Orientación a objetos en JavaScript

Marcos González Sancho
Vicent Moncho Mas
Jordi Ustrell Garrigós

PID_00220484

UOC
Universitat Oberta de Catalunya

Índice

1. POO en JavaScript

1.1. Introducción

La programación orientada a objetos (POO) es actualmente el paradigma de programación con mayor adopción entre los desarrolladores, así como por los nuevos lenguajes que van surgiendo.

La base de este paradigma es el uso de la abstracción para crear un modelo basado en el mundo real, y se trata de un modelo organizado en torno a los "objetos" en lugar de a las "acciones", y a los "datos" en lugar de a la "lógica".

Históricamente un programa era visto como un conjunto de procesos lógicos que, a partir de unos datos de entrada, se procesaban y producían unos datos de salida. La programación orientada a objetos utiliza "objetos" (estructuras de datos y métodos) y sus interacciones para diseñar aplicaciones. En este paradigma, cada objeto debe ser visto como una entidad con una serie de propiedades que le hacen único y que es responsable de realizar una serie de procesos (métodos) para los que ha sido creado basándose un patrón o tipo común para todos los objetos de su misma clase.

JavaScript implementa los cuatro principios básicos de la programación orientada a objetos (abstracción, encapsulación, herencia y polimorfismo), pero a diferencia de otros lenguajes como Java, es posible programar sin necesidad de utilizar exclusivamente estas características.

> JavaScript implementa los cuatro principios básicos de la programación orientada a objetos: **abstracción**, **encapsulación**, **herencia** y **polimorfismo**.

Si bien históricamente el uso de JavaScript se ha aplicado habitualmente a desarrollos destinados a complementar determinada funcionalidad e interacción en las páginas HTML sobre las que se ejecutaba, con el cambio que ha experimentado el desarrollo web, hoy en día su uso se ha profesionalizado notablemente, y se ha pasado de emplearlo sin prácticamente aplicar principios de POO (mediante el uso de funciones y código aislado) a no solamente trabajar con POO, sino a mejorar los editores, IDE, herramientas complementarias, *frameworks*, librerías e incluso avanzar muy notablemente en la aplicación de patrones de diseño y buenas prácticas a la hora de abordar aplicaciones completas y complejas.

La propia naturaleza del lenguaje basada en prototipos, el hecho de ser débilmente tipado y su dinamismo, junto con su evolución que se ha producido de la mano del aumento del protagonismo que el desarrollo web ha ido adquiriendo con la implantación de internet en todos los ámbitos de nuestra vida cotidiana, nos ha llevado a tener un panorama relativamente confuso a la hora de abordar el tema de la POO en JavaScript.

Es muy habitual encontrar diferentes modos de abordarla, desde una aproximación clásica hasta enfoques más modernos que han ido siendo posibles a raíz de nuevos métodos y funciones soportados por los navegadores y el lenguaje. No obstante, esto no ha de verse como un problema, sino como parte de la riqueza de este lenguaje que hoy en día es prácticamente omnipresente.

1.2. Tipos de objetos en JavaScript

Como en todos los lenguajes de programación orientados a objetos, se puede hacer una clasificación de los tipos soportados estableciendo como criterio su origen.

Objetos nativos de JavaScript

Dentro de este conjunto de objetos, se incluyen:

- Objetos asociados a los tipos de datos primitivos, como String, Number y Boolean, que proporcionan propiedades y métodos para estos tipos de datos.

- Objetos asociados a tipos de datos complejos-compuestos, como Array y Object.

- Objetos complejos con un claro enfoque de utilidad, como por ejemplo Date para el trabajo con fechas, Math para el trabajo con operaciones matemáticas y RegExp para el trabajo con expresiones regulares.

Las características de estos objetos están especificadas por el estándar ECMA-262 aunque, como suele ocurrir con los estándares, las distintas implementaciones por los creadores de navegadores pueden añadir algunas particularidades.

Objetos del navegador. BOM (*browser object model*)

Dado que JavaScript tiene su origen en un entorno basado en un navegador, existen algunos objetos predefinidos que el navegador "expone" para ser accesibles desde JavaScript, de manera que tenga acceso a algunos de sus elementos como, por ejemplo, a través de los objetos `Window` y `Navigator`, que permiten el trabajo con la ventana y la interacción con el navegador respectivamente.

El conjunto de objetos de este tipo se suele agrupar bajo el nombre BOM (*browser object model*). No existe un estándar para ello lo que, y el propio hecho de estar íntimamente ligados al navegador, hace que no sean estándar y dispongan de variaciones dependiendo del navegador e incluso de la versión que se esté utilizando.

Objetos del documento

Forman una parte de la especificación estándar del DOM (*document object model*) y definen la estructura de la interfaz de la página HTML. El consorcio W3C ha realizado esta especificación estándar del DOM para ayudar a converger las distintas estructuras implementadas.

De manera similar a los objetos del navegador, los objetos del DOM "exponen" los elementos y determinadas funcionalidades de ellos para que este DOM pueda ser manipulado desde JavaScript, siendo el principal elemento el objeto `Document`.

Objetos personalizados o definidos por el usuario

Estos son definidos y creados por el propio usuario con el objetivo de adaptar su software a la realidad de los tipos de datos y procesos que se quiere representar mediante POO. El segundo apartado de este módulo se centrará en el estudio de la creación de objetos personalizados.

Como se puede deducir de los anteriores puntos, no existe un estándar que imponga todos los aspectos de JavaScript, si bien JavaScript 1.5 es compatible con ECMA-262 (de ECMA International[1]), en su tercera edición regula los aspectos básicos del lenguaje y la especificación DOM de W3C[2] define cómo se deben presentar los documentos estructurados (páginas web) en un lenguaje de programación.

(1)Ecma International es una organización basada en membresías de estándares para la comunicación y la información. La organización fue fundada en 1961 para estandarizar los sistemas computarizados en Europa.

(2)El World Wide Web Consortium, abreviado W3C, es un consorcio internacional que produce recomendaciones para la World Wide Web. Está dirigido por Tim Berners-Lee.

Aunque la tendencia actual de las compañías es el cumplimiento de los estándares internacionales ECMA-262 y DOM de W3C, estas siguen definiendo sus propios modelos de acceso a la interfaz de usuario y creando sus propias extensiones del DOM.

1.3. Principios básicos de POO en JavaScript

A continuación, presentamos un pequeño recorrido por los principios básicos de la programación orientada a objetos, haciendo un análisis específico relativo a JavaScript y el soporte que ofrece para estos:

1) Abstracción

Este concepto, muy ligado al de encapsulación, se refiere a la representación mediante propiedades y métodos de una realidad en función de sus características y su funcionalidad.

Uno de los objetivos de la abstracción es poder gestionar de manera más cómoda la complejidad de un modelo o problema real, descomponiendo el global en componentes más pequeños y que, por tanto sean, más sencillos.

2) Encapsulación

Mientras que la abstracción define un modelo, la encapsulación oculta la parte deseada de su implementación al exterior, de manera que desde fuera no tengamos que preocuparnos de cómo está realizado, sino de las propiedades y los métodos que nos ofrece para interactuar.

Este mecanismo nos permite realizar incluso modificaciones sobre una clase sin que repercuta en el resto del programa. Imaginemos que refactorizamos un método de una clase que tenía un mal rendimiento, de manera que terminamos disponiendo de otros tres que hacen su misma función pero de modo más eficiente. Dado que el resto del programa se comunicaba con esta clase a través de los métodos expuestos gracias a la encapsulación, mientras el método público no cambie el resto del programa no sufre los cambios internos de la clase.

Refactorizar

Realizar ajustes de código internos en los métodos de una clase sin que afecte a cómo se comunica con el resto del programa y que, por tanto, no haya que cambiar código adicional.

En el caso de JavaScript, la encapsulación se logra gestionando las propiedades y los métodos que se van a exponer al exterior, lo cual se puede lograr de diferentes maneras. La realidad es que la falta de modificadores de acceso hace que no sea una tarea directa como en el caso de otros lenguajes, cosa que veremos en el segundo apartado de este módulo.

3) Herencia

El concepto de herencia se basa en crear objetos más especializados a partir de otros modelos existentes. Este principio está muy ligado también al de abstracción e incluso al de encapsulación a través de los modificadores de acceso.

En JavaScript, y debido a su naturaleza basada en prototipos, esta herencia se logra a través de los ellos, de manera que un objeto refiera como su prototipo a otro (su antecesor) a través de su propiedad `prototype`.

4) Polimorfismo

El concepto de polimorfismo está absolutamente ligado al de herencia, y se basa en que dos objetos de diferente tipo puedan compartir una misma interfaz, de manera que se pueda trabajar con esa interfaz abstrayéndose independientemente del tipo del objeto con el que se trabaja.

Existen diferentes tipos de polimorfismo:

- **Polimorfismo *ad hoc***
  A través de la sobrecarga (*overloading*) de métodos que permite disponer de varios métodos con el mismo nombre, pero que a través del contexto en el que se llaman (por ejemplo, el tipo del parámetro que recibe) pueden variar su comportamiento.
  Pero en realidad este tipo de polimorfismo se suele resolver en tiempo de compilación y, en el caso de JavaScript, debido a que es un lenguaje sin tipado estricto, se puede simular implementando en la función o el método de forma manual esa distinción, por ejemplo mediante el uso de la función `typeof`.

  Un ejemplo de polimorfismo *ad hoc* incorporado en JavaScript de forma propia son algunos de sus operadores que permiten la sobrecarga, como por ejemplo el operador de suma que funciona tanto para números como para cadenas de caracteres.

- **Polimorfismo paramétrico (*generics* en Java o *templates* en C++)**
  Permite a una función o un tipo de datos ser escrito de forma genérica, de manera que pueda manejar valores de forma idéntica sin depender de su tipo.
  El hecho de que en JavaScript no se disponga de un tipado estricto puede dar lugar en determinadas situaciones a soportar este tipo de polimorfismo

```
function log (arg) {
   console.log(arg);
}
```

  aunque no lo hace de manera completa, al contrario que TypeScript, un superconjunto de JavaScript creado por Microsoft que permite ser compilado a JavaScript puro y que adelanta en muchos aspectos las capacidades y la forma de trabajar con POO del futuro estándar ECMAScript 6[3] con el que trata de estar muy alineado.
  TypeScript añade funcionalidades a JavaScript, entre ellas el uso de tipos y objetos basados en clases. Está pensado para ser usado en proyectos grandes en los que la excesiva "flexibilidad" de JavaScript puede ser un problema.

- **Subtipado o polimorfismo clásico**
  Es la forma más habitual de polimorfismo y se basa en que el hecho de que dos objetos compartan un mismo padre a través de la herencia, nos permite trabajar con los métodos disponibles en el padre sin tener que preocuparnos del tipo de objeto que en realidad estamos manipulando, y estos pueden tener diferente comportamiento gracias a la sobreescritura (*overwriting*) de métodos.

[3] Se puede ver la tabla actual de compatibilidades por navegadores de este estándar en http://kangax.github.io/compat-table/es6/.

1.4. Implementación de POO en JavaScript

En los siguientes puntos, haremos un recorrido por las implementaciones de los aspectos más importantes de POO en JavaScript, haciendo especial hincapié en buenas prácticas y en las ventajas e inconvenientes de algunas implementaciones alternativas existentes.

Nota

Aunque hoy en día existen múltiples enfoques de trabajo con POO en JavaScript (cada una de las cuales con sus ventajas e inconvenientes) por cuestiones didácticas se ha elegido la aproximación más clásica, a pesar de que JavaScript no dispone a día de hoy de todos los elementos deseables para simplificar su uso tales como palabras reservadas para las estructuras convencionales, métodos especiales, modificadores de acceso, etc.

1.4.1. El concepto de clase y constructor

Dado que JavaScript no cuenta con la palabra reservada `class` para crear esta estructura básica de la POO, se emplean funciones (que no dejan de ser objetos) para obtener el mismo resultado en cuanto a funcionalidad.

A estas funciones se las denomina funciones constructoras, y lógicamente pueden ser nativas (como por ejemplo la clase Date) o definidas por el usuario.

Nota

Por comodidad, de aquí en adelante se empleará el término **clase** (aunque como tal no exista esta estructura de datos en JavaScript) para referirse a esas funciones constructoras, nativas o definidas por el usuario.

