ceviche.js

Capítulo 3: DOM y CSSOM

> *Luego de haber visto las bases del lenguaje, es momento de conocer más del navegador y aprender a crear y modificar la interfaz de usuario. Existen dos APIs en el navegador que permiten manipular la estructura, contenido y presentación visual de lo que se muestra dentro de un navegador: el Document Object Model, o DOM, y el Cascade Style Sheet Object Model, o CSSOM.*

DOM

El Document Object Model, o DOM, es una API para documentos HTML que representa cada elemento de una página web en forma de objetos, permitiendo su manipulación para cambiar tanto la estructura como presentación visual. De igual forma, permite manejar eventos del usuario dentro del navegador.

HTML

HyperText Markup Language, o HTML, es un lenguaje de marcado que permite definir la estructura y contenido de un documento mediante el uso de etiquetas. El proceso de convertir un documento en HTML en una estructura visual es denominado **renderizar**, y es el **motor de renderizado** el encargado de realizar esta acción.

Es este motor de renderizado el que, a su vez, se encarga de utilizar las hojas de estilo en cascada (CSS) para darle la presentación adecuada al documento HTML que se está renderizando. Actualmente, los motores de renderizado más populares son:

- Webkit, utilizado en Safari y Chrome hasta su versión 27.
- Gecko, utilizado por Firefox y los productos de la Fundación Mozilla.

- Blink (*fork* de Webkit), utilizado actualmente por Chrome a partir de su versión 28.
- Presto, utilizado por Opera, que luego pasó a utilizar Blink.
- Trident, utilizado principalmente por Internet Explorer y otros productos de Microsoft.

Nodos y Elementos

La abstracción que el DOM realiza de un documento HTML utiliza el concepto de **árbol de nodos** para representar la estructura de elementos anidados que tiene el documento. Esto quiere decir que un elemento (o nodo en el árbol) puede tener elementos anidados dentro del mismo (denominados *nodos hijos*); al tener nodos hijos, este elemento automáticamente se convierte en un nodo padre. El primer nodo de un árbol, es decir, aquel que tenga nodos hijos pero no es hijo de ningún otro nodo, es llamado **nodo raíz**.

Según la tabla de valores de la propiedad `nodeType` (ver Apéndice A), se pueden ver diferentes tipos de nodos. Al ser el DOM una abstracción basada en árboles de nodos, cada dato dentro de un documento HTML debe pertenecer a un tipo de nodo. Por ejemplo:

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div class="empty-div"></div>
  <!-- Comentario dentro de un documento HTML -->
  Contenido de texto
</body>
</html>
```

`nodeType`	Tipo de dato en HTML
`ELEMENT_NODE`	Elemento (`<div class="empty-div"></div>`)

`nodeType`	**Tipo de dato en HTML**
`ATTRIBUTE_NODE`	Atributo (`class="empty-div"`)
`TEXT_NODE`	Texto (`Contenido de texto`)
`COMMENT_NODE`	Comentario (`<!-- Comentario dentro de un documento HTML -->`)
`DOCUMENT_NODE`	Documento (`window.document`)
`DOCUMENT_TYPE_NODE`	Doctype (`<!DOCTYPE html>`)

Uno de estos tipos de nodos es el elemento (`ELEMENT_NODE`), el cual es la representación para toda etiqueta en un documento HTML. Esto quiere decir que todos los elementos son nodos, pero no todos los nodos son elementos.

`window` y `document`

El contexto global de una aplicación web recae en `window`, el cual contiene referencias a diferentes APIs y objetos del navegador, como `screen`, `navigator`, `history`, `location` y `document`. Cada iframe tiene su propio objeto `window`, y pueden acceder al `window` que lo contiene mediante la propiedad `parent`.

`document` es el objeto que representa al nodo raíz de un documento HTML, y tiene acceso a los nodos que representan a las etiquetas `<head>` y `<body>`. Los iframes tienen su propio documento HTML, por lo que, si un documento tiene iframes, cada elemento `iframe` puede acceder a su propio documento mediante la propiedad `contentDocument`.

Las propiedades y métodos de `document` están definidas por dos interfaces: `HTMLDocument` y `Document`. Adicionalmente, `Document` hereda de `Node`. Mientras `HTMLDocument` y `Document` le dan a `document` la capacidad de representar al nodo raíz de un documento, `Node` añade propiedades y métodos relacionados al manejo de nodos. Tanto `window` y `document` heredan de la interfaz `EventTarget`, por lo que tienen la capacidad de manejar eventos.

El DOM en su forma más abstracta es una interfaz que permite leer y manipular documentos en XML y HTML (incluyendo XHTML) y representarlos como un árbol de nodos; sin embargo, el tipo documento estándar mostrado en un

navegador es HTML, el cual tiene propiedades propias y distintas a un documento XML cualquiera. Es por eso que, en el navegador, el DOM tiene dos interfaces para representar a un documento: `Document` para un documento genérico, y `HTMLDocument` para un documento HTML.

`HTMLDocument` tiene propiedades propias de un documento HTML, como: `domain`, `title`, `body`, `forms`, `anchors`, `links` e `images`. Cabe aclarar que `anchors` y `links` devuelven listas de elementos de la misma etiqueta (`<a>`). Esto sucede porque esta etiqueta es usada tanto como ancla dentro del documento como para enlazar el documento actual a otros documentos, imágenes, archivos, etc.

`Document` tiene 3 propiedades: `doctype`, que devuelve el tipo de documento (DTD); `documentElement`, que representa a la etiqueta `html`; e `implementation`, que permite crear documentos (HTML o no), así como hojas de estilos. Así mismo, tiene una serie de métodos útiles para la manipulación de elementos dentro del documento, como crear nodos de tipo elemento, comentario, texto y atributo.

Agregando y eliminando nodos

Agregar un nodo en el árbol DOM consta de 2 pasos: Crear el nodo, y añadirlo. El primer paso utiliza un método de `document` que varía según el tipo de nodo que se desee agregar:

- Para agregar un nodo elemento se utiliza `document.createElement`.
- Para agregar un nodo atributo se utiliza `document.createAttribute`.
- Para agregar un nodo texto se utiliza `document.createTextNode`.
- Para agregar un nodo comentario se utiliza `document.createComment`.

Para el segundo paso, el futuro nodo padre debe ejecutar el método `appendChild`.

Para eliminar un nodo solo es necesario que el nodo padre ejecute el método `removeChild`.

Simplificando el manejo del DOM con `dom.js`

Como vimos en el punto anterior, tener que realizar 2 pasos para agregar un

nodo al DOM puede llegar a ser tedioso (sobre todo si tenemos que hacerlo varias veces). Adicionalmente, pronto notarás que la API del DOM es verbosa, por lo que sería una buena idea crear una biblioteca que permita reducir el número de palabras escritas y simplifique los pasos para manejar el DOM; así que realizaremos una biblioteca llamada `dom.js`, la cual usaremos dentro de La Buena Espina.

Empecemos por crear un constructor llamado `Dom`:

```
function Dom(selector) {
  this.selector = selector;
  this.elements = document.querySelectorAll(selector);
};
```

Dentro de esta función utilizamos `document.querySelectorAll` para poder obtener una lista de elementos a través de un selector CSS. En este caso preferimos utilizar `document.querySelectorAll` y no `document.querySelector` para darnos la flexibilidad de trabajar con múltiples nodos.

Para poder crear un elemento también necesitamos un método propio, que también nos permita definir sus atributos:

