur Windows appuie sur Ctrl+Enter ou Ctrl+A, un utilisateur Mac presserait sur Cmd+Enter ou Cmd+A, etc.

Donc, si nous voulons supporter des combinaisons comme Ctrl+click, alors pour Mac, il est logique d'utiliser Cmd+click. C'est plus confortable pour les utilisateurs de Mac.

Même si nous aimerions forcer les utilisateurs de Mac à Ctrl+click – c'est un peu difficile. Le problème est: un clic gauche avec Ctrl est interprété comme un clic droit sur MacOS, et il génère l'évènement menu contextuel, et non un click comme sur Windows/Linux.

Donc, si nous voulons que les utilisateurs de tous les systèmes d'exploitation se sentent à l'aise, alors avec ctrlKey nous devrions vérifier la metaKey.

Pour JS-code cela signifie que nous devons contrôler si if (event.ctrlKey || event.metaKey).

Les évènements de la souris

Il y a aussi les appareils mobiles

Les combinaisons de clavier sont un bon complément au flux de travail. Donc, si le visiteur utilise un clavier – ils fonctionnent.

Mais si leur appareil n'en a pas, il devrait y avoir un moyen de vivre sans touches de modification.

Cordonnées: clientX/Y, pageX/Y

Tous les événements de souris fournissent des coordonnées de deux manières :

Nous avons déjà couvert la différence entre eux dans le chapitre Coordonnées.

En résumé, les coordonnées relatives au document pageX/Y sont comptées à partir du coin supérieur gauche du document, et ne changent pas lorsque la page défile, tandis que clientX/Y sont comptées à partir du coin supérieur gauche de la fenêtre actuelle . Lorsque la page défile, ils changent.

Par exemple, si nous avons une fenêtre de taille 500x500 et que la souris est dans le coin supérieur gauche, alors clientX et clientY sont 0, peu importe comment la page est défilée.

Et si la souris est au centre, alors clientX et clientY sont 250, quelle que soit la place dans le document. Ils sont similaires à position:fixed dans cet aspect.

Les évènements de la souris

Déplacez la souris sur le champ de saisie pour voir clientX/clientY (l'exemple est dans l iframe, ainsi les cordonnées sont relatives à cet iframe) :

```
<input onmousemove="this.value=event.clientX+':'+event.clientY" value="Passer la souris !">
```

Les coordonnées relatives au document pageX, pageY sont comptées à partir du coin supérieur gauche du document, pas de la fenêtre.

Empêcher la sélection sur le mousedown

Le double clic de souris a un effet secondaire qui peut être dérangeant dans certaines interfaces: il sélectionne du texte.

Par exemple, double-cliquer sur le texte ci-dessous le sélectionne en plus de notre gestionnaire :

```
<span ondblclick="alert('dblclick')">Double-cliquez !</span>
```

Les évènements de la souris

Si on appuie sur le bouton gauche de la souris et, sans le relâcher, on déplace la souris, la sélection devient alors souvent indésirable.
Dans ce cas particulier, le moyen le plus raisonnable consiste à empêcher l'action du navigateur lors du mousedown. Il empêche ces deux sélections :

```
Avant...
<strong ondblclick="alert('Click!')" onmousedown="return false">
   Double-clicquer moi
</strong>
...Apres
```

Ainsi l'élément "strong" n'est pas sélectionné lors d'un double-clic, et si vous appuyez sur le bouton gauche de la souris, la sélection ne sera pas lancée.

Remarque: le texte à l'intérieur est toujours sélectionnable. Cependant, la sélection ne doit pas commencer sur le texte lui-même, mais avant ou après. Cela convient généralement aux utilisateurs.

Les évènements de la souris

Prévenir la copie

Si nous voulons désactiver la sélection pour protéger le contenu de notre page du copier-coller, nous pouvons utiliser un autre événement : **oncopy**.

```
<div oncopy="alert('Copier est interdit !');return false">
Cher Utilisateur
Il vous est interdit de faire du copier-coller.
Si vous connaissez JS ou HTML, alors vous pouvez tout obtenir à partir
de la page source néanmoins.
</div>
```

Si vous essayer de copier une partie de texte dans un , cela ne va pas fonctionner, parce que l'action par défaut oncopy est empêchée.

Certes, l'utilisateur a accès à la source HTML de la page et peut en extraire le contenu, mais tout le monde ne sait pas comment le faire.