La forma, por tanto, de definir una clase en JavaScript es creando una función que la represente, que por convención se nombra con la primera letra en mayúscula, lo que ya establece una forma de diferenciarlas de funciones normales del lenguaje o de nuestra aplicación.

```
function Animal () {
}
```

Este fragmento de código pondría a nuestra disposición la clase Animal, que en este punto de nuestro ejemplo no contaría con ninguna propiedad ni método.

Además JavaScript tampoco cuenta con una palabra reservada para indicar el método constructor de forma explícita como lo hacen otros lenguajes, por lo que la propia función que define la clase se convierte en constructor de la misma. Es decir, Animal es además la función constructora de la clase Animal.

1.4.2. El concepto de objeto. Instanciación

Recordemos que una clase es un modelo que presenta propiedades (características) y métodos (funcionalidades) a partir del cual se generan objetos que, por lo tanto, son representaciones particulares de esta clase. Ese proceso de creación de un objeto a partir de una clase se denomina instanciación.

Para instanciar un objeto en JavaScript a partir de una clase, se emplea el operador `new` al que se acompaña la clase o función constructora a partir de la que se quiere crear el objeto. El proceso interno que realiza este operador internamente consta de 4 pasos, y es de especial interés el segundo:

1) Crea un nuevo objeto, del tipo nativo Object.

2) Vincula el nuevo objeto a la clase a través de su propiedad especial `__proto__` (o `[[prototype]]`), a la que le asigna el prototipo de su clase. De esta manera el objeto instanciado podrá acceder a la clase de la que proviene a través de su propiedad `prototype`.

3) Hace una llamada a la función constructora a partir del objeto creado y, por tanto, ejecuta todas las instrucciones que se encuentran dentro de esta función constructora.

4) Devuelve el nuevo objeto creado.

Debido a que JavaScript es un lenguaje basado en prototipos, los mecanismos de creación de objetos así como la herencia dependen ampliamente de estas dos propiedades: `__proto__` y `prototype`. En el siguiente apartado, se profundizará algo más en estos términos.

Por lo tanto, si queremos crear una instancia de nuestra clase Animal, ampliaremos nuestro ejemplo con el siguiente código:

```
function Animal () {
}


var miAnimal = new Animal();
```

Analizando lo que implica en el punto 2 anteriormente citado, el resultado de su interpretación sería que el objeto `miAnimal` dispone de una propiedad `__proto__` no accesible directamente, que referencia a `Animal.prototype` y que es la que permite que desde el objeto se pueda acceder a las propiedades y los métodos de la clase, que son compartidas a través de la propiedad `prototype` de esa clase.

Además, en este ejemplo, y dado que el constructor está vacío, el punto 3 anteriormente indicado no lleva a cabo ninguna acción, ya que no hay sentencias en el interior de la función constructora, pero lo habitual es incluir en esa función constructora la inicialización de las propiedades del objeto, ya sea directamente o mediante la llamada a un método que realice la acción.

1.4.3. Prototype y __proto__

Aunque estos aspectos serán notablemente más importantes cuando tratemos la herencia, es interesante tener claro desde el primer momento qué son y cómo interactúan entre ellos.

Web recomendada

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto

Simplificando podríamos decir que `__proto__` es la referencia que tiene cada instancia de una clase al `prototype` de la esta, y es la propiedad que emplea para poder acceder a las propiedades y métodos definidos en la clase mediante `prototype`.

Actualmente se desaconseja el acceso directo a esta propiedad en beneficio de `Object.create` y `Object.getPrototypeOf`.

1.4.4. Las propiedades

Las propiedades permiten definir las características de un objeto y personalizarlo para diferenciarse de otros, por lo que no deben pertenecer al prototipo sino al objeto en sí mismo.

Dependiendo del criterio que se tome en POO con JavaScript, podemos dividir las propiedades en diferentes tipos. Si como criterio marcamos la pertenencia, tendremos la siguiente clasificación:

- Propiedades estáticas o de clase.
- Propiedades de objeto (las más habituales y comúnmente referidas como propiedades a secas).

Por otro lado, dentro de las propiedades de objeto, si el criterio es la accesibilidad, podemos dividirlas en:

- Propiedades públicas (accesibles externamente).
- Propiedades privadas (accesibles solamente dentro de la propia clase).

Propiedades estáticas o de clase

En JavaScript, gracias a que las funciones no dejan de ser objetos, y al igual que en otros lenguajes, las clases pueden tener vinculadas directamente propiedades, que se suelen denominar estáticas, y que se comparten a través de la clase entre todas sus instancias, y son almacenadas solamente en un lugar, la clase, y accesibles a través de su nombre.

La forma de definir estas propiedades es hacerlo directamente sobre la clase y no sobre su prototipo. A continuación, se muestra un ejemplo típico del uso de este tipo de propiedades para una hipotética clase que albergue utilidades matemáticas:

```
function MathUtils() {
}

MathUtils.factorRadianes = Math.PI/180;
MathUtils.factorGrados = 180/Math.PI;

var grados = 38;
console.log('Radianes: ' + grados * MathUtils.factorRadianes );
// output: 0.6632
```

Propiedades públicas

Este tipo de propiedades de objeto se definen en el propio objeto, y para ello se hace uso de la palabra reservada `this`, que nos permite tener acceso a un objeto dentro de cualquiera de sus métodos.

El lugar habitual para declarar estas propiedades es el constructor de la clase, es decir, en la función constructora si hablamos de JavaScript. Obviamente, y como cualquier otra función, nuestro constructor podría recibir parámetros, por ejemplo con el objetivo de inicializar algunas propiedades:

```
function Animal (nombre) {
   this.nombre = nombre;
   this.peso = 0;
   this.altura = 0;
}

var miAnimal = new Animal('Usted');
```

Esto haría que `miAnimal` contara automáticamente en su interior con estas tres propiedades, que inicializara la propiedad nombre con el parámetro que recibe, y que además por ser públicas fuesen accesibles desde la propia instancia:

```
function Animal (nombre) {
   this.nombre = nombre;
   this.peso = 0;
   this.altura = 0;
}

var miAnimal = new Animal('Usted');
miAnimal.peso = 30;
miAnimal.altura = 50;
```

Como podemos ver, las propiedades públicas son accesibles a través de la instancia simplemente accediendo a ellas a través de su nombre mediante la sintaxis de punto, aunque también serían accesibles de la siguiente manera:

```
var miAnimal = new Animal('Usted');
miAnimal['peso'] = 30;
miAnimal['altura'] = 50;
```

Esta notación nos permite acceder a una propiedad de un objeto de forma dinámica, ya que el contenido expuesto entre los corchetes puede ser una expresión evaluada en tiempo de ejecución.

```
var miAnimal = new Animal("Usted");
miAnimal["pe"+"so"] = 30;
```

Nótese que en el ejemplo se usan comillas dobles para hacer notar que en este caso no existe diferencia en su elección para indicar un literal de cadena de caracteres.

Salvo que se tenga que acceder a una propiedad o método de forma dinámica, siempre es aconsejable emplear la notación del punto antes que la notación dinámica, ya que es más rápida y más legible.

Es importante comprender que las propiedades nombre, peso y altura se asignan y se crean para cada instancia en su constructor (por usar la palabra reservada `this`), y por tanto no son compartidas a través del prototipo entre las diferentes instancias de la clase Animal. Esto permite que cada instancia u objeto pueda darles un valor, sin afectar al resto, ya que si se hiciera sobre el `prototype` de la clase las modificaciones automáticamente se aplicarían a todas las instancias creadas a partir de esa clase.

La consecuencia de este comportamiento es que el consumo de memoria para las propiedades de objeto es individual para cada objeto instanciado, mientras que el consumo de memoria de las propiedades y los métodos del prototipo es compartido para todas las instancias.

Propiedades privadas

El objetivo de las propiedades privadas es mantener su acceso restringido a la funcionalidad interna de la propia clase y que no sea accesible desde el exterior.

En JavaScript no existen propiedades privadas como tal, pero se puede lograr esta característica utilizando lo que se denominan "closures" para, a través del ámbito de las variables, obtener resultados similares. Para ello, por tanto, se definen variables locales a la función constructora a través de la declaración mediante `var`, de manera que no sean accesibles salvo desde el ámbito de la función.

Podríamos disponer en nuestro ejemplo de una propiedad privada que nos permita controlar los años de vida de nuestro animal:

```
function Animal () {
   var edad = 0;

   this.nombre = '';
   this.peso = 0;
   this.altura = 0;

}

var miAnimal = new Animal();
console.log(miAnimal.peso); // mostraría "0" por consola
console.log(miAnimal.edad); // error, undefined no es accesible
```

Este ejemplo muestra cómo la propiedad `edad` no es accesible a través de la instancia, ya que su uso está fuera del ámbito interno de la propia clase. Es muy importante tener en cuenta que el uso de propiedades privadas no es algo excesivamente natural en JavaScript por la naturaleza que tienen los propios objetos en el lenguaje, y que existen multitud de aproximaciones para lograr este efecto, cada una con sus pros y sus contras.

Web recomendada

http://www.2ality.com/2012/03/private-data.html

Entre las diferentes opciones que encontramos para gestionar esta característica en JavaScript, aparece como opción MUY válida el uso de una convención en la nomenclatura a través del prefijo "_" para identificar las variables, aunque realmente sean públicas. Más adelante, cuando analicemos los diferentes tipos de métodos, profundizaremos en las formas de acceso interno a una propiedad privada.

Para reforzar este enfoque, cabe destacar que TypeScript omite el uso de propiedades privadas una vez que el código es compilado a JavaScript.

Ejemplo

Ejemplo clase Animal con propiedad privada edad en TypeScript:

http://www.typescriptlang.org/Playground#src=class%20Animal%20%7B%0A
%20%20%20%20private%20edad%3A%20number%20%3D%200%3B%0A
%09public%20nombre%3A%20string%20%3D%20''%3B%0A
%09public%20peso%3Anumber%20%3D%200%3B%0A
%09public%20altura%3Anumber%20%3D%200%3B%0A%7D%0A%0Avar
%20miAnimal%20%3D%20new%20Animal()%3B

1.4.5. Los métodos

Los métodos permiten definir la funcionalidad de la clase e interactuar con los objetos instanciados a partir de esta, para poder dar vida al modelo real que pretende representar nuestro programa. Al igual que ocurría con las propiedades, dispondremos de:

- Métodos estáticos.
- Métodos de objeto que a su vez se dividen en:
  - públicos,
  - privados.

Veremos además un tercer tipo de método denominado privilegiado, que surge en JavaScript como consecuencia de las limitaciones en el uso de propiedades privadas en la implementación de la POO clásica.

Métodos estáticos

Al igual que contábamos con propiedades estáticas vinculadas a la clase, es posible definir métodos estáticos que pueden ser llamados directamente a través del nombre de la clase. Continuando con nuestro ejemplo de utilidades matemáticas, tendríamos:

```
function MathUtils() {
}

MathUtils.factorRadianes = Math.PI/180;
MathUtils.factorGrados = 180/Math.PI;

MathUtils.redondearDecimales = function(num, digits)
{
      return Math.round(num * Math.pow(10,digits)) / Math.pow(10,digits);
}

console.log( MathUtils.redondearDecimales(1.123456789, 3) );
// output: 1.123
```

Métodos públicos

Los métodos públicos definidos en una clase tienen como finalidad dotar al objeto de una interfaz externa sobre la que podrán interactuar otras partes de nuestro programa.

Continuando con nuestro ejemplo de Animal, podremos definir en la clase dos métodos públicos nacer y crecer de la siguiente manera:

```
function Animal (nombre) {
   var edad = 0;

   this.nombre = nombre;
   this.peso = 0;
   this.altura = 0;
}

Animal.prototype.nacer = function (peso, altura) {
   this.peso = peso;
   this.altura = altura;
}

Animal.prototype.crecer = function(peso, altura) {
   this.peso += peso;
   this.altura += altura;
}

var miAnimal = new Animal('Usted');
miAnimal.nacer(2, 10); // nace con 2 kg y 10 cm
miAnimal.crecer(5, 10); // crece 5 kg y 10 cm
```

Es importante entender que los definimos en la clase a través de su propiedad prototype para que estos métodos estén presentes en todas las instancias de la clase, ya que como dijimos esa propiedad prototype de la clase es la que se asigna a la propiedad __proto__ del objeto instanciado y, por tanto, da acceso a estos métodos.