```
Dom.createElement = function(options) {
  var element = document.createElement(options.tag),
      attributes = Object.keys(options.attributes || {}),
      i = 0;

  for(i; i < attributes.length; i++) {
    element.setAttribute(attributes[i],
options.attributes[attributes[i]]);
  }

  return element;
};
```

Luego de haber creado un elemento, debemos agregarlo a un elemento padre:

```
Dom.prototype.append = function(newChildElement) {
  this.elements[0].appendChild(newChildElement);

  return this;
};
```

En `append` no iteramos por todos los elementos de la instancia, dado que un nodo (`newChildElement`) solo puede ser agregado a un elemento, y tener un solo nodo padre.

En el caso anterior, `newChildElement` debería ser un elemento; pero también debería poder aceptar un objeto similar al pasado en `Dom.createElement`:

```
Dom.prototype.append = function(newChildElement) {
  if (!(newChildElement instanceof Element)) {
    if (newChildElement.hasOwnProperty('tag')) {
      newChildElement = Dom.createElement(newChildElement);
    }
  }

  this.elements[0].appendChild(newChildElement);

  return this;
};
```

De esta forma, verificamos si el parámetro pasado a `Dom.prototype.append` es un objeto plano o una instancia de `Element`.

```
var nav = new Dom('header nav');

nav.append({
```

```
    tag: 'a',
    content: 'Reservaciones',
    attributes: {
      href: '#reservaciones'
    }
  });
```

Recorriendo nodos y elementos

Tanto las interfaces `Node` como `Element` tienen propiedades que permiten obtener los nodos (y elementos) hijos de otro nodo, así como obtener los nodos *hermanos* de un nodo en específico (un nodo *hermano* es aquel nodo que está al mismo nivel que otro y comparten el mismo nodo padre).

Cada interfaz tiene sus propias propiedades para obtener nodos hijos y nodos hermanos; así, `childNodes` devuelve todos los nodos hijos, incluyendo nodos textos, comentarios o elementos; mientras que `children` devuelve todos los nodos hijos que son elementos. La diferencia es más notoria con las propiedades `firstChild` y `firstElementChild`, o `nextSibling` y `nextElementSibling`.

Tanto `childNodes` como `children` devuelven una lista *viva* (también llamada colección *viva*).

Lista *viva*

Algunas propiedades y métodos del DOM devuelven listas "vivas". Una lista *viva* es una lista de elementos que automáticamente actualiza su contenido cuando estos cambian en otra parte del programa. Es decir, tanto si se agrega un elemento que concuerde con la lista o si se elimina un elemento que se encuentre en la lista, esta se actualizará con los elementos nuevos o quitando los eliminados posteriormente.

Vamos a extender `dom.js` para que permita obtener la lista de elementos hijos, que es con la que usualmente se trabaja.

```
Dom.prototype.children = function() {
  return this.elements[0].children;
};
```

Este método no es tan útil, ya que nos devuelve una lista nativa que no tendrá los métodos de `Dom`, por lo que deberíamos *envolver* esta lista en una instancia de `Dom`. Para esto vamos a actualizar el constructor:

```
function Dom(selectorOrElements) {
  if (typeof selectorOrElements === 'string') {
    this.selector = selectorOrElements;
    this.elements = document.querySelectorAll(selectorOrElements);
  }
  else {
    if (selectorOrElements instanceof Node) {   // aprovechamos
para verificar si se envolverá un solo elemento o una lista de
elementos
      this.elements = [selectorOrElements];
    }
    else {
      this.elements = selectorOrElements;
    }
  }
};
```

De esta forma, `Dom.prototype.children` quedará de la siguiente forma:

```
Dom.prototype.children = function() {
  if (this.elements[0] !== undefined) {
    return new Dom(this.elements[0].children);
  }
  else {
    return Dom.empty([]);
```

```
  }
};
```

Adicionalmente, agregamos una validación simple para saber si existen elementos dentro de la instancia de `Dom` que permitan leer nodos hijos, de no existir elementos en `this.elements`, debería devolver una lista vacía. Si bien una instancia de `NodeList` no es un arreglo, ambas tienen una propiedad llamada `length`, por lo que, para efectos prácticos, sirve para representar una lista vacía.

Atributos

En HTML, las etiquetas pueden guardar información sobre sus propiedades mediante atributos. Los atributos más comunes son `id` y `class` y, en el caso de elementos de formulario, los atributos más importantes son `type` y `name`.

La interfaz `Element` tiene métodos para leer, definir, eliminar y verificar si un atributo está definido: `getAttribute`, `setAttribute`, `removeAttribute` y `hasAttribute`, respectivamente. Estos métodos reciben un parámetro en forma de cadena para el nombre (y otro para el valor, en el caso de `setAttribute`).

Dentro de `dom.js` crearemos los métodos `get`, `set`, `unset` y `has`:

```
Dom.prototype.get = function(attributeName) {
  var i = 0,
      attributeValues = [];

  for (i; i < this.elements.length; i++) {

attributeValues.push(this.elements[i].getAttribute(attributeName));
  }

  return attributeValues;
};
```

```
Dom.prototype.set = function(attributeName, attributeValue) {
  var i = 0;

  for (i; i < this.elements.length; i++) {
    this.elements[i].setAttribute(attributeName, attributeValue);
  }
};

Dom.prototype.unset = function(attributeName) {
  var i = 0;

  for (i; i < this.elements.length; i++) {
    this.elements[i].removeAttribute(attributeName);
  }
};

Dom.prototype.has = function(attributeName) {
  var i = 0,
      hasAttributeValues = [];

  for (i; i < this.elements.length; i++) {

hasAttributeValues.push(this.elements[i].hasAttribute(attributeName));
  }

  return hasAttributeValues;
};
```

Como los atributos también son nodos dentro del DOM, no solo se pueden manipular atributos con los métodos `getAttribute`, `setAttribute` y `removeAttribute`. También es posible utilizar nodos de tipo atributo en vez de cadenas como parámetros con los métodos `getAttributeNode`, `setAttributeNode` y `removeAttributeNode`.

Eventos

Los eventos permiten comunicar acciones realizadas tanto por el navegador como por el usuario, y ayudan a mejorar la interacción entre una persona y un sitio o aplicación web. Como ejemplo: cuando un usuario hace clic en un enlace, se puede capturar el evento *click* de ese elemento y lanzar una acción diferente a la habitual (la cual es enviar al usuario al documento enlazado). Otro ejemplo es validar formularios antes de ser enviados, capturando el evento *submit* de el elemento `<form>`.

Todos los elementos del DOM, además de `window`, heredan de la interfaz `EventTarget`, el cual permite enlazar eventos a callbacks definidos dentro de la aplicación. La interfaz `EventTarget` tiene 3 métodos: `addEventListener`, `removeEventListener` y `dispatchEvent`.

addEventListener

Para enlazar un evento a un callback se utiliza `addEventListener`:

```
window.addEventListener('load', function(e) {
  console.log('window:load', e);
});
```

El ejemplo anterior agrega un *listener* al evento `load` de `window`, donde el callback pasado como segundo parámetro es la función que se ejecutará cuando el evento se dispare (que es cuando el navegador termina de cargar el documento).

Todos los callbacks enlazados a eventos toman un solo parámetro (en este caso, `e`). Este parámetro puede ser instancia de `FocusEvent`, `MouseEvent`, `KeyboardEvent`, `UIEvent` o `WheelEvent`, dependiendo del evento que sea lanzado. Todos los eventos heredan de la interfaz `Event`.

removeEventListener

Para eliminar un *listener* de un elemento se utiliza el método `removeEventListener`, que toma los mismos valores de `addEventListener`. Esto quiere decir que, para eliminar un *listener* de un elemento, es obligatorio mandar como parámetro el mismo callback utilizado en `addEventListener`.

En este ejemplo el evento no se eliminará, puesto que los callbacks son diferentes:

```
window.addEventListener('load', function(e) {
  console.log('window:load', e);
});

window.removeEventListener('load', function(e) {
  console.log('window:load', e);
});
```

Por eso, es necesario guardar el callback utilizado en `addEventListener` en una variable para utlizarla luego en `removeEventListener`:

```
var windowOnLoad = function(e) {
  console.log('window:load', e);
};

window.addEventListener('load', windowOnLoad);

window.removeEventListener('load', windowOnLoad);
```

Ahora que ya vimos como agregar y eliminar *listeners* a un elemento, vamos a añadir esta funcionalidad a `dom.js`:

```
Dom.prototype.on = function (eventName, callback) {
  var i = 0,
      eventIdentifier = this.selector + ':' + eventName;

  if (this.events === undefined) {
    this.events = {};
  }
```

```
    if (this.events[eventIdentifier] === undefined) {
      this.events[eventIdentifier] = [];
    }

    this.events[eventIdentifier].push(callback);

    for (i; i < this.elements.length; i++) {
      this.elements[i].addEventListener(eventName, callback, true);
    }
  };

  Dom.prototype.off = function(eventName) {
    var i = 0,
        e = 0,
        eventIdentifier = this.selector + ':' + eventName;

    if (this.events === undefined) {
      this.events = {};
    }

    if (this.events[eventIdentifier] != undefined) {
      for (e; e < this.events[eventIdentifier].length; e++) {
        var callback = this.events[eventIdentifier][e];

        for (i; i < this.elements.length; i++) {
          this.elements[i].removeEventListener(eventName, callback,
  true);
        }
      }

      this.events[eventIdentifier] = [];
    }
  };
```

Como debemos tener constancia de los callbacks que están siendo utilizados en `addEventListener`, los guardamos en la propiedad `this.events`. Luego, si

queremos eliminarlos, iteramos dentro de esa propiedad y eliminamos los *listeners* con `removeEventListener`.

Así, utilizamos nuestros métodos de la siguiente forma:

```
var win = new Dom(window);

win.on('load', function(e) {
  console.log('window:load');
});
```

Para que nuestro `dom.js` funcione con `window` debemos cambiar el constructor una vez más, y debemos verificar si el argumento pasado al constructor es instancia de `EventTarget` (de todas formas, `Node` hereda de `EventTarget`):

```
function Dom(selectorOrElements) {
  if (typeof selectorOrElements === 'string') {
    this.selector = selectorOrElements;
    this.elements = document.querySelectorAll(selectorOrElements);
  }
  else {
    if (selectorOrElements instanceof EventTarget) {
      this.elements = [selectorOrElements];
    }
    else {
      this.elements = selectorOrElements;
    }
  }
};
```

Eventos propios

Adicionalmente a los eventos nativos del navegador (como `load`, `click`, y otros), también se pueden crear eventos propios. Estos eventos propios son

utilizados para propósitos propios de la aplicación. Por ejemplo, y siguiendo con el caso de La Buena Espina, como desarrollador es vital saber cuándo un usuario ha cambiado de sección. Podemos saber esto mediante el uso de eventos propios.

Existen dos formas de crear eventos personalizados:

La primera es utilizando el método `document.createEvent`. Este método funciona en todos los navegadores, incluyendo Internet Explorer 9 y superiores:

```
var sectionChangedEvent = document.createEvent('CustomEvent');
sectionChangedEvent.initCustomEvent('sectionchanged', true, false,
{ previousSection: 'carta', nextSection: 'locales' });

document.addEventListener('sectionchanged', function(e) {
  console.log(e.detail.previousSection + ' → ' +
e.detail.nextSection);
});

document.dispatchEvent(sectionChangedEvent);
// "carta → locales"
```

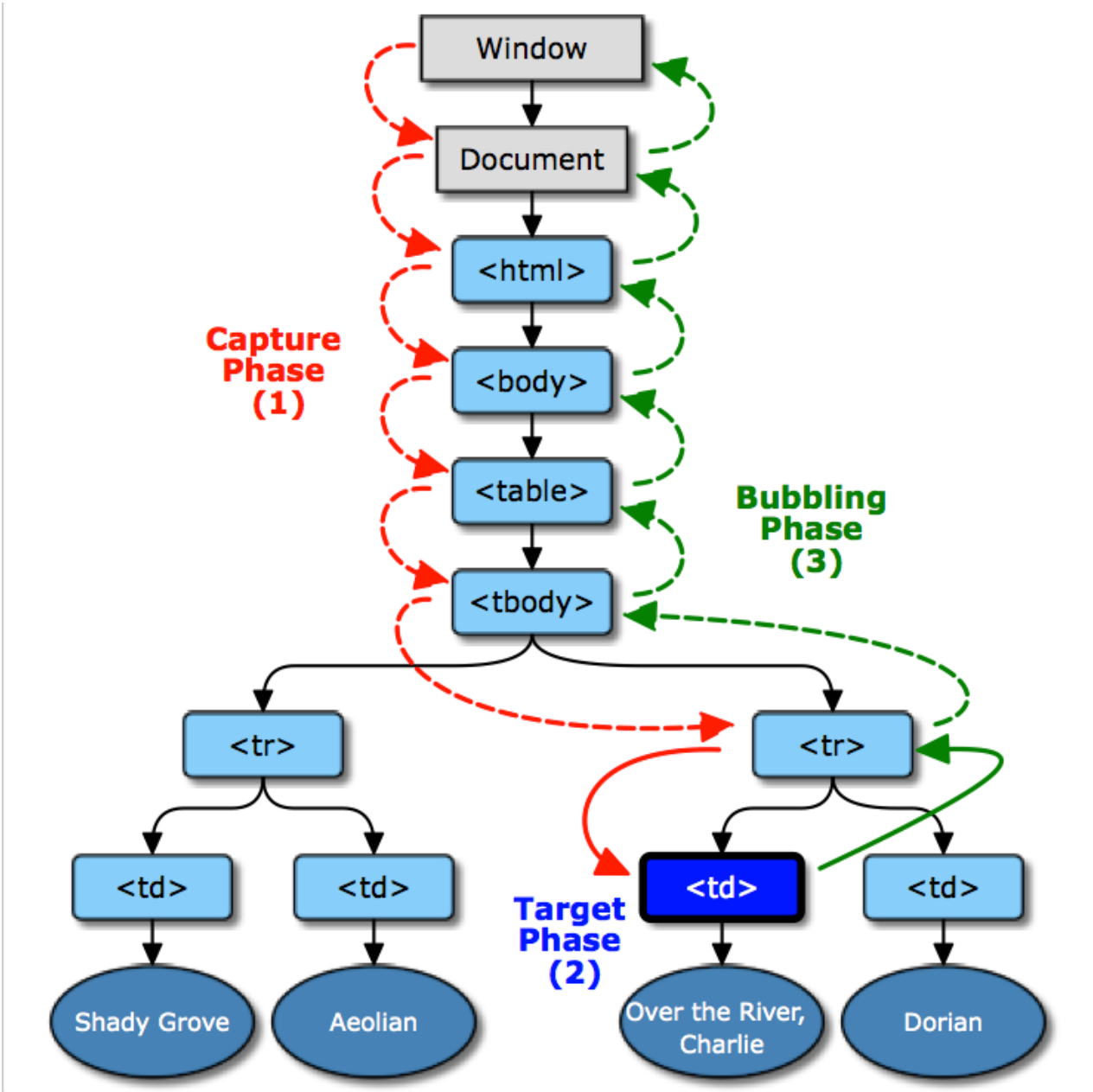
La segunda forma es utilizando el constructor de `CustomEvent`, el cual funciona para todos los navegadores, excepto Internet Explorer:

```
var sectionChangedEvent = new CustomEvent('sectionchanged', {
  bubbles: true,
  cancelable: false,
  detail: {
    previousSection: 'carta',
    nextSection: 'locales'
  }
});

document.addEventListener('sectionchanged', function(e) {
```

```
  console.log(e.detail.previousSection + ' → ' +
e.detail.nextSection);
});

document.dispatchEvent(sectionChangedEvent);
// "carta → locales"
```

Event flow

Cuando un evento es lanzado, este pasa por 3 fases, en el siguiente orden: *Capture phase*, *Target phase* y *Bubbling phase*. El hecho de pasar por las 3 fases es denominado *event flow*.



Document Object Model (DOM) Level 3 Events Specification. *http://www.w3.org/TR/DOM-Level-3-Events/*

En el *event flow*, cada evento lanzado en el DOM empieza en el contexto global (es decir, `window`), pasa por el nodo raíz del documento (`document`) y sigue un camino a través de una serie de nodos hijos (*Capture phase*) que le permita llegar al elemento que lanza dicho evento (*Target phase*). En la *target phase*, el evento es lanzado. Luego, empieza la *bubbling phase*, siguiendo el mismo camino de la *capture phase*, pero en sentido inverso, hasta llegar al contexto global (`window`).

Cuando se registra un *listener*, se puede definir para que sea ejecutado en la *capture phase* o en la *bubbling phase*. El orden en que un *listener* es ejecutado depende de la *fase* en la que está agregado:

```
window.addEventListener('click', function() {
  console.log('Bubbling click event');
}, false);                                    // Este listener
se ejecutará segundo

window.addEventListener('click', function() {
  console.log('Capturing click event');
}, true);                                    // Este listener se
ejecutará primero
```

Para definir la fase en la que se ejecutará un *listener* se pasa un tercer parámetro a `addEventListener`, el cual debe tener un valor booleano: si el parámetro es `true`, el *listener* se ejecutará en la *capture phase*, y si es `false` el *listener* se ejecutará en la *bubbling phase*. Por defecto, el valor de este parámetro es `false`. Cabe señalar que también debe ser pasado a `removeEventListener` si existen dos *listeners*, uno para cada fase, que apunten al mismo evento y elemento.

En `dom.js` definimos el tercer parámetro como `true`, tanto en `Dom.prototype.on` como en `Dom.prototype.off`, haciendo que todos los *listeners* sean ejecutados en la *capture phase*. De esta forma, el orden en el que agregamos los *listeners* será el mismo en el que son lanzados.

Rendimiento

Trabajar con el DOM puede traer consecuencias inesperadas en temas de rendimiento si no se toman en cuenta algunas características propias de los navegadores. Por ejemplo, cuando se manipula el árbol DOM, el navegador recalcula posiciones y re-renderiza la pantalla (ver *Reflow y repaint*).

Reflow y *repaint*

Cuando se renderiza un documento HTML en un navegador ocurren dos acciones: *reflow* (o *layout*) y *repaint*. Al realizar el *reflow*, el navegador calcula las dimensiones y posiciones de cada elemento visible y los coloca en la posición previamente calculada dentro de la zona visible del navegador (o *viewport*). Cuando se realiza el *repaint*, el navegador obtiene la información de las hojas de estilo del documento, así como de los estilos del sistema y del navegador, y muestra los elementos de la forma como fue ideada (bordes, fondos, colores, imágenes, etc). Cuando se realiza un *reflow*, también se realiza un *repaint*, pero no es así de forma inversa (puede cambiarse el fondo de un elemento y el navegador no tendrá que recalcular posiciones de elementos).

Algunas de las acciones que obligan al navegador a realizar *reflow* (y su respectivo *repaint*) está relacionadas al uso de CSS; y otras a la manipulación del árbol DOM con JavaScript, como:

- Agregar o eliminar un elemento al documento.
- Cambiar el contenido de un elemento con `innerText` e `innerHTML`.
- Cambiar la visibilidad de un elemento con la propiedad `display` del CSS (manipulando el atributo `style` de un elemento).
- Cambiar la clase CSS o los estilos de un elemento (atributos `className`, `classList` o `style`).
- Redimensionar la ventana o el *viewport*.
- Utilizar el método `getComputedStyle`.
- Leer las propiedades de `MouseEvent`: `layerX`, `layerY`, `offsetX` y `offsetY`.
- Realizar scroll con los métodos `scrollIntoView`, `scrollIntoViewIfNeeded`, `scrollByLines` o `scrollByPages`.
- Leer algunas propiedades de elementos: `clientLeft/Top/Width/Height`, `scrollLeft/Top/Width/Height`, `offsetLeft/Top/Width/Height`, entre otras.

`document.createDocumentFragment`

Este método permite crear una versión más ligera y limitada de `document`, y sirve para mejorar el rendimiento de operaciones donde se necesiten agregar muchos nodos.

Cuando se agregan nodos dentro de un bucle (la operación más común cuando

se quieren agregar un indeterminado número de elementos), se utiliza el método `appendChild`, el cual hace que el navegador recalcule posiciones y renderice la pantalla tantas veces como iteraciones tuvo el bucle.

La ventaja de utilizar un fragmento (una instancia de `DocumentFragment`) es que, al estar separado del árbol DOM del documento y es guardado en memoria, evita que el navegador tenga que renderizar de nuevo cada vez que se agregue un nodo al fragmento.

Para probar este método implementaremos un método llamado `html`, el cual permitirá agregar elementos a un nodo pero utilizando cadenas. Inicialmente usaremos la propiedad `innerHTML`:

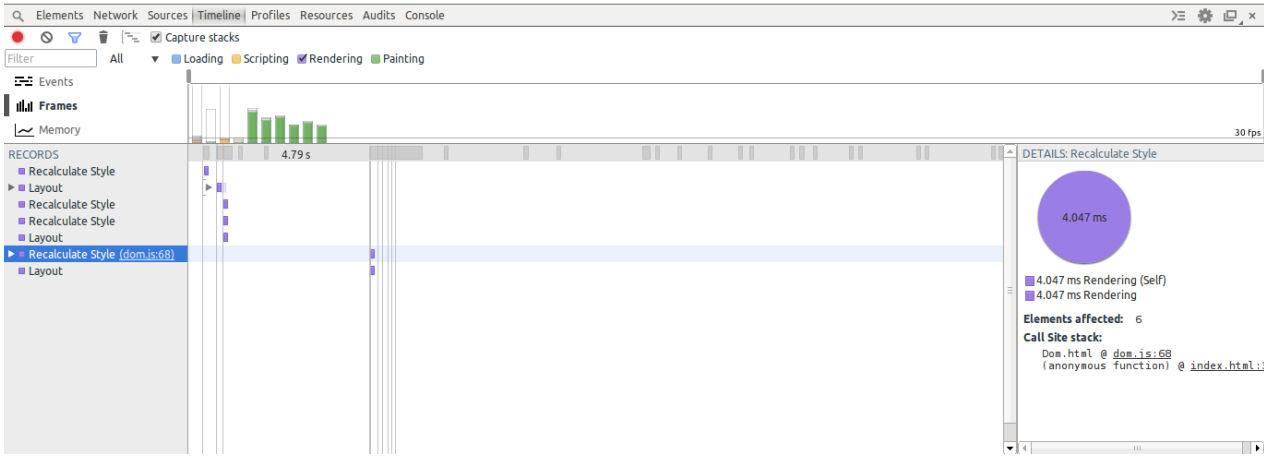
```
Dom.prototype.html = function(htmlString) {
  var i = 0;