Les évènements de la souris

A l'inverse du **mouseout**:

- **event.target** – est l'élément que la souris a quitté.
- **event.relatedTarget** – est le nouvel élément situé sous le pointeur, celui pour lequel la souris a quitté (**target** → **relatedTarget**).

Dans l'exemple ci-contre chaque aspect facial est un élément. Lorsque vous déplacez la souris, vous pouvez voir les évènements de souris dans la zone de texte.
Chaque événement contient les informations sur **target** et **relatedTarget** :
Télécharger le zip "***smiley.zip***" pour visualiser cet exemple.

relatedTarget peut être null

La propriété relatedTarget peut être null. C'est normal et cela signifie simplement que la souris de provient pas d'un autre élément, mais hors de la fenêtre Windows. Ou bien qu'elle a quitté la fenêtre Windows. Nous devons garder cette éventualité à l'esprit lorsqu'on utilise event.relatedTarget dans notre code. Si nous accédons a la propriété event.relatedTarget.tagName, alors il y aura une erreur.

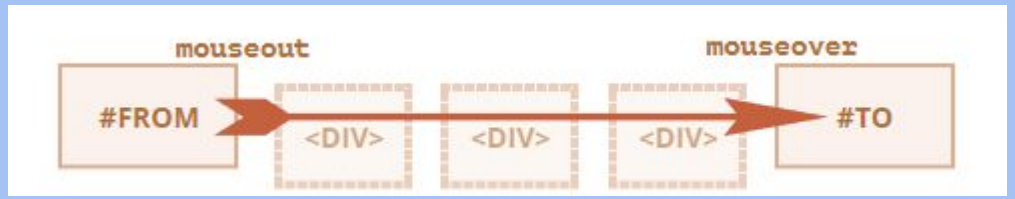
Les évènements de la souris

Ignorer des éléments

L'évènement **mousemove** se déclenche lorsque la souris se déplace. Mais cela ne signifie pas chaque pixel mène à un évènement.

Le navigateur vérifie la position de la souris de temps en temps. Et s'il remarque des changements, déclenche les événements.

Cela signifie que si le visiteur déplace la souris très rapidement, certains éléments DOM peuvent être ignorés :



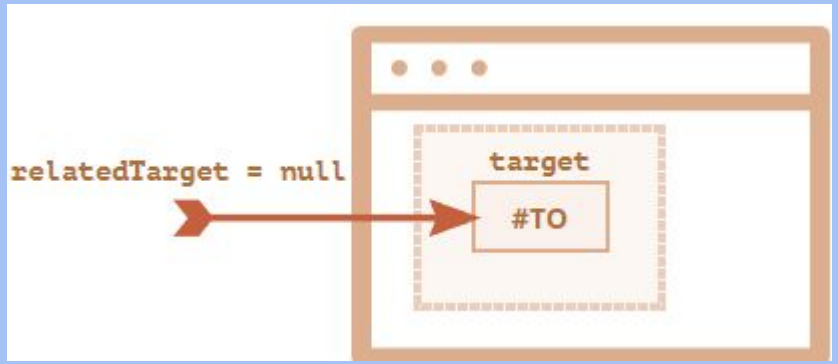
Si la souris se déplace très rapidement de #FROM aux éléments #TO telle que décrite en haut, alors le intermédiaire (ou certains d'entre eux) peuvent être sautés. L'évènement mouseout peut être déclenche sur #FROM et ensuite immédiatement le mouseover sur #TO.

Les évènements de la souris

C'est bon pour la performance, car s'il peut y avoir beaucoup d'éléments intermédiaires. Nous ne voulons pas vraiment traiter dans et hors de chacun d'entre eux.

D'autre part, nous devons garder à l'esprit que le pointeur de la souris ne "visite" pas tous les éléments le long du chemin. Il peut "sauter".

En particulier, il est possible que le pointeur saute directement au centre de la page depuis l'extérieur de la fenêtre. Dans ce cas relatedTarget est null, parce qu'il venait de "nulle part" :



Les évènements Glisser-Déposer de la souris

Drag'n'Drop est une excellente solution d'interface. Prendre quelque chose et le faire glisser est un moyen clair et simple de faire beaucoup de choses, de la copie et du déplacement de documents (comme dans les gestionnaires de fichiers) à la commande (déposer des articles dans un panier).
Dans la norme HTML moderne, il y a une section à propos du Drag and Drop avec des événements spéciaux tels que **dragstart**, **dragend**, etc.
Ces événements nous permettent de prendre en charge des types spéciaux de glisser-déposer, tels que la gestion du glisser-déposer d'un fichier depuis le gestionnaire de fichiers du système d'exploitation et le déposer dans la fenêtre du navigateur. Ensuite, JavaScript peut accéder au contenu de ces fichiers.