Para llamar a los métodos, y al igual que ocurría con las propiedades, simplemente tenemos que emplear su nombre con la notación del punto, y aunque es algo muy poco frecuente, también sería viable hacerlo con la notación dinámica con corchetes:

```
var miAnimal = new Animal("Usted");
miPerro['nacer'](2, 10);
```

Métodos privados

Los métodos privados de una clase son métodos que están diseñados para ser empleados por la propia clase y no de manera externa. Son métodos, por tanto, que no forman parte de la interfaz que queremos exponer al exterior, sino de apoyo a otros métodos.

Al igual que para lograr disponer de propiedades privadas se empleaban variables locales a la función constructora, para lograr métodos privados se emplean funciones locales en este constructor.

```
function Animal (nombre) {
   var edad = 0;
   var controlEdad = -1;

   this.nombre = nombre;
   this.peso = 0;
   this.altura = 0;

   function envejecer() {
      edad += 1;
   }

   this.nacer(2, 10);
   controlEdad = setInterval(envejecer, 10000);
}

Animal.prototype.nacer = function (peso, altura) {
   this.peso = peso;
   this.altura = altura;
}

Animal.prototype.crecer = function(peso, altura) {
   this.peso += peso;
   this.altura += altura;
}

var miAnimal = new Animal('Usted');
miAnimal.crecer(5, 10); // crece 5 kg y 10 cm
```

En este ejemplo hemos introducido diversos cambios que son importantes para reforzar los problemas que se derivan del uso de métodos y propiedades privados, al lograr esta funcionalidad a través de *closures*, jugando con el ámbito de las variables y funciones.

Por un lado, hemos declarado una propiedad privada (variable local al constructor) denominada `controlEdad`, para disponer de un intervalo que nos permita automatizar el envejecimiento del animal. Por otro lado, hemos creado un método privado (función local al constructor) denominado `envejecer` que lo que hace es sumarle 1 a la propiedad privada `edad`.

> Para declarar un método privado no se usa la palabra reservada **this**, ya que esto haría que el método fuera accesible para las instancias de la clase. Esta es la causa de que se emplee una función local.

Finalmente vemos que en el propio constructor hemos llamado a la función `nacer`, y justo inmediatamente hemos lanzado y guardado en `controlEdad` el envejecimiento mediante la función `setInterval` de JavaScript, que nos permite ejecutar una función de manera periódica (el intervalo lo marca el tiempo en milisegundos pasado como segundo parámetro).

Lo lógico hubiera sido haber lanzado el intervalo a `envejecer` dentro de la función `nacer`, pero a causa de los ámbitos el método privado `envejecer` no sería accesible desde el método público `nacer`, algo que para nada tiene que ver con los criterios de POO que encontramos en otros lenguajes.

Sin embargo, dentro del constructor sí que tenemos acceso a este método ya que está dentro de su ámbito o *scope*. De hecho la propiedad `controlEdad` también es perfectamente accesible al estar exactamente en el mismo ámbito o *scope* en el que nos encontramos.

Además dado que `envejecer` está en el mismo ámbito que la propiedad `edad` (ambos locales al constructor), desde dentro del método `envejecer` podemos tener acceso a la propiedad `edad` sin problema, ya que aunque no la encuentra dentro del ámbito envejecer, subiendo un nivel hacia arriba en la jerarquía de ámbitos llega al constructor en el que efectivamente está definida.

Sin embargo, y como se puede intuir, desde los métodos públicos declarados en el `prototype` tampoco tendríamos acceso a las propiedades privadas, por lo que la propiedad `edad` no sería accesible más que desde el método `envejecer` y ningún otro método de la clase o incluso el resto del programa podría tener acceso a este valor.

Otro gran problema de esta implementación es que no tendríamos acceso al propio objeto (propiedades públicas y métodos públicos) desde los métodos privados, ya que en el interior de ellos la palabra reservada `this` hace alusión al propio método (por ser una función y por ende un objeto) y no al objeto en el que se encuentra incluida la función.

La forma de salvar este problema sería añadir una closure en el constructor, como se ve en el siguiente ejemplo simplificado:

```
function MiClase()
```

```
{
   var privada = 'privada';
   var _this = this;

   this.publica = 'publica';

   function metodoPrivado() {
      privada = nuevoValor;

      _this.publica = nuevoValor;
      _this.metodoPublico();
   }
}

MiClase.prototype.metodoPublico = function() {
}
```

De esta manera, desde el método privado usamos `_this` para acceder a los métodos y propiedades públicos de la clase.

Métodos privilegiados

Los métodos privados de una clase son métodos que están diseñados para ser empleados conjuntamente con propiedades privadas, de manera que solventen algunos de los problemas anteriormente comentados en el acceso a estas propiedades.

Nota

Debido a los complejos problemas que conlleva el uso de métodos privados a modo de funciones locales en el constructor, los métodos privilegiados son la aproximación recomendada para esta asignatura cuando se requiera trabajar con propiedades privadas.

Otra aproximación aceptada será el uso de la nomenclatura con prefijo de guión bajo para especificar propiedades y métodos privados, aunque en realidad se apliquen en el prototipo y, por tanto, sean públicas.

Se basan en declarar los métodos como propios del objeto dentro del constructor mediante la palabra reservada `this`, de manera que a través del ámbito local puedan tener acceso a las propiedades privadas y a su vez puedan ser usados en métodos declarados en el prototipo, fuera del constructor o directamente a través de la instancia (ya que estos métodos no dejan de ser públicos).

```
function Animal (nombre) {
   var edad = 0;
   var controlEdad = -1;

   this.nombre = nombre;
   this.peso = 0;
   this.altura = 0;
```

```
    this.getEdad = function() {
       return edad;
    }

    this.setEdad = function(nuevaEdad) {
       edad = nuevaEdad;
    }

    this.getControlEdad = function() {
       return controlEdad;
    }

    this.initControlEdad = function(peso, altura) {
       var _this = this;
       controlEdad = setInterval( function() {
          _this.envejecer();
       }, 10000);
    }
}

Animal.prototype.envejecer = function() {
   this.setEdad( this.getEdad()+1 );
}

Animal.prototype.nacer = function (peso, altura) {
   this.peso = peso;
   this.altura = altura;
   this.initControlEdad();
}

Animal.prototype.crecer = function(peso, altura) {
   this.peso += peso;
   this.altura += altura;
}

var miAnimal = new Animal('Usted');
miAnimal.crecer(5, 10); // crece 5 kg y 10 cm
```

La contrapartida del uso de estos métodos es que al ser creados en la instancia y no en el prototipo (por declararse en `this`) tienen un consumo mayor de memoria en función de la cantidad de instancias creadas a pesar de que solucionan en buena parte los graves problemas derivados del uso de métodos privados como funciones locales del constructor.

Pérdidas del ámbito de la clase

En el ejemplo anterior, se da la necesidad de usar una *closure* para no perder el ámbito de la clase al aplicar una función como `setInterval` sobre un método de la clase (lo mismo ocurriría si asignamos una llamada a un método de una clase como consecuencia de un evento, por ejemplo, al clic de un botón).

Para ello se declara una variable local `_this` a la que se asigna el objeto mismo a través de `this`, y de esta manera sí que podremos tener acceso a esta variable local desde la función anónima y, por tanto, no perder el ámbito de la clase.

1.4.6. Herencia

La herencia persigue el objetivo de especialización, es decir, que una o varias clases se beneficien de las propiedades y los métodos de una clase existente para, en vez de partir de cero, poder simplemente especializarse en determinados aspectos.

JavaScript soporta la herencia como mecanismo para la aplicación de POO en nuestros programas, y para ello hace uso de los prototipos que ya hemos ido comprendiendo en los puntos anteriores.

El modo como se implementa la herencia es asignando al prototipo de una clase (clase hija) una instancia de la clase de la que se desea heredar (clase padre). Obviamente esta aproximación refleja la incapacidad de disponer de herencia múltiple en JavaScript, al no poder asignar a un mismo prototipo dos elementos diferentes; no obstante existen diferentes aproximaciones más complejas para lograr otros tipos de herencia.

Tipos de herencia

Tipos de herencia en JavaScript: http://javascript.crockford.com/inheritance.html

Si quisiéramos contemplar en nuestro ejemplo una clase Perro, que obviamente comparte las circunstancias generales para todos los animales, podríamos extender mediante herencia esta clase Animal, en vez de declarar de nuevo todas sus propiedades y sus métodos.

A continuación, se muestra un ejemplo de este proceso que se centra exclusivamente en las partes de código que intervienen en el proceso de herencia, es decir, la definición de la clase hija y la aplicación de la herencia sobre la clase Animal:

```
function Perro (nombre) {
   this.parent.constructor.call(this, nombre);
   }

Perro.prototype = new Animal();
Perro.prototype.constructor = Perro;
Perro.prototype.parent = Animal.prototype;

Perro.prototype.ladrar = function() {
   console.log('guau');
}
```

```
var miPerro = new Perro('Usted');
miPerro.crecer(5, 10); // crece 5 kg y 10 cm
miPerro.ladrar();
```

Vamos a analizar cada uno de los apartados clave del proceso de herencia representado en el código anterior.

1) Como se puede observar, la declaración de la clase hija no tiene ninguna particularidad con respecto a la clase padre; no obstante, dado que se quiere aprovechar la funcionalidad del constructor de Animal, se hace una llamada a este constructor mediante:

```
this.parent.constructor.call(this, nombre);
```

2) A continuación, se aplica el mecanismo para lograr la herencia que, como vemos, simplemente asigna al `prototype` de la clase hija una instancia de la clase padre. Es fundamental hacer esto en segundo lugar, para que el resto de operaciones sobre el `prototype` de la nueva clase sean "añadidos" o "sobreescrituras" sobre las propiedades y los métodos de la clase padre.

Si hiciéramos esta operación en último lugar, en realidad podríamos "machacar" métodos de la clase hija con los de la clase padre, que es justamente lo contrario de lo que se persigue.

La herencia en ese sentido funciona de manera similar a los ámbitos o *scopes*, ya que busca una propiedad o un método en la clase actual, y si no la encuentra sube jerárquicamente a través de su prototype a la clase padre para buscarla allí, y así sucesivamente.

3) Posteriormente se corrige el constructor de la clase hija, ya que al aplicar al `prototype` de una clase la herencia se sobrescribe el método y, por tanto, es necesario hacer el ajuste correspondiente si queremos evitar que el constructor de la clase hija sea el de la clase padre (en este caso **sin esta línea** el constructor de la clase Perro sería la función Animal).

4) Guardamos una referencia a la clase padre para que esta sea accesible desde el constructor y métodos de la clase hija, para poder lograr, por ejemplo, la llamada al constructor de Animal desde el constructor de Perro que se citó en el paso 1 (este paso es para disponer de algo similar a *super* existentes en otros lenguajes OO).

5) Finalmente, declaramos todos los métodos públicos de la nueva clase a través de su `prototype`. Estos métodos pueden ser nuevos y, por tanto, extender la funcionalidad de la clase padre, o pueden ser métodos existentes en la clase padre y, por tanto, estaríamos sobrescribiéndolos para tener una funcionalidad diferente a través de los mismos métodos (base del concepto de polimorfismo).

Como podemos ver en la parte del código en la que se instancia un objeto de la clase Perro, este objeto no solamente cuenta con sus propios métodos (ladrar), sino que cuenta con los métodos de la clase padre (crecer).

Los beneficios del uso de herencia en los casos aplicables son muchos, pero entre todos ellos destacamos la agilidad para dotar de funcionalidad a varias clases de nuestro código cuando la ampliación que se requiere afecta a las clases superiores en la jerarquía de herencia, ya que con aplicar el código común en ellas se traslada automáticamente a todas las hijas (tantos niveles de profundidad como tenga la herencia).

Esta misma característica es la que hace que la herencia casi siempre permita una reducción de la cantidad de nuestro código (gracias a la reutilización de los métodos heredados) y, por tanto, aumente la legibilidad y manejo de nuestras aplicaciones.