  // Eliminamos el contenido de todos los elementos
  for (i; i < this.elements.length; i++) {
    this.elements[i].textContent = '';
  }

  // Agregamos el nuevo contenido a todos los elementos
  for (i = 0; i < this.elements.length; i++) {
    this.elements[i].innerHTML = htmlString;
  }
};
```

Vamos a probar en un nodo vacío ejecutando la siguiente instrucción:

```
new Dom('#background').html(
  '<div class="slide" id="slide-1" title="Créditos:
http://www.flickr.com/photos/saucesupreme/6774616862/"></div>' +
  '<div class="slide" id="slide-2" title="Créditos:
http://www.flickr.com/photos/c32/4775267221/"></div>' +
  '<div class="slide" id="slide-3" title="Créditos:
```

```
    http://www.flickr.com/photos/renzovallejo/7998183161/"></div>' +
      '<div class="slide" id="slide-4" title="Créditos:
    http://www.flickr.com/photos/renzovallejo/7998217897/"></div>' +
      '<div class="slide" id="slide-5" title="Créditos:
    http://www.flickr.com/photos/renzovallejo/7998185723/"></div>' +
      '<div class="slide" id="slide-6" title="Créditos:
    http://www.flickr.com/photos/renzovallejo/7998141711/"></div>'
    );
```

Este método nos da el siguiente gráfico dentro de la pestaña *Timeline* de Chrome:

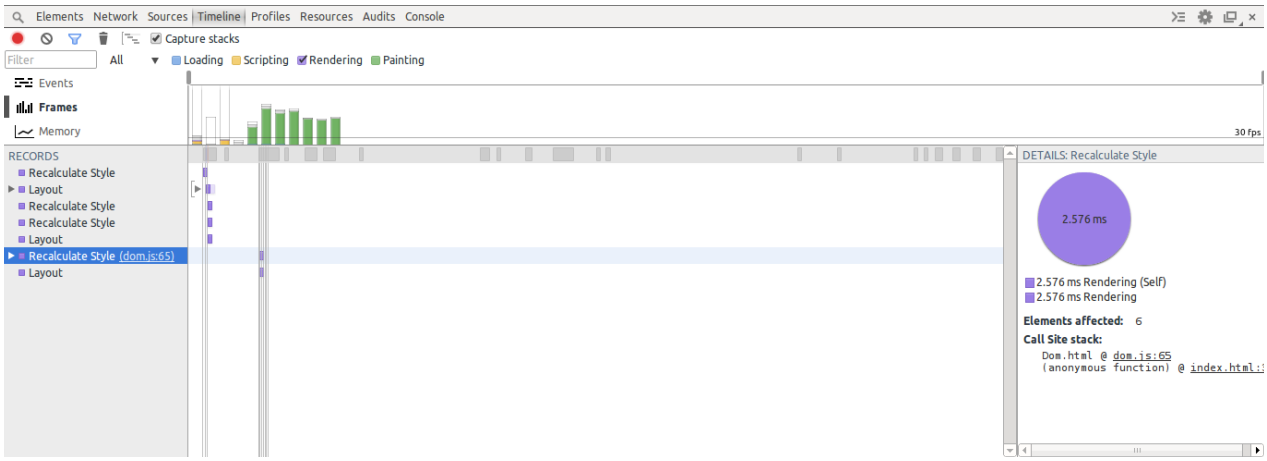


Utilizando innerHTML

Ahora vamos a utilizar un fragmento:

```
  Dom.prototype.html = function(htmlString) {
    var i = 0,
        f = 0;

    var fragment = document.createDocumentFragment(),
        root = Dom.createElement({
          tag: 'div',
          attributes: {
            id: 'root'
          }
```

```
      });

    root.innerHTML = htmlString;

    for (f; f < root.childNodes.length; f++) {
      fragment.appendChild(root.childNodes[f].cloneNode(true));
    }

    // Eliminamos el contenido de todos los elementos
    for (i; i < this.elements.length; i++) {
      this.elements[i].textContent = '';
    }

    root = null;

    // Agregamos el fragmento a todos los elementos
    for (i = 0; i < this.elements.length; i++) {
      this.elements[i].appendChild(fragment.cloneNode(true));
    }
  };
```



Utilizando fragmento

El tiempo utilizado por el navegador para recalcular estilos luego de añadir los elementos al DOM, segun la técnica:

Técnica	Tiempo (ms)

Técnica	Tiempo (ms)
`Element.prototype.innerHTML`	4.047ms
`document.createDocumentFragment`	2.576ms

Enlazando eventos a múltiples elementos

Uno de los casos más comunes de uso de eventos es el de enlazar eventos a diferentes elementos que son similares (por ejemplo, un cliente de correo tiene el mismo enlace "marcar como importante" para cada correo en la bandeja). Con el DOM, se agrega un *listener* de un evento a un elemento utilizando `addEventListener`, pero no se puede agregar un *listener* a una lista de elementos.

Agregar un *listener* por cada elemento puede ser una solución pero, a medida que existan más elementos del mismo tipo, se necesitarán agregar más *listeners*, aumentando la cantidad de memoria utilizada por el navegador.

Event delegation

Event delegation es una técnica que permite disminuir la cantidad de *listeners* creados, y que tiene como ventaja adicional el permitir que un elemento recién creado pueda *escuchar* un evento, sin necesidad de tener su propio *listener*.

Esta técnica utiliza el *event flow* para agregar un *listener* al elemento padre de todos los elementos que compartirán la misma funcionalidad. Debido a la naturaleza del *event flow*, el elemento padre lanzará el evento si tiene un *listener* registrado.

Existen dos formas de obtener el elemento que lanza el evento: el contexto del mismo (`this`) o la propiedad `target` del evento (el parámetro del callback): Cuando se usa `addEventListener`, `this` y `target` referencian al mismo elemento, mientras que en *event delegation*, `this` será el elemento que ejecute `addEventListener` (es decir, el elemento padre), mientras que `target` será el elemento que lance el evento (es aquí donde ocurre la *target phase*).

Dentro del callback del *listener*, se verifica que el elemento referenciado en `target` sea el que se desea utilizar (usualmente comparando las clases o el id de `target`).

Vamos a implementar el *event delegation* en el método `delegate`:

```
Dom.prototype.delegate = function(eventName, selector, callbak) {
  var i = 0,
      eventIdentifier = selector + ':' + eventName;

  if (this.events === undefined) {
    this.events = {};
  }

  if (this.events[eventIdentifier] === undefined) {
    this.events[eventIdentifier] = [];
  }

  this.events[eventIdentifier].push(callback);

  for (i; i < this.elements.length; i++) {
    this.elements[i].addEventListener(eventName, function(e) {
      if (e.target.webkitMatchesSelector(selector)) {
        callback(e);
      }
    }, true);
  }
};
```

En esta primera implementación utilizamos el método `webkitMatchesSelector`, el cual verifica que un elemento concuerde con un selector CSS dado. Si lo dejamos de esta forma, `Dom.prototype.delegate` solo funcionará en navegadores basados en Webkit y Blink, así que crearemos un método auxiliar:

```
Dom.match = function(element, selector) {
  var matchesSelector = element.matchesSelector ||
element.webkitMatchesSelector || element.mozMatchesSelector ||
```

```
  element.oMatchesSelector || element.msMatchesSelector;

    return matchesSelector.call(element, selector);
  };
```

Así, cambiamos `e.target.webkitMatchesSelector(selector)` por `Dom.match(e.target, selector)`:

```
  Dom.prototype.delegate = function(eventName, selector, callbak) {
    var i = 0,
        eventIdentifier = selector + ':' + eventName;

    if (this.events === undefined) {
      this.events = {};
    }

    if (this.events[eventIdentifier] === undefined) {
      this.events[eventIdentifier] = [];
    }

    this.events[eventIdentifier].push(callback);

    for (i; i < this.elements.length; i++) {
      this.elements[i].addEventListener(eventName, function(e) {
        if (Dom.match(e.target, selector)) {
          callback(e);
        }
      }, true);
    }
  };
```

Y lo utilizamos de la siguiente manera (podemos probar en el archivo http://cevichejs.com/files/3-dom-cssom/index.html):

```
var doc = new Dom(document);

doc.delegate('click', 'nav a', function(e) {
  console.log(e.target.getAttribute('href'));
});

// Si empezamos a hacer clic a los enlaces de la barra superior nos
saldrán los siguientes mensajes:
// "#carta"
// "#locales"
// "#historia"
```

Si agregamos un enlace más a la barra superior y hacemos clic en él, el evento `click` también se lanzará:

```
var nav = new Dom('nav');
nav.append({
  tag: 'a',
  attributes: {
    href: '#reservaciones'
  },
  content: 'Reservaciones'
});

// Hacemos clic en el enlace recientemente creado y saldrá el
siguiente mensaje:
// "#reservaciones"
```

Esta es una de las ventajas de utilizar *event delegation*: con un solo *listener* hemos capturado eventos de 4 enlaces.

Mejorando `dom.js`

Existen algunos métodos útiles que podemos agregar a `dom.js` y que nos servirán en La Buena Espina.

Por ejemplo, necesitaremos agregar y eliminar clases a elementos dentro del DOM:

```
Dom.prototype.addClass = function(className) {
  var i = 0;

  for (i; i < this.elements.length; i++) {
    this.elements[i].classList.add(className);
  }
};

Dom.prototype.removeClass = function(className) {
  var i = 0;

  for (i; i < this.elements.length; i++) {
    this.elements[i].classList.remove(className);
  }
};

Dom.prototype.hasClass = function(className) {
  var i = 0,
      hasClass = [];

  for (i; i < this.elements.length; i++) {
    hasClass.push(this.elements[i].classList.contains(className));
  }
};
```

Así como saber cuál es el primer y último elemento hijo de un contenedor:

```
Dom.prototype.firstChild = function() {
```

```
  return new Dom(this.elements[0].firstElementChild);
};

Dom.prototype.lastChild = function() {
  return new Dom(this.elements[0].lastElementChild);
};
```

O el lugar que ocupa un elemento entre sus elementos hermanos:

```
Dom.prototype.index = function() {
  return
Array.prototype.indexOf.call(this.elements[0].parentNode.children,
this.elements[0]);
};
```

En este método utilizamos el método `indexOf` de `Array`. Las propiedades en el DOM como `children` no devuelven arreglos, si no listas instancias de `NodeList` (o `HTMLCollection` dependiendo del navegador); y si bien estas listas no son arreglos, es posible utilizar los métodos propios de instancias de `Array` juntando dos conceptos vistos en el capítulo anterior: *prototypes* y el método `call`.

A grandes rasgos, `Array.prototype.indexOf` es una función que itera a partir de los elementos del *arreglo* que lo ejecuta (es decir, su *contexto*) mientras busca por el elemento que es pasado como parámetro, cuando lo encuentra devuelve el número de la iteración en la que ha sido encontrado, el cual es el índice en el que se encuentra dicho elemento. En nuestro caso, `children` no es un arreglo, pero todas sus propiedades pueden ser accesibles mediante índices que empiezan en 0, y tiene una propiedad `length` que contiene el número de elementos dentro de la lista. Este tipo de objetos son llamados *array-like objects* y se pueden ver en muchas partes del DOM y del lenguaje en sí (por ejemplo, la palabra reservada `arguments` o las instancias de `CSSStyleDeclaration` y `NamedNodeMap` son *array-like objects*).

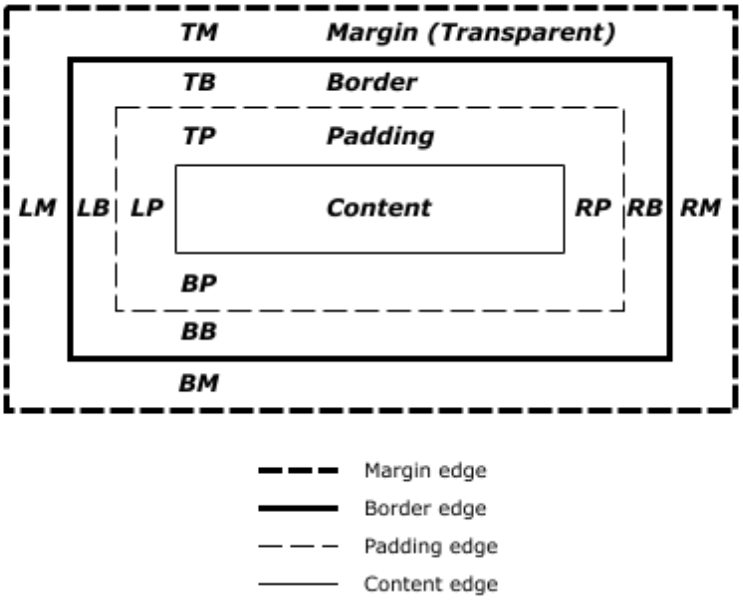
CSSOM

El Cascade Style Sheet Object Model, o CSSOM, toma los conceptos de DOM y los lleva a las hojas de estilo en cascada que componen un documento HTML. Esto permite tener un control más profundo de las reglas y propiedades que se aplican tanto a un elemento como a un documento HTML, utilizando JavaScript.

Box model

En el navegador, cada elemento mostrado en pantalla tiene forma de *caja* (o rectángulo), por lo que tiene un alto y un ancho, el cual puede ser definido por el navegador mismo o por el usuario mediante CSS.

La especificación de CSS define el denominado *box model*, el cual empieza definido como un rectángulo que limite el contenido del elemento (*content area*). Este rectángulo *content area* está rodeado por otro rectángulo más grande, denominado *padding*; el *padding*, a su vez, está rodeado por un rectángulo externo denominado *border*. Por último, el rectángulo *border* está rodeado por el rectángulo *margin*:



Box model. *http://www.w3.org/TR/CSS2/box.html*

Los rectángulos *padding*, *border* y *margin* pueden ser personalizados mediante CSS. Incluso, los 4 lados de cada rectángulo pueden tener valores diferentes.

Reglas y propiedades

Cada elemento de una hoja de estilos es una *regla*, y cada regla está definida por dos partes: el *selector* (es decir, el elemento o elementos a los cuales aplicará la regla), y una lista de *propiedades*.