Mais les événements de drag natifs ont également des limites. Par exemple, nous ne pouvons pas empêcher de faire glisser depuis une certaine zone. Nous ne pouvons pas non plus faire de glisser "horizontal" ou "vertical" uniquement. Et il existe de nombreuses autres tâches de glisser-déposer qui ne peuvent pas être effectuées en les utilisant. En outre, la prise en charge des appareils mobiles pour de tels événements est très faible.

Les évènements Glisser-Déposer de la souris

Ici, nous verrons comment implémenter le glisser-déposer à l'aide d'événements de souris.

l'algorithme Drag'and'Drop

- Sur **mousedown** – préparer l'élément pour le déplacement, si nécessaire (éventuellement en créer une copie, y ajouter une classe ou autre).
- Puis sur **mousemove** le déplacer en changeant left/top et position:absolute.
- Sur **mouseup** – effectue toutes les actions liées à un Drag'n'Drop terminé.

Voici la mise en œuvre de faire glisser une balle :

```
<img src="ball.svg" style="cursor: pointer; position: absolute; z-index:
1000; left: 314px; top: 7px;" width="40" height="40" id="ball">
```

Les évènements Glisser-Déposer de la souris

```
ball.onmousedown = function(event) {
  // (1) la préparer au déplacement :  réglé en absolute et en haut par z-index
  ball.style.position = 'absolute';
  ball.style.zIndex = 1000;
  // déplacez-le directement dans body pour le placer par rapport à body
  document.body.append(ball);
  // Centrer la balle aux coordonnées (pageX, pageY)
  function moveAt(pageX, pageY) {
    ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
    ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
  }
  // déplacer notre balle en positionnement absolu sous le pointeur
  moveAt(event.pageX, event.pageY);
  function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
  }
  // (2) déplacer la balle sur le déplacement de la souris
  document.addEventListener('mousemove', onMouseMove);
  // (3) laisser tomber la balle, retirer les gestionnaires inutiles
  ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
  };
};
```

Les évènements Glisser-Déposer de la souris

Si nous exécutons le code, nous pouvons remarquer quelque chose d'étrange. Au début de l'action Glisser-Déposer, la balle est prise en "fourchette": Nous commençons à faire glisser son "clone". Essayez de faire glisser-déposer avec la souris et vous verrez un tel comportement.

C'est parce que le navigateur a son propre Glisser-Déposer pour les images et quelques autres éléments qui s'exécute automatiquement et entre en conflit avec le nôtre.

Pour le désactiver:

```
ball.ondragstart = function() {
  return false;
};
```

Un autre aspect important—nous suivons l'évènement mousemove sur le document, pas sur la balle. A première vue, il semblerait que la souris soit toujours sur la balle, et nous pouvons lui appliquer l' évènement mousemove.

Mais si nous nous rappelons, l'évènement mousemove se déclenche souvent, mais pas sur chaque pixel. Donc après un mouvement rapide, le curseur peut sauter de la balle pour aller quelque part au milieu du document (ou bien même hors de la fenêtre). Donc nous devons écouter le document pour le capturer.

Les évènements Glisser-Déposer de la souris

Positionnent correcte

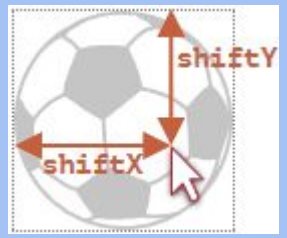
Dans les exemples ci-dessus la balle est toujours déplacée ainsi, de sorte à ce que son centre soit au-dessous du curseur:

```
ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
```

Pas mal, mais il y a un effet secondaire. Pour initier le "Glisser-Déposer", nous pouvons appliquer un mousedown n'importe où sur la balle. Mais si nous le faisons sur le rebord, alors la balle "saute" soudainement pour être centrée sous le curseur.

Ce serait mieux si nous gardions le changement initial de l'élément relativement au curseur.

Par exemple, si nous commençons le glissement par le rebord de la balle, alors le curseur doit rester sur le rebord pendant le déplacement.