1.4.7. Polimorfismo

Como veíamos al hablar de herencia, una de las posibilidades que nos da su implementación en JavaScript es contar con el llamado *polimorfismo subtipado o clásico*.

Este tipo de polimorfismo nos permite trabajar con un objeto a un determinado nivel sin tener que preocuparnos del tipo exacto de este, gracias al trabajo con él a un nivel superior (clase padre) que garantiza que gracias a la herencia siempre contemos con los métodos a los que llamamos, ya sean los originales de la clase padre o las versiones sobrescritas en las clases hijas.

En cualquier caso, el beneficio de este mecanismo es que el propio lenguaje es quien resuelve esta problemática, sin que al nivel que es empleado en nuestro código nos tengamos que preocupar por estas comprobaciones de tipos. Veamos un ejemplo para comprenderlo mejor:

```
function MiApp() {
   this.pantallaActual = null;
}
MiApp.PANTALLA_MENU = 'MENU';
MiApp.PANTALLA_DETALLE = 'DETALLE';

MiApp.prototype.inicializar = function() {
   this.actualizarPantalla( MiApp.PANTALLA_MENU );
```

```
}

MiApp.prototype.actualizarPantalla = function(nuevaPantalla) {
   if (this.pantallaActual === null || nuevaPantalla !== this.pantallaActual.getNombre() )
   {
      if (this.pantallaActual !== null) {
         this.pantallaActual.ocultar();
         this.pantallaActual.destruir();
      }

      switch (nuevaPantalla)
      {
         case MiApp.PANTALLA_MENU:
         this.pantallaActual = new PantallaMenu();
         break;

         case MiApp.PANTALLA_DETALLE:
         this.pantallaActual = new PantallaDetalle();
         break;
      }

      this.pantallaActual.setNombre(nuevaPantalla);
      this.pantallaActual.mostrar();
   }
}

function Pantalla () {
   this._nombre = '';
}

Pantalla.prototype.setNombre = function(nombre) {
   this._nombre = nombre;
}

Pantalla.prototype.getNombre = function() {
   return this._nombre;
}

Pantalla.prototype.mostrar = function() {
      // Código para mostrar la pantalla
}

Pantalla.prototype.ocultar = function() {
   // Código para ocultar la pantalla
}

Pantalla.prototype.destruir = function() {
```

```
    // Código para eliminar la pantalla
}

function PantallaMenu() {
}
PantallaMenu.prototype = new Pantalla();
PantallaMenu.prototype.constructor = PantallaMenu;
PantallaMenu.prototype.parent = Pantalla.prototype;

function PantallaDetalle() {
}
PantallaDetalle.prototype = new Pantalla();
PantallaDetalle.prototype.constructor = PantallaDetalle;
PantallaMenu.prototype.parent = Pantalla.prototype;

var app = new MiApp();
app.inicializar();
```

El beneficio del polimorfismo en este ejemplo se centra en el método `actualizarPantalla` de la clase `MiApp`, a través del cual podemos ver cómo en varias ocasiones se hace uso de llamadas a métodos sobre un objeto del que no tenemos la certeza de qué tipo es pero que se resuelven correctamente, ya que unificamos su uso a través de los métodos de la clase padre y que, por tanto, garantiza que sea cual sea el tipo de la instancia (`PantallaMenu` o `PantallaDetalle`) tendremos a nuestra disposición los métodos empleados (todos de la clase padre `Pantalla`).

Para controlar manualmente estas situaciones, JavaScript cuenta con la función `instanceof` que nos permite saber si un objeto es instancia de una clase, cumpliéndose siempre que una instancia de una clase hija siempre se considera instancia también de la clase padre.

```
instanciaPadre instanceof ClasePadre; // true
instanciaPadre instanceof ClaseHija; // false
instanciaHija instanceof ClasheHija; // true
instanciaHija instanceof ClasePadre; // true
```

En este ejemplo, y por simplificar su código, las clases `PantallaMenu` y `PantallaDetalle` no personalizan los métodos `mostrar` y `ocultar`, pero podrían hacerlo sin afectar al funcionamiento del programa, de manera que cada una de ellas cambie la forma en la que se muestra u oculta simplemente sobrescribiendo los métodos.

En otros lenguajes, este mecanismo también se puede lograr mediante el uso de interfaces (mecanismo de POO del que JavaScript como tal no tiene soporte, no así por ejemplo TypeScript).

1.4.8. Composición

La composición es un mecanismo que tiene lugar en POO cuando una clase contiene como propiedades objetos de otras clases. Este concepto se puede observar en el anterior ejemplo, en el que la clase `MiApp` dispone de una propiedad `pantallaActual` que es de tipo `Pantalla`, por lo tanto entre `MiApp` y `Pantalla` se está dando una relación de composición.

Dependiendo de las relaciones y condiciones del problema que se va a representar mediante POO, la composición puede ser un mecanismo válido como alternativa a la herencia y, en otras ocasiones, un mecanismo complementario a ella.

1.5. Alternativa al uso de constructores en POO

Además de la aproximación a la POO que hemos visto durante todo este apartado del módulo y que se asocia a la POO clásica, existe una alternativa que se suele conocer como *prototypal object oriented programming* o aproximación por prototipos que se basa en el uso exclusivo de prototipos en lugar de funciones constructoras.

Existen multitud de artículos entre la comunidad de desarrolladores JavaScript profesionales en la que se evalúan estas dos alternativas, y analizan los pros y contras de cada una de ellas sin existir un claro vencedor a día de hoy.

1.5.1. POO basada en prototipos y Object.create

La aproximación a POO por prototipos en JavaScript se basa en el método `create` de la clase `Object` que está disponible desde ECMAScript 5 y, aunque no cuenta con soporte en IE8, se puede subsanar este problema de forma manual, extendiendo mediante una función propia que realice la misma tarea, de manera que garanticemos que contamos con este método sea cual sea el navegador que usemos:

```
if (typeof Object.create !== "function") {
   Object.create = function (o) {
      function F() {}
      F.prototype = o;
      return new F();
   };
}
```

Object.create

Object.create:

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Object/create

Soporte de Object.create (ECMAScript 5) en navegadores:

http://kangax.github.io/compat-table/es5/

Este método recibe un objeto como parámetro y devuelve un nuevo objeto que tiene el objeto original pasado como su `[[Prototype]]` (`__proto__`).

A partir de este método, se puede contemplar la POO básica como refleja el siguiente ejemplo:

```
var persona = {
   dni: '',
   nombre: '',
   amigos: null,

   init: function(dni, nombre) {
      this.dni = dni;
      this.nombre = nombre;
      this.amigos = new Array();

      return this;
   },

   saludar: function() {
      console.log('Hola me llamo '+this.nombre);
   }
   };

var mi = Object.create(persona).init('1', 'Marcos');
mi.saludar();
```

En el ejemplo, el objeto `Persona` (definido mediante una notación literal con formato JSON) se usa como prototipo de la instancia `mi`, de manera que `mi` contará con las propiedades y los métodos de `Persona`.

Se ha trabajado durante este módulo con el enfoque de POO clásica por considerarse más didáctica, requerir de una curva de aprendizaje menor para quienes vienen de otros lenguajes no basados en prototipos y porque está alineada con el futuro estándar ECMAScript 6.

Ampliando esta aproximación a la herencia, tendríamos un escenario como el siguiente:

```
var persona = {
   dni: '',
   nombre: '',
   amigos: null,

   init: function(dni, nombre) {
```

```
        this.dni = dni;
        this.nombre = nombre;
        this.amigos = new Array();

        return this;
    },

    saludar: function() {
        console.log('Hola me llamo '+this.nombre);
    }
    };

var estudiante = Object.create(persona);
estudiante.estudiar = function() {
    console.log(this.nombre + ' está estudiando.');
};

var yo = Object.create(estudiante).init('1', 'Marcos');
yo.saludar();
yo.estudiar();
```

Como podemos ver, toda la cadena jerárquica se basa en el uso de `Object.create` y los prototipos. Como es lógico, en torno a esta manera de implementar la POO existen métodos para disponer de propiedades y métodos privados, e incluso patrones de diseño que permiten disponer de herencia múltiple.

Otro detalle intencionado es que los nombres de las "clases" ahora van en minúsculas, porque este acercamiento defiende el uso de objetos como tales y, por tanto, deja de tener sentido la necesidad de aplicar la convención habitual en otros lenguajes al hablar de "clases".

Como complemento a la forma revisada en el anterior ejemplo, es importante destacar que el método `Object.create` admite un segundo parámetro que sería un objeto en notación JSON nuevamente, que permite extender precisa y directamente en la llamada a `Object.create` el objeto pasado como primer parámetro con propiedades y métodos adicionales, e incluso controlar algunos aspectos de la capacidad de visualización y acceso a estas propiedades y métodos. El ejemplo anterior podría ser más completo y representativo de la siguiente manera:

```
var persona = {
    dni: '',
    nombre: '',
    amigos: null,

    init: function(dni, nombre) {
```

```
        this.dni = dni;
        this.nombre = nombre;
        this.amigos = new Array();

    return this;
    },

    saludar: function() {
        console.log('Hola me llamo '+this.nombre);
    }
    };

    var estudiante = Object.create(persona, {
    numeroMatricula: {
        get: function() {
            return this.value;
        },
        set: function(newValue) {
            this.value = newValue;
        }
    },

    init: {
        value: function(dni, nombre, numeroMatricula) {
            persona.init.call(this, dni, nombre);
            this.numeroMatricula = numeroMatricula;
            return this;
        }
    },

    estudiar: {
        value: function() {
            console.log('estudiando > '+this.numeroMatricula);
        }
    }
});

    var yo = Object.create(estudiante).init('1','Marcos','12');
yo.saludar();
yo.numeroMatricula = '333';
yo.estudiar();
```

Si nos fijamos en él, vemos que hemos hecho las siguientes modificaciones:

1) Tanto las propiedades como los métodos pasados en el objeto que actúa como segundo parámetro se han de declarar a su vez como objetos literales. En el caso de los métodos, contienen una propiedad `value` que es la que alberga la función que representa al método.

2) Ahora un estudiante añade una propiedad pública que requiere del uso de un *getter* y un *setter* que con este formato se declaran como dos métodos `get` y `set` en el objeto que define esta nueva propiedad.

3) Se sobrescribe la función `init` que actúa como constructora, de manera que se amplía un parámetro y, en su interior, se llama a la función `init` del objeto extendido mediante el uso de la función `call` en la línea:

```
persona.init.call(this, dni, nombre);
```

4) Se extiende con un nuevo método `estudiar` el objeto original `persona` y, desde el método, se demuestra que se puede acceder a la nueva propiedad `numeroMatricula`.

Aunque no se ha representado en el ejemplo, este mecanismo o enfoque cuenta con lo que se denominan **descriptores de propiedad**, tales como: enumerable, writeable, configurable, value, get y set, y que son los mismos que se aplican a través de `Object.defineProperty`, que también es empleado en este enfoque.

Descriptores de propiedad en MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Object/defineProperty

Siguiendo este ejemplo, se pueden ver muchos paralelismos con la aproximación clásica, siendo las diferencias más importantes en cuanto al formato de declaración, y es que en realidad la única diferencia entre usar el operador `new` y `Object.create` es que el segundo NO hace la llamada a la función constructora que iba implícita en el caso del primero.

2. Objetos predefinidos

Como se ha comentado anteriormente, en JavaScript existe un conjunto de objetos incorporados que permite acceder a muchas de las funciones que se encuentran en cualquier otro lenguaje de programación. Nos referimos a los objetos: Object, Array, Boolean, Date, Function, Math, Number, String y RegExp.

En este módulo se van a presentar las propiedades más utilizadas y comúnmente soportadas por los principales navegadores.

Web recomendada

Es posible consultar en línea la especificación completa de los objetos predefinidos en el estándar ECMA-262.

2.1. Los objetos Object y Function

2.1.1. El objeto Object

Se trata del objeto padre o abuelo a partir del que heredan todos los objetos existentes en el lenguaje o que serán creados en el lenguaje. De esta manera, este objeto define las propiedades y métodos que son comunes a todos los objetos, por lo que cada objeto particular podrá rescribir métodos o propiedades si lo necesita para adecuarlo a su objetivo.