```
body {
  font-family: 'Arial';
  background: green;
  color: white;
}
```

En este caso, existe una regla cuyo selector es `body`, y sus propiedades son las definidas por `font-family`, `background` y `color`.

Agregando reglas

Cada hoja de estilos en un documento es accesible mediante la propiedad `styleSheets` de `document`. Esta devuelve una lista viva, instancia de `StyleSheetList` (similar a `NodeList`), conteniendo objetos instancias de `CSSStyleSheet`. Por cada hoja de estilos, enlazada mediante la etiqueta `<link>` o definida dentro del documento con `<style>`, existe una instancia de `CSSStyleSheet` dentro de `document.styleSheets`.

Cada instancia de `CSSStyleSheet` tiene una propiedad `cssRules`, una lista *viva* instancia de `CSSRuleList` (similar a `StyleSheetList` y `NodeList`). Esta lista contiene objetos de diferentes interfaces, dependiendo del tipo de regla que referencia en la hoja de estilos:

Tipo de regla	Interfaz
Estilos	`CSSStyleRule`

Tipo de regla	Interfaz
Definir un charset (`@charset`)	`CSSCharsetRule`
Importar una hoja de estilos (`@import`)	`CSSImportRule`
Media query (`@media`)	`CSSMediaRule`
Font face (`@font-face`)	`CSSFontFaceRule`
Definir estilos para impresión (`@page`)	`CSSPageRule`
Una lista de key frames (`@keyframes`)	`CSSKeyframesRule`
Un elemento de una lista de key frames	`CSSKeyframeRule`

Solo en Webkit y Blink, los objetos `CSSStyleSheet` que referencian a hojas de estilos enlazadas desde un dominio diferente al del documento (como un CDN) no permiten leer sus reglas, mientras que en el resto de casos y navegadores sí es posible.

Para poder agregar una regla a una hoja de estilos se utiliza el método `addRule` o `insertRule`, disponible en cada instancia de `CSSStyleSheet`. Mientras `addRule` acepta 3 parámetros (selector, propiedades, índice donde agregar la regla), `insertRule` acepta solo 2 (cadena con el contenido completo de la regla, índice donde agregar la regla).

`window.getComputedStyle`

Este método devuelve un objeto instancia de `CSSStyleDeclaration` con todos los estilos de un elemento pasado como parámetro. Estos estilos son calculados por el navegador a partir de estilos propios del sistema operativo, el navegador y hojas de estilos incluidas en el documento que contengan reglas aplicables a dicho elemento. Estos estilos son de solo lectura, a diferencia de los obtenidos por la propiedad `style` de cada instancia de `CSSStyleRule`.

En `dom.js` crearemos un método `style`:

```
Dom.prototype.style = function() {
  var i = 0;
```

```
    var styles = [];

    for (i; i < this.elements.length; i++) {
      styles.push(window.getComputedStyle(this.elements[i]));
    }

    return styles;
  };
```

Sin embargo, `window.getComputedStyle` es un método pesado que impacta en el rendimiento de la aplicación, por lo que hay que tener cuidado en qué momentos se va a utilizar. Además, `window.getComputedStyle` devuelve una lista *viva*, así que cada vez que cambiemos algún estilo dentro de un elemento, se verá reflejado en cualquier variable o propiedad que guarde un valor previo de `window.getComputedStyle`:

```
var bodyStyle = window.getComputedStyle(document.body);

bodyStyle.backgroundColor;
// "rgb(255, 255, 255)"

document.body.style.background = 'rgba(10, 10, 10, 0.2)';

bodyStyle.backgroundColor;
// "rgba(10, 10, 10, 0.2)"
```

Media queries

Los *media queries* permiten utilizar CSS para actualizar la presentación de un documento en cuanto cumpla algunas condiciones, como el ancho y alto del *viewport*, orientación del dispositivo, entre otras. Si bien muchas de las condiciones se pueden verificar mediante JavaScript y el DOM, este tipo de operaciones son lentas y suelen perjudicar notablemente el rendimiento de una aplicación web (este tipo de perjuicio se nota aún más en dispositivos móviles),

por lo que, de ser posible, es recomendable utilizar *media queries*.

`window.matchMedia`

Con este método se puede saber si una o más condiciones corresponden al estado actual del navegador, y devuelve una instancia de `MediaQueryList` que permite agregar *listeners* con el método `addListener`. A diferencia del DOM, este método solo acepta un parámetro, el cual es un callback que se ejecutará cada vez que cambie alguna de las condiciones que se están evaluando. Este manejo de eventos es útil para precargar estilos, *scripts*, o imágenes específicas según resoluciones diferentes.

Para manejar mejor este método crearemos un archivo `cssom.js`, el cual contendrá un objeto `CSSom`:

```
CSSom = {
  mediaQueries: {}
};
```

Como las instancias de `MediaQueryList` permiten agregar *listeners*, crearemos dos métodos para manejarlos de forma más simple:

```
CSSom.on = function(mediaQueryString, callback) {
  var mediaQueryList;

  if (this.mediaQueries[mediaQueryString] === undefined) {
    this.mediaQueries[mediaQueryString] = [];
  }

  mediaQueryList = window.matchMedia(mediaQueryString);
  mediaQueryList.addListener(callback);

  this.mediaQueries[mediaQueryString].push({
    mediaQueryList: mediaQueryList,
    callback: callback
```

```
  });
};

CSSom.off = function(mediaQueryString) {
  var i = 0,
      mediaQueryResult;

  if (this.mediaQueries[mediaQueryString] !== undefined) {
    for (i; i < this.mediaQueries[mediaQueryString]; i++) {
      mediaQueryResult = this.mediaQueries[mediaQueryString];

mediaQueryResult.mediaQueryList.removeListener(mediaQueryResult.callba
    }

    this.mediaQueries[mediaQueryString] = [];
  }
};
```

De esta forma, podemos definir eventos para determinados media queries:

```
CSSom.on('(orientation: portrait)', function(mq) {
  if (mq.matches) {
    document.body.className = 'portrait';
  }
  else {
    document.body.className = 'landscape';
  }
});
```

Transiciones y Animaciones

Las transiciones y animaciones son nuevos estilos en CSS que permiten animar, valga la redundancia, elementos dentro de un documento, interpolando los valores de algunas de sus propiedades, como el alto, ancho y posición (aunque

muchas otras propiedades pueden ser animadas).

Por ejemplo, **La Buena Espina** se vería bien si le ponemos un efecto simple para cambiar la imagen de fondo cada cierto tiempo. Inicialmente necesitaremos tener una serie de elementos `div` con clase `slide`, y definimos que todos esos elementos por defecto no deben ser visibles, dándole un valor de 0 a la propiedad `opacity`, mientras que el *slide* que quiera mostrarse debe tener el valor de 1 en la misma propiedad:

```
.slide {
  opacity: 0;
}

.slide.current {
  opacity: 1;
}
```

El siguiente paso es definir una transición para la propiedad `opacity`, la cual durará 5 segundos. El navegador es el encargado de calcular los valores que tendrá `opacity` durante los 5 segundos que dure la transición:

```
.slide {
  opacity: 0;
  transition-property: opacity;
  transition-duration: 5.0s;
}

.slide.current {
  opacity: 1;
}
```

Las animaciones en CSS son similares a las transiciones, con la principal diferencia que las animaciones dan más control al usuario con respecto a los valores que puede tener una propiedad dentro del ciclo de vida de una animación (mientras que, en una transición, es el navegador el que calcula dichos valores).

Para definir una animación en CSS, necesitamos definir una regla del tipo *keyframes*, la cual contendrá una lista de puntos en donde una propiedad cambiará de valor. De esta forma, el navegador se encargará de tomar dos *keyframes* (uno de partida y uno de fin) y calculará los valores intermedios para el tiempo que debe transcurrir entre estos dos *keyframes*. Si bien en esto las animaciones son similares a las transiciones, en las animaciones pueden haber más de dos *keyframes*, lo que le da más control al usuario al momento de animar un elemento.

Basados en el ejemplo anterior, podemos utilizar animaciones en vez de transiciones:

```
.slide {
  opacity: 0;
}

.slide.current {
  animation-name: slide;
  animation-duration: 5s;
  animation-fill-mode: forwards;
}

@keyframes slide {
  0% {
    opacity: 0;
  }

  50% {
    opacity: 0.3;
  }

  80% {
    opacity: 0.7;
  }
```

```
    100% {
      opacity: 1;
    }
  }
```

En este caso, con las animaciones tenemos el poder de definir qué valores tendrá `opacity` al inicio (0%), fin (100%), a los 2 segundos y medio (50%), y a los 4 segundos (80%) de transcurrida la animación.

El hecho de tener más control en las animaciones también se ve reflejado en el CSSOM. Mientras que solo existe el evento `transitionend` para las transiciones, las animaciones tienen 3 eventos: `animationstart`, `animationiteration`, `animationend`.

Por ejemplo, si queremos agregar un *listener* al evento `transitionend` que muestre en la consola el tiempo transcurido en la transición:

```
dom('.slide').on('transitionend', function(e) {
  console.log('transitionend', e.elapsedTime);
});
```

Mientras que, si utilizamos animaciones, podemos agregar un *listener* al evento `animationstart` para saber cuándo empezó una animación:

```
dom('.slide').on('animationstart', function(e) {
  console.log('animationstart', e);
});
```

Y agregar un *listener* al evento `animationend` para saber en qué momento terminó una animación:

```
dom('.slide').on('animationend', function(e) {
  console.log('animationend', e);
```

```
});
```

El evento `animationiteration` es lanzado cada vez que empieza una iteración de la animación. Una animación puede ser definida para ser ejecutada un número determinado de veces (y cada vez es una **iteración**), mediante la propiedad `animation-iteration-count`.

Sabiendo un poco más sobre transiciones y animaciones crearemos un *script* simple para crear el efecto para cambiar la imagen de fondo. Primero, debemos tener un poco de HTML base:

```
<div id="background">
  <div class="slide current" id="slide-1" title="Créditos:
http://www.flickr.com/photos/saucesupreme/6774616862/"></div>
  <div class="slide" id="slide-2" title="Créditos:
http://www.flickr.com/photos/c32/4775267221/"></div>
  <div class="slide" id="slide-3" title="Créditos:
http://www.flickr.com/photos/renzovallejo/7998183161/"></div>
</div>
```

Notemos que el primer *slide* tiene la clase `current`, de esta forma nos aseguramos de mostrar una imagen al cargar el sitio.

Ahora, agregamos el CSS respectivo:

```
#background {
  display: block;
  width: 100%;
  height: 100%;
  position: relative;
}

#background .slide {
```

```
  display: block;
  width: 100%;
  height: 100%;
  position: absolute;
  opacity: 0;
  transition-property: opacity;
  transition-duration: 3.5s;
}

#background .slide.current {
  opacity: 1;
}

#slide-1 {
  background: url('../images/slides/slide-1.jpg') no-repeat center
center;
  background-size: 100% auto;
}

#slide-2 {
  background: url('../images/slides/slide-2.jpg') no-repeat center
center;
  background-size: 100% auto;
}

#slide-3 {
  background: url('../images/slides/slide-3.jpg') no-repeat center
center;
  background-size: 100% auto;
}
```

Hemos definido una transición para la propiedad `opacity` que dure 3 segundos y medio. De esta forma, si le agregamos la clase `current` a un elemento con la clase `slide`, se *disparará* la transición que cambie el valor de `opacity` de 0 a 1.

Hasta este momento solo se mostrará la primera imagen de fondo y no

cambiará. Para empezar con la secuencia de imágenes, usaremos el evento `load` de `window` para agregarle la clase `current` al siguiente elemento:

```
dom(window).on('load', function() {
  var current = dom('.slide.current'),
      next = current.next();

  current.removeClass('current');
  next.addClass('current');
});
```

Este código tiene dos particularidades que no hemos visto en este capítulo:

- La función `dom`: En realidad solo devuelve una instancia de `Dom`, pero es útil ya que evita tener que crear una instancia de `Dom` cada vez que queramos trabajar con el DOM.
- El método `next`: Toma el elemento actual (en este caso, el *slide* que tenga la clase `current`) y devuelve su siguiente elemento hermano, con `nextElementSibling`.

Con este código, ya podremos ver que cambia de la primera imagen a la segunda mediante una transición de opacidad (propiedad `opacity`), pero al terminar esta transición no cambia a la tercera. Para lograrlo usaremos el evento `transitionend`:

```
dom('#background').delegate('transitionend', '.slide.current',
function(e) {
  var current = dom(e.target),
      next = current.next();

  current.removeClass('current');
  next.addClass('current');
});
```

Estamos utilizando el *event delegation* para definir un solo evento

`transitionend`, en vez de tener uno por cada elemento con clase `slide`. Así mismo, definimos el evento solo para el *slide* visible en el momento, ya que el evento `transitionend` se disparará tanto para cuando termina la transición del valor de `opacity` de 0 a 1 (invisible a visible) como de 1 a 0 (visible a invisible).

Sin embargo, cuando ya se haya mostrado el último elemento (aquel que tiene id `slide-3`) y se lance el evento `transitionend`, el callback tratará de encontrar el siguiente elemento con `current.next()`. Dado que `current` guarda una referencia al último *slide* de la lista, `next()` no encontrará ningún valor, por lo que, internamente, la propiedad `elements` tendrá valor `null` y dará error al intentar ejecutar el método `addClass`.

Para corregir este pequeño *bug*, verificamos si se llegó al último elemento mediante el método `isLastSibling`. Si es el último elemento de la lista (en otras palabras, el último de sus elementos hermanos), `next` guarda una referencia al primer elemento de la lista con el método `firstSibling`.

```
dom('#background').delegate('transitionend', '.slide.current',
function(e) {
  var current = dom(e.target),
      next = current.next();

  current.removeClass('current');

  if (current.isLastSibling()) {
    next = current.firstSibling();
  }

  next.addClass('current');
});
```

← 2. Capítulo 2: Funciones 4. Capítulo 4: APIs del navegador →

Acerca de | Disponible en Amazon.com