Les évènements Glisser-Déposer de la souris

1. Lorsqu'un visiteur appuye sur le bouton (mousedown) – nous pouvons garder en mémoire la distance du curseur au coin gauche en haut de la balle dans des variables shiftX/shiftY. Nous devons garder cette distance en faisant le glissement. Pour obtenir ces changements nous pouvons soustraire les coordonnées:

```
// onmousedown
let shiftX = event.clientX - ball.getBoundingClientRect().left;
let shiftY = event.clientY - ball.getBoundingClientRect().top;
```

2. Ensuite pendant qu'on fait le glissement nous positionnons la balle sur le même changement relativement au curseur, ainsi:

```
// onmousemove - la balle a une position:absolute
ball.style.left = event.pageX - shiftX + 'px';
ball.style.top = event.pageY - shiftY + 'px';
```

Le code final avec un meilleur positionnement:

Les évènements Glisser-Déposer de la souris

```
ball.onmousedown = function(event) {
  let shiftX = event.clientX - ball.getBoundingClientRect().left;
  let shiftY = event.clientY - ball.getBoundingClientRect().top;
  ball.style.position = 'absolute';
  ball.style.zIndex = 1000;
  document.body.append(ball);
  function moveAt(pageX, pageY) {
    ball.style.left = pageX - shiftX + 'px';
    ball.style.top = pageY - shiftY + 'px';
  }
  moveAt(event.pageX, event.pageY);
  function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
  }
  document.addEventListener('mousemove', onMouseMove);
  ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
  };
  ball.ondragstart = function() {
   return false;
  };
};
```

Les évènements Glisser-Déposer de la souris

Potentiels Cibles pour un Déposer (déposables)

Dans les exemples précédents la balle pouvait être déposée juste "n'importe où" pour qu'elle y soit. En réalité, nous prenons souvent un élément et le déposons sur un autre. Par exemple, un fichier dans un dossier, ou autre chose.

En d'autres termes, nous prenons un élément "déplaçable" et le déposons sur un élément ou l'on peut déposer un élément "déposable".

Nous avons besoin de savoir :

- où l'élément a été déposé à la fin du glisser-déposer – pour effectuer l'action correspondante,
- et, de préférence, connaitre l'élément droppable que nous déplaçons pour le mettre en surbrillance.

Quelle peut être la première idée ? Probablement pour définir des gestionnaires de mouseover/mouseup sur des "droppables" potentiels ?

Mais cela ne marche pas.

Les évènements Glisser-Déposer de la souris

Le problème est que, pendant que nous exécutons le déplacement, l'élément déplaçable est toujours sur les autres éléments. Et les évènements de la souris ont lieu seulement sur l'élément, pas sur ceux se trouvant au-dessous de ce dernier.

Par exemple, ci-dessous deux éléments ** **, un en rouge au-dessus d'un autre en bleu (couvert complètement). Il n'y a aucun moyen de capturer un évènement sur celui en bleu, parce que le rouge se trouve au-dessus de celui-ci:

```
<style>
  div { width: 50px; height: 50px; position: absolute; top: 0; }
</style>
<div style="background:blue" onmouseover="alert(Ne marche jamais')"></div>
<div style="background:red" onmouseover="alert('sur le rouge!')"></div>
```

De la même manière avec l'élément déplaçable, la balle est toujours au-dessus des autres éléments, donc les évènements s'exécutent sur lui. Quels que soient les gestionnaires que l'on assigne aux éléments se trouvant en bas, ils ne vont pas fonctionner. C'est pourquoi l'idée initiale de mettre les gestionnaires sur des potentiels objets déposables ne fonctionne pas en pratique. Ils ne vont pas démarrer.

Les évènements Glisser-Déposer de la souris

Alors, que faire?

Il existe une méthode appelée document.elementFromPoint(clientX, clientY). Elle retourne l'élément le plus imbriqué sur les coordonnées données relatif à une fenêtre (ou bien null si les coordonnées données sont hors de la fenêtre). S'il y a plusieurs éléments qui se chevauchent sur les mêmes coordonnées, le plus haut est renvoyé. Nous pouvons l'utiliser dans n'importe lequel de nos gestionnaires d'événements à la souris pour détecter le droppable potentiel sous le pointeur, comme ceci :

```
// dans un gestionnaire d'événements de souris
ball.hidden = true; // (*) hide the element that we drag
let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
// elemBelow est l'élément situé sous la balle, peut être droppable
ball.hidden = false;
```

Remarque: nous devons cacher la balle avant l'appel (*). Sinon, nous aurons généralement une balle sur ces coordonnées, car c'est l'élément supérieur sous le pointeur : elemBelow=ball. Donc, nous le cachons et montrons immédiatement à nouveau.