Un uso del objeto, aunque no resulta muy común o recomendable, es la utilización como método de creación alternativo de objetos, por ejemplo:

```
var coche = new Object();
coche.marca = "Ford";
coche.modelo = "Focus";
coche.aireAcon = true;
coche.muestra = function()
      {
      alert(this.marca +" "+this.modelo);
      }
```

En el anterior ejemplo se ha creado un objeto coche a partir de Object.

Otra de las estructuras que se generan mediante el uso de Object son los arrays asociativos. En éstos, a diferencia de los arrays básicos, cada elemento del array se referencia por el nombre que se le ha asignado y no mediante el índice que indica su posición.

Ved también

Los arrays asociativos se han tratado en el apartado 2 de este módulo.

A continuación, se presenta un ejemplo de su uso:

```
var direcciones = new Object();
```

```
direcciones["Víctor"] = "Santiago de Cuba";
direcciones["Pablo"] = "Madrid";
direcciones["Miguel"] = "Valencia";
```

de este modo, se puede recuperar la dirección de Miguel utilizando la siguiente sintaxis:

```
var dirMiq = direcciones["Miguel"];
```

Por lo tanto, como se puede observar, simplemente se simula un array mediante la asignación dinámica de propiedades a un objeto del tipo Object. Aunque sea un array asociativo y en el ejemplo se vea su acceso "dinámico", como objetos que son es más aconsejable usar la notación de punto.

Tabla 1. Propiedades del objeto Object

Nombre	Descripción
constructor	Referencia a la función que se ha llamado para crear el objeto genérico.
prototype	Representa el prototipo para la clase.

Tabla 2. Métodos del objeto Object

Nombre	Descripción
toString()	Devuelve al objeto una cadena, de manera predeterminada "[objeto Object]".
valueOf()	Devuelve el valor primitivo asociado con el objeto, de manera predeterminada "[objeto Object]".

2.1.2. El objeto Function

Es el objeto del que derivan las funciones de JavaScript; proporciona propiedades que transmiten información útil durante la ejecución de la función. Un ejemplo de estas propiedades es el array arguments[ ].

El constructor para el objeto Function es:

```
new Function (arg1, arg2..., argN, función)
```

donde:

- arg1, arg2..., argN: parámetros opcionales de la función.
- función: es una cadena que contiene las sentencias que conforman la función.

La propiedad arguments[ ] permite conocer el número de argumentos que se han pasado en la llamada a la función.

En el siguiente ejemplo, se crea una función, en cuyo interior existe un bucle for cuyo extremo superior está definido por el número de argumentos de la propia función, ya que el objetivo del bucle es manipular los argumentos pasados en la llamada a la función pero que a priori se desconoce su número.

```
function lista(tipo) {
   document.write("<" + tipo + "l>");
   for (var i=1; i<lista.arguments.length; i++) {
      document.write("<li>" + lista.arguments[i]);
  }
  document.write("</" + tipo + "l>");
}
```

La llamada a la función con los siguientes argumentos:

```
lista("U", "Uno", "Dos", "Tres");
```

tendría como resultado lo siguiente:

```
<ul>
<li>Uno
<li>Dos
<li>Tres
</ul>
```

Tabla 3. Propiedades del objeto Function

Nombre	Descripción
arguments	Array con los parámetros pasados a la función.
caller	Nombre de la función que ha invocado a la que se está ejecutando.
constructor	Referencia a la función que se ha llamado para crear el objeto.
length	Número de parámetros pasados a la función.
prototype	Valor a partir del que se crean las instancias de una clase.

La propiedad caller

Esta propiedad sólo es accesible desde el cuerpo de la función. Si se usa fuera de la propia función, su valor es null.

Tabla 4. Métodos del objeto Function

Nombre	Descripción	Sintaxis	Parámetros
apply	Invoca la función.	apply(obj[arg1,arg2..., argN])	obj: nombre de la función arg1..., argN: lista de argumentos de la función
call	Invoca la función.	call(obj[args])	obj: nombre de la función args: array de argumentos de la función

Nombre	Descripción	Sintaxis	Parámetros
toString	Retorna un string que representa al objeto especificado.	toString()	
valueOf	Retorna el valor primitivo asociado al objeto.	valueOf()	

2.2. Los objetos Array, Boolean y Date

2.2.1. El objeto Array

Nota

En el módulo "Programación en el lado del cliente: lenguajes script en los navegadores" vimos una introducción al objeto Array, en este apartado repasaremos y ampliaremos los conceptos vistos anteriormente.

Los arrays almacenan listas ordenadas de datos heterogéneos. Los datos se almacenan en índices enumerados empezando desde cero, a los que se accede utilizando el operando de acceso a arrays ([ ]).

Creación de arrays

En el módulo "Programación en el lado del cliente: lenguajes script en los navegadores" habíamos visto que el constructor del objeto admite las siguientes sintaxis:

```
var vector1 = new Array();
var vector2 = new Array(longitud);
var vector3 = new Array(elemento0, elemento1..., elementoN);
```

La primera sintaxis del constructor crea un array vacío sin una dimensión definida, en la segunda sintaxis se le pasa el tamaño del array como parámetro y en el tercer constructor se le pasan los valores que se almacenarán en el array.

Por ejemplo, se puede definir la longitud del array y después rellenar cada una de sus posiciones:

```
colores = new Array(16);
colores[0] = "Azul";
colores[1] = "Amarillo";
colores[2] = "Verde";
```

o bien, iniciar el array en el mismo momento de su creación:

```
colores = new Array("Azul", "Rojo", "Verde");
```

También existe la posibilidad de crear arrays utilizando literales de array, por ejemplo:

```
var vector1 = [];
var vector2 = [,,,,,,,,,,,,];
```

```
var vector3 = ["elemento0","elemento1"...,"elementoN"];
```

Acceso a los elementos

El acceso a los elementos de un array se realiza utilizando el nombre del array seguido del índice del elemento que consultar encerrado entre corchetes.

A continuación, se muestra un ejemplo de acceso a los elementos de un array:

```
var vector = new Array("Platano", "Manzana", "Pera");
var fruta1 = vector[0];
```

Añadir y modificar elementos a un array

Recordemos que en JavaScript el tamaño del array es gestionado directamente por el lenguaje y podemos añadir elementos indicando un índice sin necesidad de realizarlo de manera consecutiva. Podemos realizarlo de la siguiente manera:

```
var vector1 = [2, 54, 16];
vector1[33] = 25
```

Al modificar o añadir elementos a un array, es necesario tener en cuenta que los array son objetos o tipos por referencia, por lo que si un array está asignado a dos variables, la modificación de un elemento del array afectará a las dos variables y no sólo a aquélla desde la que se ha modificado:

```
var vector2 = [2, 4, 6, 8];
var vector3 = vector2;
vector3[0] = 1;
```

La asignación del valor 1 al array vector3 no sólo modifica el contenido de éste, sino también el contenido del array vector2.

Eliminar elementos de un array

Los elementos de un array se pueden eliminar utilizando la sentencia "delete":

```
delete vector1[3];
```

La sentencia anterior elimina el cuarto elemento del array vector1, en el sentido de que le asigna el valor "undefined", pero no modifica el tamaño del array.

Nota

"Delete" en realidad no borra si no que, en esa posición, deja un valor "undefined". Esto tiene connotaciones con lo que se puede esperar de la propiedad "length" tras usar "de-

lete". Para borrar y eliminar de verdad ese elemento se usa "splice". "Splice" se explica en detalle más adelante, ya que tiene otros usos además de eliminar.

La propiedad length

Esta propiedad nos devuelve la siguiente posición disponible al final del array, sin tener en cuenta los índices vacíos que haya en medio. La sintaxis es la siguiente:

```
longitud = vector1.length;
```

El método sort

El método sort requiere indicar la función de comparación para ordenar el array. Si no se especifica dicha función, los elementos del array se convierten a strings y se ordenan alfabéticamente. Por ejemplo, "Barcelona" estaría antes que "Zaragoza", y "80" antes que "9".

En el caso de no especificar la función de comparación, los elementos se ordenan según el retorno de esta función:

- Si compara(a,b) es menor que 0, b tendrá un índice en el array menor que a.
- Si compara(a,b) es 0, no modificará las posiciones.
- Si compara(a,b) es mayor que 0, b tendrá un índice en el array mayor que a.

La función de comparación tiene la siguiente forma:

```
function compara(a, b) {
   if (a < b por un criterio de ordenación)
      return -1;
   if (a > b por un criterio de ordenación)
      return 1;
   return 0; //son iguales
}
```

El criterio de ordenación definirá si la ordenación será numérica, alfanumérica o cualquiera que el programador pueda definir.

Para ordenar números, la función se define de la siguiente manera:

```
function compara(a, b){
   if (a < b)
      return -1;
   else if (a === b)
      return 0;
   else
      return 1;
```

```
}
```

En un array con los valores 1, 2 y 11, si se aplica el método sort() sin modificar la función de comparación, el resultado de la ordenación será 1, 11, 2, es decir, se ordenaran los valores alfabéticamente.

En el siguiente ejemplo se muestra cómo modificar la función de comparación:

```
numeros = new Array(1, 2, 11);
function compara(a, b){
   if (a < b)
      return -1;
   else if (a === b)
      return 0;
   else
      return 1;
}
function ordena(){
   numeros.sort(compara);
   document.write(numeros);
}
```

El resultado de este ejemplo sería el siguiente: 1, 2, 11.

Simulación de pilas LIFO

Una pila LIFO es una estructura que se utiliza para almacenar datos siguiendo el orden de último en entrar, primero en salir. La analogía sería una pila de documentos sobre una mesa de manera que el primer documento que retiramos para estudiar coincidirá con el último que depositamos en la mesa.

Para simular el uso de pilas con arrays se utilizan los métodos "push()" y "pop()"; cuando se invoca el método "push()" se añaden los argumentos dados al final del array y se incrementa el tamaño de éste, que se refleja en la propiedad "length".

El método "pop()" elimina el último elemento del array, lo devuelve y disminuye la propiedad "length" en una unidad.

En el siguiente ejemplo se pueden observar estos métodos en acción:

```
var pila = []; //[]
pila.push("primero");  //["primero"]
pila.push(15, 30);  //["primero",15, 30]
pila.pop();   //["primero",15] y devuelve el valor 30
```

El hecho de que estos dos métodos se utilicen para la simulación de pilas no implica que no se puedan utilizar los métodos para insertar y eliminar valores situados al final del array.

Simulación de colas FIFO

Una cola FIFO se basa en el orden de primero dentro, primero fuera. La analogía sería cualquier cola en el pago en un comercio o en la entrada al cine, etc. JavaScript dispone de los métodos "unshift()" y "shift()", que, a diferencia de los dos anteriores, añaden y eliminan datos del inicio de array.

De esta manera, "unshift()" introduce los argumentos al principio del array y provoca que los elementos existentes cambien a índices más altos, y como es de esperar aumentará el valor de la propiedad length.

Por su parte, "shift()" elimina el primer elemento del array, lo devuelve y reduce el índice del resto de elementos del array, y acaba con la disminución obligatoria de la propiedad "length".

En el siguiente ejemplo se observa su comportamiento:

```
var cola = ["Juan", "Pedro", "Andrés", "Vicente"];
cola.unshift("Claudia", "Raquel"); //cola contendrá ["Claudia","Raquel","Juan",
     //"Pedro", "Andrés", "Vicente"]
var primero = cola.shift();  //primero contendrá el valor "Claudia"
```

La simulación de una cola se realiza con la combinación de los métodos "push()", que añade elementos al final de la cola, y "shift()", que extrae de la cola el primer elemento de ésta:

```
var cola = ["Juan", "Pedro", "Andrés", "Vicente"];
cola.push("Claudia");            //cola contendrá ["Juan","Pedro", "Andrés","Vicente","Claudia"]
var siguiente = cola.shift();    //siguiente contendrá el valor "Juan"
```

Concatenación de arrays

El método concat() devuelve el array resultado de añadir los argumentos al array sobre el que se ha realizado la llamada. Por ejemplo:

```
var color = ["Rojo", "Verde", "Azul"];
var colorAmp = color.concat("Blanco", "Negro");
```

En el ejemplo anterior se obtiene un nuevo array "colorAmp" con el siguiente contenido: ["Rojo", "Verde", "Azul", "Blanco", "Negro"], mientras que el array "color" no se ha modificado.