Nous pouvons utiliser ce code pour voir quel élément nous sommes entrain de "survoler" à n'importe quel moment. Et gérer le déposer lorsque cela survient.

Le Clavier: les évènements keydown et keyup

Avant que nous en arrivions au clavier, veuillez noter que sur des appareils modernes il y a d'autres manières de "récupérer quelque chose". Par exemple, les gens utilisent la reconnaissance vocale (en particulier sur les appareils mobiles) oubien le copier/coller avec la souris.

Donc si nous voulons contrôler une entrée dans un champ <input>, alors les évènements du clavier ne sont pas assez suffisants. Il y a un autre évènement nommé input pour gérer les changements d'un champ <input>, par n'importe quelle moyen. Et il peut être un meilleur choix pour une telle tâche. Nous allons traiter cela plus tard dans le chapitre Les événements: change, input, cut, copy, paste.

Les évènements du clavier doivent être utilisés lorsqu'on veut gérer les actions sur le clavier (Le clavier virtuel compte aussi). Par exemple, pour réagir sur les touches de directions Up et Down oubien les touches de raccourcis (y compris les combinaisons de touches)..

Télécharger la démo sur le Github : **keys.zip**

Le Clavier: les évènements keydown et keyup

Keydown et keyup

Les évènements keydown surviennent lorsqu'une touche est appuyée, et ensuite intervient keyup – lorsqu'elle est relâchée.

event.code et event.key

La propriété key de l'objet évènement permet d'obtenir un caractère, tandis que la propriété code de l'objet évènement objet permet d'obtenir le "code de la touche physique".

Par exemple, la même touche Z peut être appuyée avec ou sans Shift. Cela nous donne deux caractères différents : minuscule z et majuscule Z.

La propriété **event.key** est exactement le caractère, et il sera différent.
Cependant **event.code** est la même:

Le Clavier: les évènements keydown et keyup

key	event.key	event.code
Z	z (minuscule)	KeyZ
Shift+Z	Z (majuscule)	KeyZ

Si un utilisateur travaille avec des langues différentes, alors le fait de changer vers une autre langue aura pour effet de créer un caractère totalement différent de "Z". Cela va devenir la valeur de event.key, tandis que event.code est toujours la même que: "KeyZ".

"KeyZ" et autres codes de touches

Chaque touche a un code qui dépend de sa position sur le clavier. Les codes des touches sont décrits dans le document Spécification des codes des évènements Interfaces Utilisateurs.

Le Clavier: les évènements keydown et keyup

Par exemple:

- Les touches des lettres ont des codes de type: "Key<letter>": "KeyA", "KeyB" etc.
- Les touches numériques ont des codes de type: "Digit<number>": "Digit0", "Digit1" etc.
- Les touches spéciales sont codées par leurs noms: "Enter", "Backspace", "Tab" etc.

Il existe plusieurs formats de claviers usuels, différents de par la présentation, et la spécification donne des codes pour les touches pour chacun d'entre eux.

Et si une touche ne donne aucun caractère ? Par exemple, Shift ou F1 ou autres. Pour ces touches, event.key est approximativement la même chose que event.code :

Key	event.key	event.code
F1	F1	F1
Backspace	Backspace	Backspace
Shift	Shift	ShiftRight or ShiftLeft

Le Clavier: les évènements keydown et keyup

Veuillez noter que **event.code** spécifie exactement quelle touche est appuyée. Par exemple, la plupart des claviers ont deux touches de Shift: à gauche et à droite. La propriété **event.code** nous dit exactement laquelle fut appuyée, et **event.key** est responsable de la " signification" de la touche: comment il s'agit d'un ("Shift").
Disons, nous voulons gérer un raccourci : Ctrl+Z (ou Cmd+Z pour Mac). La plupart des éditeurs accrochent l'action du "Défaire" sur cette touche. Nous pouvons mettre un écouteur lorsqu'on déclenche l'évènement keydown et chercher à savoir quelle touche est appuyée.

Il existe un dilemme ici: Dans cet écouteur d'évènement, devons-nous contrôler la valeur de event.key ou bien d'event.code?