Se pueden concatenar dos arrays simplemente si se cumple la condición de que el argumento del método concat() sea un array.

Conversión en una cadena

Para la conversión de un array en una cadena se utiliza el método "join()", que convierte el array en una cadena y permite especificar a través de su parámetro cómo se separarán los elementos de la cadena. Es común utilizar "join()" para mostrar los elementos de un array con un separador específico:

```
var color = ["Rojo", "Verde", "Azul"];
var cadena = color.join("-");
```

En el ejemplo anterior, la variable *cadena* adquiere el siguiente contenido: "Rojo-Verde-Azul".

Invertir el orden de los elementos

El método "reverse()" invierte el orden de los elementos de un array y, a diferencia de los dos métodos anteriores, el mismo array almacenará los elementos, ya con el orden invertido:

```
var color = ["Rojo", "Verde", "Azul"];
color.reverse();
```

En el ejemplo anterior el array "color", después de ejecutarse el método, tiene el siguiente contenido: ["Azul", "Verde", "Rojo"].

Extracción de fragmentos de un array

El método "slice()" devuelve un fragmento del array sobre el que se invoca; no actúa realmente sobre el array (tal como lo hacía "reverse()") y éste queda intacto. El método toma dos argumentos, el índice inicial y el final, y devuelve un array que contiene los elementos que se encuentran localizados entre el índice inicial y el final (excluyendo el final).

En el caso de que sólo reciba un argumento, el método devuelve el array que componen todos los elementos desde el índice indicado hasta el final de array.

Una característica interesante es el hecho de que admite valores negativos para los índices, y cuando éstos son negativos, se cuentan desde el final del array. En los siguientes ejemplos se observa el uso de este método:

```
var carrera = [21, 25, 12, 23];
carrera.slice(2);  //devuelve [12, 23]
carrera.slice(1,3);  //devuelve [25, 12]
```

```
carrera.slice(-2, -1)  //devuelve [12]
```

Añadir, eliminar y modificar elementos de un array

El método "splice()" se puede utilizar para añadir, reemplazar o eliminar elementos en un array y devuelve los elementos que se han eliminado. De los argumentos que puede tomar sólo el primero es obligatorio, y su sintaxis es:

```
splice(inicio, elementosBorrar, elementosAñadir);
```

El significado de los argumentos es el siguiente:

- inicio indica el índice a partir del que se va a realizar la operación.
- elementosBorrar es el número de elementos que se van a eliminar empezando por aquél marcado por el primer parámetro. Si se omite este parámetro, se eliminan todos los elementos desde el inicio hasta el final del array y, tal como se ha comentado, son devueltos (se podrán almacenar en una variable).
- elementosAñadir es una lista de elementos separados por comas, no obligatoria, que sustituirán a los eliminados.

A continuación, se muestra un ejemplo de su uso:

```
var carrera = [21, 25, 12, 23];
carrera.splice(2, 2);  //devuelve los elementos eliminados [12, 23] y el array
        //contendrá los valores [21, 25]
carrera.splice(2, 0, 31, 33); //no elimina ningún valor y por tanto devuelve []
//y añade los valores [31, 33] a la cadena
```

Arrays multidimensionales

Un array multidimensional es un array en el que cada elemento es a su vez un array. En el siguiente ejemplo se define un array bidimensional:

```
var matriz = [[1,2,3],[4,5,6],[7,8,9]];
```

El acceso a los arrays multidimensionales se realiza a partir de la concatenación de corchetes que indican los elementos a los que se accede:

```
matriz[0][1];  //devuelve el valor 2, ya que estamos accediendo al segundo
               //elemento del primer array
```

Uso de prototipos

Tal como se ha explicado, es posible añadir nuevos métodos y propiedades a cualquier objeto utilizando los prototipos. A continuación, se definirá un nuevo método "muestra()" que enseñará en una ventana el contenido de un array:

```
function mMuestra(){
   if (this.length != 0)
      alert(this.join());
   else
      alert("El array está vacío");
}
Array.prototype.muestra = mMuestra;
```

Para finalizar con el objeto Array, se muestra en una tabla las principales propiedades y métodos del objeto.

Tabla 5. Propiedades del objeto Array

Nombre	Descripción
constructor	Referencia a la función que se ha llamado para crear el array actual.
length	Número de elementos del array.
prototype	Representa el prototipo para la clase.

Tabla 6. Métodos del objeto Array

Nombre	Descripción	Sintaxis	Parámetros	Retorno
concat	Concatena dos arrays.	concat(array2)	array2: nombre del array que concatenar al que ha invocado al método.	Nuevo array unión de los dos.
join	Une todos los elementos del array en un string, separados por el símbolo indicado.	join(separador)	separador: signo que separará los elementos del string.	String.
pop	Borra el último elemento del array.	pop()		El elemento borrado.
push	Añade uno o más elementos al final del array.	push(elt1, ..., eltN)	elt1..., eltN: elementos que añadir.	El último elemento añadido.
reverse	Transpone los elementos del array.	reverse()		
shift	Elimina el primer elemento del array.	shift()		El elemento eliminado.
slice	Extrae una parte del array.	slice(inicio,final)	inicio: índice inicial del array que extraer. final: índice final del array que extraer.	Un nuevo array con los elementos extraídos.

Nombre	Descripción	Sintaxis	Parámetros	Retorno
splice	Cambia el contenido de un array, añadiendo nuevos elementos a la vez que elimina los ya existentes.	splice(índice, cuántos, nuevoEl1, ..., nuevoElN)	índice: índice inicial a partir del cual se empieza a cambiar. cuántos: número de elementos a eliminar. nuevoEl1..., nuevoElN: elementos que añadir.	Array de elementos eliminados.
sort	Ordena los elementos del array.	sort(compara)	compara: función que define el orden de los elementos.	
toString	Convierte los elementos del array a texto.	toString()		String que contiene los elementos del array pasados a texto.
unshift	Añade uno o más elementos al inicio del array.	unshift(elt1..., eltN)	elt1..., eltN: elementos que añadir.	La nueva longitud del array.

2.2.2. El objeto Boolean

Es el objeto incorporado en el lenguaje que representa los datos de tipo lógico. Se trata de un objeto muy simple, ya que no dispone de propiedades ni métodos exceptuando aquéllas que hereda del objeto Object. El constructor del objeto es:

```
new Boolean(valor)
```

Si el parámetro se omite, o tiene los valores 0, null o false, el objeto toma el valor inicial como false. En cualquier otro caso, toma el valor true. Estas características hacen que este objeto se pueda usar para convertir un valor no booleano a booleano.

Tabla 7. Propiedades del objeto Boolean

Nombre	Descripción
constructor	Referencia a la función que se ha llamado para crear el objeto.
prototype	Representa el prototipo para la clase.

Tabla 8. Métodos del objeto Boolean

Nombre	Descripción	Sintaxis	Retorno
toString	Representa un objeto mediante un string.	toString()	"true" o "false" según el valor del objeto.
valueOf	Obtener el valor que tiene el objeto.	valueOf()	string "true" o "false" según su valor

2.2.3. El objeto Date

El objeto Date proporciona una extensa variedad de métodos que permiten manipular fechas y horas. Pero el objeto no contiene un reloj en funcionamiento, sino un valor de fecha estático que además internamente se almacena como el número de milisegundos desde la medianoche del 1 de enero de 1970.

El constructor del objeto puede recibir un abanico interesante de parámetros. A continuación, se presentan los más utilizados:

```
new Date()
new Date(año_num, mes_num, dia_num)
new Date(año_num, mes_num, dia_num, hora_num, min_num, seg_num)
```

en los que el significado de los parámetros es el siguiente:

año_num, mes_num, dia_num, hora_num, min_num, seg_num son enteros que forman parte de la fecha. En este caso, se debe tener en cuenta que el mes 0 corresponde a enero y el mes 11, a diciembre.

Trabajando con fechas

En el siguiente ejemplo, se puede ver el uso de las funciones que permiten mostrar la fecha actual en la página del navegador:

```
meses = new Array("Enero", "Febrero", "Marzo","Abril", "Mayo", "Junio", "Julio",
"Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre");
var fecha = new Date();
var mes = fecha.getMonth();
var anyo = fecha.getFullYear();
document.write("Hoy es" + fecha.getDate() + "de" + meses[mes] + "de" + anyo);
```

Tal como se observará a continuación, los objetos Date disponen de un conjunto muy amplio de métodos que permiten establecer o leer una propiedad directamente del objeto realizando internamente las conversiones necesarias, ya que, como se ha indicado antes, el objeto almacena sus valores en milisegundos.

Tabla 9. Propiedades del objeto Date

Nombre	Descripción
constructor	Referencia a la función que se ha llamado para crear el objeto.
prototype	Representa el prototipo para la clase. Es de sólo lectura.

Tabla 10. Métodos del objeto Date

Nombre	Descripción	Sintaxis	Parámetros	Retorno
getDate	Retorna el día del mes para una fecha.	getDate()		Entero entre 1 y 31.
getDay	Retorna el día de la semana para una fecha.	getDay()		Entero entre 0 y 6. El 0 es el domingo, el 1 es el lunes...
getHours	Retorna la hora para una fecha.	getHours()		Entero entre 0 y 23.
getMinutes	Retorna los minutos para una fecha.	getMinutes()		Entero entre 0 y 59.
getMonth	Retorna el mes para una fecha.	getMonth()		Entero entre 0 y 11.
getSeconds	Retorna los segundos para una fecha.	getSeconds()		Entero entre 0 y 59.
getTime	Retorna un número correspondiente al tiempo transcurrido para una fecha.	getTime()		Número de milisegundos transcurridos desde el 1 de enero de 1970 a las 00:00:00 horas.
getTimezoneOffset	Retorna la diferencia horaria, entre la hora local y la hora GMT (Greenwich Mean Time).	getTimezoneOffset()		Número de minutos que marca la diferencia horaria.
getFullYear	Retorna el año para una fecha.	getFullYear()		Retorna los cuatro dígitos.
parse	String que contiene los milisegundos transcurridos para la fecha.	Date.parse(fechaString)	fechaString: fecha en formato string.	Milisegundos transcurridos desde el 1 de enero de 1970 a las 00:00:00 horas.
setDate	Pone el día a una fecha.	setDate(valordia)	valordia: entero entre 1 y 31 que representa el día.	
setDay	Pone el día a una fecha.	setDate(valordia)	valordia: entero entre 1 y 31 que representa el día.	
setHours	Pone la hora a una fecha.	setHours(valorhora)	valorhora: entero entre 0 y 23 que representa la hora.	
setMinutes	Pone los minutos a una fecha.	setMinutes (valorminutos)	valorminutos: entero entre 0 y 59 que representa los minutos.	
setMonth	Pone el mes a una fecha.	setMonth(valormes)	valormes: entero entre 0 y 11 que representa el mes.	
setSeconds	Pone los segundos a una fecha.	setSeconds (valorsegundos)	valorsegundos: entero entre 0 y 59 que representa los segundos.	
setTime	Pone el valor a una fecha.	setTime(valorhorario)	valorhorario: milisegundos transcurridos desde el 1 de enero de 1970 a las 00:00:00 horas.	
setYear	Pone el año a una fecha.	setYear(valoranio)	valoranio: entero que representa el año.	

Nombre	Descripción	Sintaxis	Parámetros	Retorno
toGMTString	Convierte una fecha a string, usando las convenciones GMT de Internet.	toGMTString()		
toLocaleString	Convierte una fecha a string, usando las convenciones locales. Ver retorno.	toLocaleString()		Ejemplo: sábado, 07 de diciembre de 2002 21:22:59
toLocaleDateString	Convierte una fecha a string, usando las convenciones locales. Ver retorno.	toLocaleDateString()		Ejemplo: Sat Dec 7 21:22:59 UTC +0100 2002
toLocaleTimeString	Convierte una fecha a string, usando las convenciones locales.			
UTC	Milisegundos transcurridos para la fecha indicada.	Date.UTC(año, mes, día, horas, mins, segs)	Enteros. Los tres últimos son opcionales.	Milisegundos transcurridos desde el 1 de enero de 1970 a las 00:00:00 horas.

2.3. Los objetos Math, Number y String

2.3.1. El objeto Math

El objeto Math está formado por un conjunto de constantes y métodos que permiten realizar operaciones matemáticas de cierta complejidad. Este objeto es estático, por lo que no se puede crear una instancia y el acceso a sus constantes y métodos se realiza directamente utilizando el propio objeto.

El siguiente ejemplo muestra un conjunto de cálculos realizados con el objeto Math:

```
function calcula(numero){
   cosValue = Math.cos(numero); //Almacena en cosValue el valor del coseno
   sinValue = Math.sin(numero); //Almacena en sinValue el valor del seno
   tanValue = Math.tan(numero); //Almacena en cosValue el valor del tangente
   sqrtValue = Math.sqrt(numero); //Almacena en sqrtValue el valor de la raíz
   powValue = Math.pow(numero,3); //Almacena en powValue el valor de elevar a 3 el valor
   expValue = Math.exp(numero); //Almacena en expValue el valor de e elevado al valor del número
}
```

Tabla 11. Propiedades del objeto Math

Nombre	Descripción	Ejemplo

E	Constante de Euler. Su valor aproximado es 2,718. Es de sólo lectura.	function valorE() {
		return **Math.E**
		}

Nombre	Descripción	Ejemplo

LN10	Logaritmo neperiano de 10. Su valor aproximado es 2,302. Es de sólo lectura.	function valorLN10() {
		return **Math.LN10**
		}

LN2	Logaritmo neperiano de 2. Su valor aproximado es 0,693. Es de sólo lectura.	function valorLN2() {
		return **Math.LN2**
		}

LOG10E	Logaritmo en base 10 del número E. Su valor aproximado es 0,434. Es de sólo lectura.	function valorLog10e() {
		return **Math.LOG10E**
		}

LOG2E	Logaritmo en base 2 del número E. Su valor aproximado es 1,442. Es de sólo lectura.	function valorLog2e() {
		return **Math.LOG2E**
		}

PI	Número PI. Su valor aproximado es 3,1415. Es de sólo lectura.	function valorPi() {
		return **Math.PI**
		}

SQRT1_2	1 dividido por la raíz cuadrada de 2. Su valor aproximado es 0,707. Es de sólo lectura.	function valorSQRT1_2() {
		return **Math.SQRT1_2**
		}

SQRT2	Raíz cuadrada de 2. Su valor aproximado es 1,414. Es de sólo lectura.	function valorSQRT2() {
		return **Math.SQRT2**
		}

Tabla 12. Métodos del objeto Math

Nombre	Descripción	Sintaxis	Parámetros	Retorno
abs	Calcula el valor absoluto de un número.	abs(x)	x: un número.	Valor absoluto de x.
acos	Calcula el arco coseno de un número.	acos(x)	x: un número.	Valor del arcocoseno de x en radianes. Valor entre [0, PI].
asin	Calcula el arco seno de un número.	asin(x)	x: un número.	Valor del arcoseno de x en radianes. Valor entre [–PI/2,PI/2].
atan	Calcula el arco tangente de un número.	atan(x)	x: un número.	Valor del arco tangente de x en radianes. Valor entre [–PI/2,PI/2].
atan2	Calcula el arco tangente del cociente de dos números.	atan2(y,x)	y: un número (coordenada y). x: un número (coordenada x).	
ceil	Retorna el menor entero, mayor o igual a un número.	ceil(x)	x: un número.	El primer entero mayor o igual a x (para 9,8 retornaría 10).
cos	Calcula el coseno de un número.	cos(x)	x: un número.	Valor del coseno de x. Valor entre [–1,1].
exp	Calcula el valor del número E elevado a un número.	exp(x)	x: un número.	Valor de E^x.
floor	Retorna el mayor entero, menor o igual a un número.	floor(x)	x: un número.	El primer entero menor o igual a x (para 9,8 retornaría 9).

Nombre	Descripción	Sintaxis	Parámetros	Retorno
log	Calcula el logaritmo en base E de un número.	log(x)	x: un número.	El logaritmo en base E de x.
max	Retorna el mayor de dos números.	max(x,y)	x: un número. y: un número.	Si x > y, retorna x. Si y > x, retorna y.
min	Retorna el menor de dos números.	min(x,y)	x: un número. y: un número.	Si x > y, retorna y. Si y > x, retorna x.
pow	Calcula la potencia entre dos números.	pow(x,y)	x: un número que representa la base. y: un número que representa el exponente.	Valor de x^y.
random	Retorna un número aleatorio entre 0 y 1.	random()		Número aleatorio entre [0,1].
round	Redondea un número al entero más cercano.	round(x)	x: un número.	El valor redondeado de x.
sin	Calcula el seno de un número.	sin(x)	x: un número.	Valor del seno de x. Valor entre [-1,1].
sqrt	Calcula la raíz cuadrada de un número.	sqrt(x)	x: un número.	Raíz cuadrada de x.
tan	Calcula la tangente de un número.	tan(x)	x: un número.	Valor de la tangente de x.

2.3.2. El objeto Number

Este objeto es utilizado principalmente para acceder a métodos de formateo de números, además de proporcionar propiedades estáticas que definen constantes numéricas. El constructor del objeto Number es el siguiente:

```
new Number (valor)
```

donde "valor" es el valor numérico que estará contenido en el objeto creado. Si el parámetro es omitido, la instancia se inicializará con el valor 0.

Tabla 13. Propiedades del objeto Number

Nombre	Descripción	Ejemplo de uso
Constructor	Referencia a la función que se ha llamado para crear el objeto.	

MAX_VALUE	Es el valor numérico más grande que se puede representar en JavaScript. Es de sólo lectura.	if (numero <= Number.MAX_VALUE)
		miFuncion(numero)
		else
		alert("número demasiado grande")

MIN_VALUE	Es el valor numérico más pequeño que se puede representar en JavaScript. Es de sólo lectura.	if (numero >= Number.MIN_VALUE)
		miFuncion(numero)
		else
		miFuncion(0)

NaN	Indica que no se trata de un valor numérico (Not-A-Number). Es de sólo lectura.	
NEGATIVE_INFINITY	Representa el valor negativo infinito. Es de sólo lectura.	

Nombre	Descripción	Ejemplo de uso
POSITIVE_INFINITY	Representa el valor positivo infinito. Es de sólo lectura.	
Prototype	Representa el prototipo para la clase.	

Valores del objeto Number

- El número más grande que se puede representar en JavaScript es el 1.79E+308. Los valores mayores a este adquieren el valor "Infinity".
- El número más pequeño que se puede representar en JavaScript es el 5E-324. La propiedad MIN_VALUE no representa el menor número negativo, sino el positivo, más próximo al 0, que puede representar el lenguaje. Los valores menores a éste se convierten en 0.
- Las propiedades MAX_VALUE, MIN_VALUE, POSITIVE_INFINITY y NEGATIVE_INFINITY son estáticas. Por lo tanto, es necesario utilizarlas con el propio objeto, por ejemplo: Number.MAX_VALUE.
- Para ver el valor de la propiedad NaN, se debe usar la función isNaN.
- Un valor menor a NEGATIVE_INFINITY se visualiza como "–Infinity"
- Un valor mayor a POSITIVE_INFINITY se visualiza como "Infinity".

Tabla 14. Métodos del objeto Number

Nombre	Descripción	Sintaxis	Parámetros	Retorno
toExponential	Expresa un número en notación exponencial.	toExponential(n)	n: número de decimales.	String
toFixed	Para redondear a partir del número de decimales indicado.	toFixed(n)	n: número de decimales.	String
toString	String que representa el objeto especificado.	toString([base])	base: número entre 2 y 16 que indica la base en la que se representa el número.	String
toPrecision	El dato expresado según la precisión indicada.	toPrecision(n)	n: número de dígitos de precisión.	String
valueOf	Obtener el valor que tiene el objeto.	valueOf()		String

2.3.3. El objeto String

El objeto String es el contenedor de tipos primitivos de tipo cadena de caracteres. Proporciona un conjunto muy amplio de métodos que permiten la manipulación, extracción y conversión de cadenas a texto HTML.

La sintaxis del constructor del objeto String es:

```
new String(texto)
```

donde "texto" es una cadena de caracteres opcional en el constructor.

Los métodos del objeto se pueden invocar en cadenas primitivas, es decir, sin haber creado el objeto con el constructor. Esta característica hace que la creación de cadenas con la sintaxis anterior sea poco común:

```
var cadena = "Hola Mundo";
longitud = cadena.length;
```

La propiedad "length" del objeto String, a diferencia del objeto Array, no puede ser establecida por el programador; se trata de una propiedad de sólo lectura y cambia cuando la cadena modifica su tamaño en caracteres.

En general, todos los métodos del objeto String no modifican el contenido del objeto, sino que devuelven el resultado de la acción, aunque sin modificar el contenido inicial de la cadena; para modificar una cadena se le debe asignar un nuevo valor:

```
var cadena = "Hola Mundo";
cadena.toUpperCase();
```

La variable *cadena* sigue almacenando el valor "Hola Mundo"; la función ha actuado y ha devuelto la cadena cambiando los caracteres a mayúsculas, pero no ha modificado la variable que contiene la cadena original:

```
var cadena = "Hola Mundo";
cadena = cadena.toUpperCase();
```

En este caso, se ha modificado la variable cadena y ahora tiene el contenido "HOLA MUNDO".

Trabajando con cadenas

El método charAt(entero) devuelve el carácter que se encuentra en la posición indicada por el número pasado como parámetro; se debe tener en cuenta que igual que en los arrays, el primer carácter de una cadena tiene el índice 0:

```
var cadena = "Hola Mundo";
caracter = cadena.charAt(2); //La variable caracter recupera el valor "l"
```

El método indexOf(cadena) devuelve el índice de la primera aparición del argumento en la cadena. Siguiendo con el ejemplo anterior:

```
var cadena = "Hola Mundo";
posicion = cadena.indexOf("Mun"); //La variable posicion recupera el valor 6.
```

En el caso de que el argumento no se encuentre en la cadena, el método devuelve el valor –1. El método acepta un segundo parámetro opcional que especifica el índice desde el que se debe comenzar la búsqueda:

```
var cadena = "Hola Mundo";
posicion = cadena.indexOf("o",3); //La variable posicion recupera el valor 7.
```

El método lastIndexOf(cadena) devuelve el índice de la última aparición de la cadena pasada como argumento. Al igual que el anterior, dispone de un argumento opcional que indica el índice en el que debe terminar la búsqueda:

```
var cadena = "Hola Mundo";
posicion = cadena.lastIndexOf("o",3); //La variable posicion recupera el valor 1.
```

Igual que el anterior, devuelve el valor –1 cuando no encuentra la cadena buscada.

El método substring(inicio,fin) realiza una extracción de la cadena, de manera que el primer argumento especifica el índice en el que empieza la cadena que se desea extraer y el segundo argumento (opcional) señala el final de la subcadena.

En el caso de que no se pase el segundo argumento, se extrae la subcadena hasta el final de la cadena original:

```
var cadena = "Hola Mundo";
var subCadena = cadena.substring(5); //subCadena adquiere el valor "Mundo"
var subCadena2 = cadena.substring(5,7); //subCadena2 adquiere el valor "Mu"
```

El método concat() realiza la concatenación de todas las cadenas que se le pasan como parámetros (acepta cualquier cantidad de argumentos) y devuelve la cadena resultado de concatenar la cadena original con aquéllas que se le han pasado como parámetros:

```
var cadena = "Hola Mundo";
var cadena2 = cadena.concat(" libre", " y feliz"); //cadena 2 adquiere el
valor "Hola Mundo libre y feliz"
```

Pero la concatenación es más común realizarla mediante el operador "+":

```
var cadena2 = cadena + " libre" + " y feliz";
```

El método split() divide la cadena en cadenas separadas según un delimitador pasado como argumento del método; el resultado es almacenado en un array:

```
var vector = "Hola Mundo libre".split(" ");
```

El ejemplo anterior asigna a la variable vector un array con tres elementos: "Hola", "Mundo", "libre".