D'une part, la valeur de event.key est un caractère, elle change en fonction de la langue. Si le visiteur a plusieurs langues dans le système d'exploitation et bascule entre elles, la même clé donne des caractères différents. Il est donc logique de vérifier event.code, c'est toujours pareil.

Le Clavier: les évènements keydown et keyup

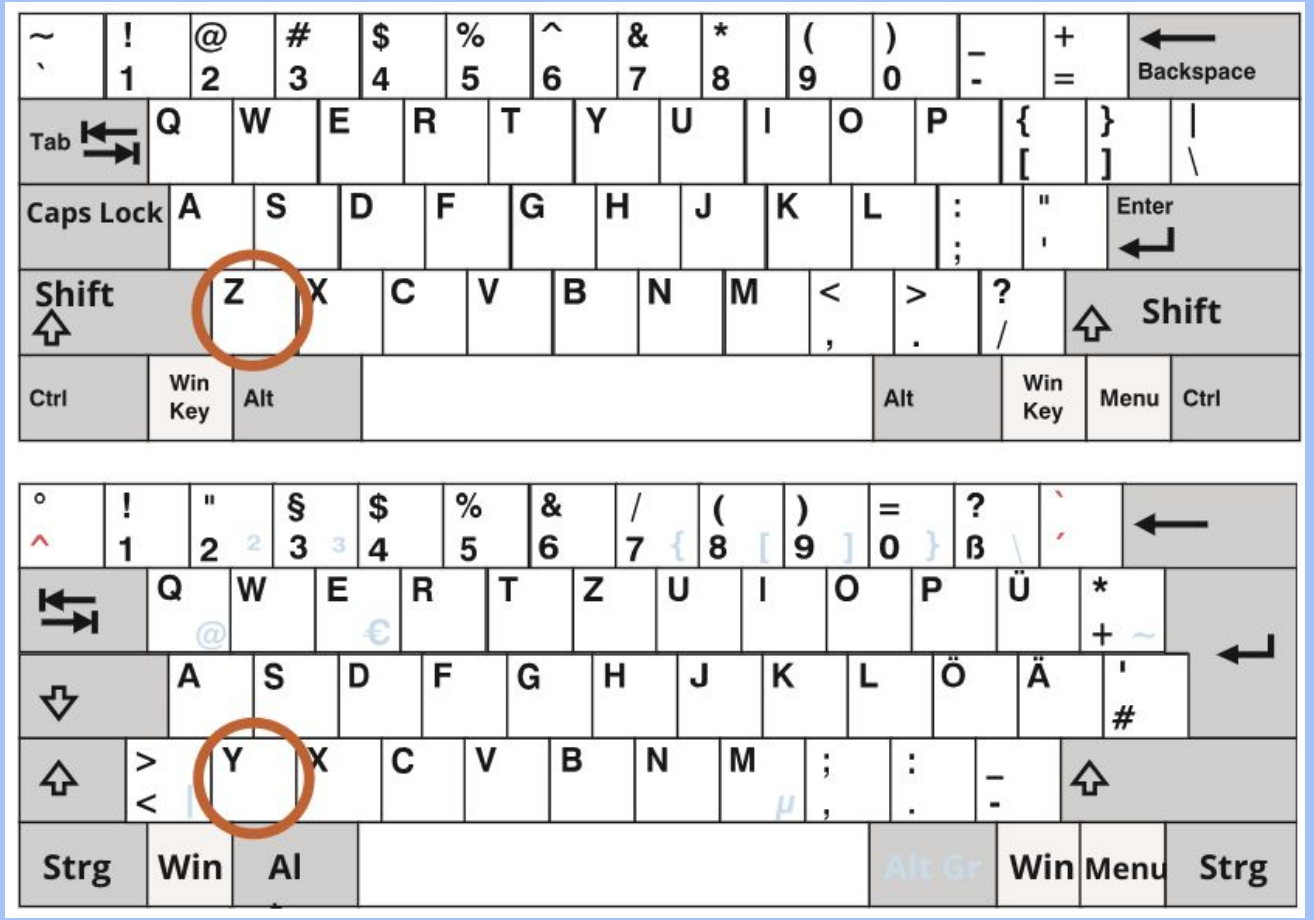
Ainsi :

```
document.addEventListener('keydown', function(event) {
  if (event.code == 'KeyZ' && (event.ctrlKey || event.metaKey)) {
    alert('Undo!')
  }
});
```

D'autre part, il existe un problème avec event.code. Pour des dispositions de clavier différentes, la même touche peut avoir des caractères différents.

Par exemple, voici la disposition du clavier Américain ("QWERTY") et Allemand ("QWERTZ") dessous (de Wikipedia) :

Le Clavier: les évènements keydown et keyup



Le Clavier: les évènements keydown et keyup

Pour la même touche, le clavier Américain a un "Z", tandis que celui Allemand a un "Y" (les lettres sont échanges).
Donc, event.code sera égal à KeyZ pour les gens utilisant le clavier Allemand lorsqu'ils appuient sur Y.
Si on vérifie event.code == 'KeyZ' dans notre code, alors pour les personnes avec la disposition allemande ce genre de test passera quand ils appuient sur Y.
Cela semble être bizarre, mais c'est ainsi. La specification mentionne explicitement ce comportement.

Donc, event.code peut correspondre à un caractère incorrect pour une disposition inattendue. Les mêmes lettres dans différentes disposition peuvent mapper sur différentes clés physiques, conduisant à des codes différents. Heureusement, cela ne se produit qu'avec plusieurs codes, par exemple keyA, keyQ, keyZ (comme nous l'avons vu), et ne se produit pas avec des touches spéciales telles que Shift. Vous pouvez trouver la liste dans la specification.

Pour suivre de manière fiable les caractères dépendant de la disposition, event.key peut être un meilleur moyen.

Le Clavier: les évènements keydown et keyup

D'un autre côté, event.code a l'avantage de rester toujours le même, lié à l'emplacement de la clé physique. Ainsi, les raccourcis clavier qui en dépendent fonctionnent bien même en cas de changement de langue.
Voulons-nous gérer des clés dépendantes de la disposition ? Alors event.key est la voie à suivre.
Ou voulons-nous un raccourci clavier même après un changement de langue ? Alors, event.code peut être une meilleure option.

Auto-repeat

Si une touche est appuyée assez longtemps, elle commence la répétition avec la propriété "auto-repeat": l'évènement keydown se déclenche de manière répétitive, et ensuite lorsqu'elle est relâchée nous obtenons finalement l'évènement keyup. Donc c'est normal d'avoir plusieurs keydown et un unique évènement keyup.

Pour les évènements déclenchés par auto-repeat, l'évènement objet a une propriété event.repeat dont la valeur est assignée à true.

Le Clavier: les évènements keydown et keyup

Actions par défaut

Les actions par défaut varient, comme il y a beaucoup de possibilités pouvant être initiées par le clavier.

Pa exemple:

- Un caractère apparait sur l'écran (le résultat le plus évident).
- Un caractère est supprimé (Delete key).
- Une page est défilée (PageDown key).
- Le navigateur ouvre la boite de dialogue "Sauvegarder la Page" dialog (Ctrl+S)

...ainsi de suite.

L'empêchement de l'action par défaut à l'évènement keydown peut annuler la plupart d'entre elles, à l'exception des touches spéciales spécifiques aux systèmes d'exploitations. Par exemple, sur Windows Alt+F4 ferme la fenêtre en cours du navigateur. Et il n'existe aucun moyen de l'arrêter en empêchant l'action par défaut JavaScript.

Le Clavier: les évènements keydown et keyup

Par exemple, la balise <input> ci dessous s'attend à recevoir un numéro de téléphone, alors elle n'accepte que les touches numériques, +, () oubien -:

```
<script>
function checkPhoneKey(key) {
  return (key >= '0' && key <= '9') ||
    ['+','(',')','-','ArrowLeft','ArrowRight','Delete','Backspace'].includes(key);
}
</script>
<input onkeydown="return checkPhoneKey(event.key)" placeholder="Phone, please" type="tel">
```

Maintenant les flèches et la suppression marchent bien.

Même si nous avons le filtre de clavier, on peut toujours entrer n'importe quoi à l'aide d'une souris et faire un clic droit + Coller. Les appareils mobiles offrent d'autres moyens de saisir des valeurs. Le filtre n'est donc pas fiable à 100%.

L'approche alternative serait de suivre l'événement oninput – il se déclenche après toute modification. Là, nous pouvons vérifier le nouveau input.value et le modifier/mettre en surbrillance le <input> lorsqu'il est invalide. Ou nous pouvons utiliser les deux gestionnaires d'événements ensemble.