Tabla 15. Propiedades del objeto String

Nombre	Descripción	Ejemplo
constructor	Referencia a la función que se ha llamado para crear el objeto.	
length	Longitud del string.	var tema = "Programación" alert("La longitud de la palabra Programación es " + tema.length)

Tabla 16. Métodos del objeto String

Nombre	Descripción	Sintaxis	Parámetros	Retorno
anchor	Crea un ancla HTML.	anchor(nombre_ancla)	nombre_ancla: texto para el atributo NAME de la etiqueta <A NAME=.	
big	Muestra un texto en letras grandes.	big()		
blink	Muestra el texto parpadeando.	blink()		
bold	Muestra el texto en negrita.	bold()		
charAt	Retorna un carácter del texto, el que está en la posición indicada.	charAt(indice)	indice: número entre 0 y la longitud-1 del texto.	Carácter del texto que está en la posición indicada por el parámetro.
charCodeAt	Retorna el código ISO-Latin-1 del carácter que está en la posición indicada.	charCodeAt(indice)	indice: número entre 0 y la longitud-1 del texto.	Código del carácter que está en la posición indicada por el parámetro.
concat	Une dos cadenas de texto.	concat(texto2)	texto2: cadena de texto a unir a la que llama al método.	String resultado de la unión de los otros dos.
fixed	Muestra el tipo con fuente de letra teletipo.	fixed()		
fontcolor	Muestra el texto en el color especificado.	fontcolor(color)	color: color para el texto.	
fontsize	Muestra el texto en el tamaño especificado.	fontsize(size)	size: tamaño para el texto.	
fromCharCode	Retorna el texto creado a partir de caracteres en código ISO-Latin-1.	fromCharCode(num1..., numN)	numN: códigos ISO-Latin-1	String
indexOf	Retorna el índice en el que se encuentra por primera vez la secuencia de caracteres especificada.	indexOf(valor, inicio)	valor: carácter o caracteres que buscar. inicio: índice a partir del cual empieza a buscar.	Número que indica el índice Retorna –1 si no se encuentra el valor especificado.
italics	Muestra el texto en cursiva.	italics()		

Nombre	Descripción	Sintaxis	Parámetros	Retorno
lastIndexOf	Retorna el índice en el que se encuentra por última vez la secuencia de caracteres especificada.	lastIndexOf(valor, inicio)	valor: carácter o caracteres que buscar. inicio: índice a partir del cual empieza a buscar.	Número que indica el índice Retorna –1 si no se encuentra el valor especificado.
link	Crea un link HTML.	link(href)	href: string que especifica el destino del link.	
match	Retorna las partes del texto que coinciden con la expresión regular indicada.	match(exp)	exp: expresión. Puede incluir los flags /f (global) y /i (ignorar mayúsculas y minúsculas)	Array que contiene los textos encontrados.
replace	Substituye una parte del texto por el nuevo texto indicado.	replace(exp, texto2)	exp: expresión regular para la búsqueda del texto que sustituir. texto2: texto que sustituirá al encontrado.	El nuevo String.
search	Retorna el índice del texto indicado en la expresión regular.	search(exp)	exp: expresión para la búsqueda.	
slice	Extrae una porción del String.	slice(inicio,[fin])	inicio: índice del primer carácter que extraer. fin: índice del último carácter que extraer. Puede ser negativo, entonces indica cuántos hay que restar desde el final. Si no se indica, extrae hasta el final.	String que contiene los caracteres que estaban entre *inicio* y *fin*.
small	Muestra el texto en fuente pequeña.	small()		
split	Crea un array, separando el texto según el separador indicado.	split([separador], [limite])	separador: carácter que indica por dónde separar. Si se omite, el array contendrá sólo un elemento, que será el string completo. limite: Indica el máximo número de partes que poner en el array.	Array
strike	Muestra el texto tachado.	strike()		
sub	Muestra el texto como subíndice.	sub()		
substr	Retorna una porción del texto.	substr(inicio, [longitud])	inicio: índice del primer carácter que extraer. longitud: número de caracteres que extraer. Si se omite, se extrae hasta el final del String.	String
substring	Retorna una porción del texto.	substring(inicio, fin)	inicio: índice del primer carácter que extraer. fin: índice+1 del último carácter quw extraer.	
sup	Muestra el texto como superíndice.	sup()		

Nombre	Descripción	Sintaxis	Parámetros	Retorno
toLocaleLowerCase	Convierte el texto a minúsculas, teniendo en cuenta el lenguaje del usuario.	toLocaleLowerCase()		Nuevo String en minúsculas.
toLocaleUpperCase	Convierte el texto a mayúsculas, teniendo en cuenta el lenguaje del usuario.	toLocaleUpperCase()		Nuevo String en mayúsculas.
toLowerCase	Convierte el texto a minúsculas.	toLowerCase()		Nuevo String en minúsculas.
toUpperCase	Convierte el texto a mayúsculas.	toUpperCase()		Nuevo String en mayúsculas.
toString	Obtiene el String que representa el objeto.	toString()		String
valueOf	Obtiene el String que representa el valor del objeto.	valueOf()		String

Códigos ISO-Latin-1

El conjunto de códigos ISO-Latin-1 toma valores entre 0 y 255. Los 128 primeros se corresponden con el código ASCII.

3. Expresiones regulares y uso de *cookies*

3.1. Las expresiones regulares

3.1.1. Introducción a la sintaxis

Las expresiones regulares son un mecanismo que permite realizar búsquedas, comparaciones y ciertos reemplazamientos complejos.

Por ejemplo, si se escribe en la línea de comandos de Microsoft Windows el comando:

```
dir *.exe,
```

Se está utilizando una expresión regular que define todas las cadenas de caracteres que empiecen con cualquier valor seguida de ".exe", es decir, todos los archivos ejecutables independientemente de su nombre.

La acción anterior en la que se compara la cadena de texto con el patrón (expresión regular) se denomina reconocimiento de patrones (*pattern matching*).

Las expresiones regulares ayudan a la búsqueda, comparación y manipulación de cadenas de texto.

En Javascript, las expresiones regulares se basan en las del lenguaje Perl, de manera que son muy parecidas a las de este y se representan por el objeto RegExp (de *regular expresion*).

Se puede crear una expresión regular con el constructor del objeto RegExp o bien utilizando una sintaxis especialmente creada para ello.

En el siguiente ejemplo, se observa esto último:

```
var patron = /Cubo/;
```

La expresión regular anterior es muy simple: en una comparación con una cadena devolvería *true* en el caso de que la cadena con la que se compara fuera "Cubo" (todas las expresiones regulares se escriben entre barras invertidas).

De la misma manera, es posible crear la anterior expresión regular utilizando el objeto RegExp:

```
var patron = new RegExp("Cubo");
```

Pero en este caso lo que se le pasa al constructor es una cadena, por lo tanto en lugar de usar / se encierra entre comillas dobles.

Las expresiones regulares se pueden crear utilizando su sintaxis específica o con el objeto RegExp.

Para complicar un poco más el ejemplo anterior, se supone que se quiere comprobar si la cadena es “Cubo” o “Cuba”. Para ello se usan los corchetes, que indican opción, es decir, al comparar con /[ao]/ devolvería cierto en caso de que la cadena fuera la letra “a” o la letra “o”.

```
var patron = /Cub[ao]/;
```

¿Y si se quisiera comprobar si la cadena es Cub0, Cub1, Cub2, …, Cub9? En lugar de tener que encerrar los 10 dígitos dentro de los corchetes se utiliza el guión, que sirve para indicar rangos de valores. Por ejemplo, 0-9 indica todos los números de 0 a 9 inclusive.

```
var patron = /Cub[0-9]/;
```

Si además se busca que el último carácter sea un dígito (0-9) o una letra minúscula (a-z), se soluciona fácilmente escribiendo dentro de los corchetes un criterio detrás del otro.

```
var patron = /Cub[0-9a-z]/;
```

¿Y qué ocurriría si en lugar de tener solo un número o una letra minúscula se quiere comprobar que puedan haber varias, pero siempre minúsculas o números?

En el caso anterior, se recurrirá a los siguientes marcadores:

- +: indica que lo que tiene a su izquierda puede estar 1 o más veces.
- *: indica que puede estar 0 o más veces (en el caso de + el número o la minúscula tendría que aparecer al menos una vez, con * Cub también se aceptaría).

- ?: indica opcionalidad, es decir, lo que se tiene a la izquierda puede o no aparecer (puede aparecer 0 o 1 veces).

- {}: sirve para indicar exactamente el número de veces que puede aparecer o un rango de valores. Por ejemplo, {3} indica que tiene que aparecer exactamente 3 veces, {3,8} indica que tiene que aparecer entre 3 y 8 veces y {3,} indica que tiene que aparecer al menos 3 veces.

Se tiene que tener cuidado con las {} ya que exigen que se repita lo último, de manera que cuando no se esté seguro de lo que va a hacer se utilizarán los (). Para ilustrar lo anterior, se plantea el siguiente ejemplo de uso de expresiones regulares:

```
var patron = /Cub[ao]{2}/;
document.write("Cubocuba".search(patron));
document.write("Cuboa".search(patron));
```

La función *search* de String comprueba si la cadena representada por el patrón que se le pasa como argumento se encuentra dentro de la cadena sobre la que se llama. En el caso de que así sea, devuelve la posición (por ejemplo, para la cadena Cubo con el patrón /C/ devolvería 0, 1 si el patrón es u, 2 si es b, etc.) y -1 si no se encuentra.

Otro uso interesante es el uso del método *replace* del objeto String, cuya sintaxis cadena.replace(patron, sustituto) indica que se sustituye la cadena sobre la que se llama las ocurrencias del patrón por la cadena especificada en la llamada.

Uno de los elementos más interesantes de las expresiones regulares es la especificación de la posición en la que se tiene que encontrar la cadena. Para ello se utilizan los caracteres "^" y "$", que indican que el elemento sobre el que actúa debe ir al principio de la cadena o al final de esta respectivamente.

En el siguiente ejemplo, se puede observar su uso:

```
var patron = /^aa/; //Se busca la cadena aa al inicio de la cadena
var patron = /uu$/; //Se busca la cadena uu al final de la cadena
```

Otras expresiones interesantes son las siguientes:

- \d: un dígito, equivale a [0-9].
- \D: cualquier carácter que no sea un dígito.
- \w: cualquier carácter alfanumérico, equivale a [a-zA-Z0-9_].
- \W: cualquier carácter no alfanumérico.
- \s: espacio.
- \t: tabulador.

El siguiente enlace apunta a un portal web dedicado a las expresiones regulares, en el que se tiene que tener en cuenta que estas son compatibles en la mayoría de los lenguajes de programación que las implementan:

```
http://www.regular-expressions.info/
```

3.1.2. El objeto RegExp

El objeto RegExp contiene el patrón de una expresión. El constructor para el objeto RegExp es el siguiente:

```
new RegExp("patron", "indicador")
```

donde:

- patron: texto de la expresión regular.
- Indicador: es opcional y puede tomar tres valores:
  - g: se tendrán en cuenta todas las veces que la expresión aparece en la cadena.
  - i: ignora mayúsculas y minúsculas.
  - gi: tienen efecto las dos opciones, g e i.

En el patrón de la expresión regular, se pueden usar caracteres especiales. Los caracteres especiales sustituyen una parte del texto. A continuación, se listan los caracteres especiales que se pueden usar:

Tabla 17

Caracteres especiales en expresiones regulares	
\	Para los caracteres que normalmente se interpretan como literales, indica que el carácter que le sigue no se debe interpretar como un literal. Por ejemplo: /b/ se interpretaría como “b”, pero /\b/ se interpretaría como indicador de límite de palabra. Para los caracteres que normalmente no se interpretan como literales, indica que en este caso sí debe interpretarse como un literal y no como un carácter especial. Por ejemplo, el * es un carácter especial que se utiliza como comodín, /a*/ puede significar ninguna o varias a. Si la expresión contiene /a*/, se interpreta como el literal “a*”.
^	Indica inicio de línea. Por ejemplo, /^A/ no encontrará la A en la cadena “HOLA”, pero sí la encontrará en “ALARMA”.
$	Indica final de línea. Por ejemplo, /$A/ no encontrará la A en la cadena “ADIÓS”, pero sí la encontrará en “HOLA”.
*	Indica que el carácter que le precede puede aparecer ninguna o varias veces. Por ejemplo, /ho*/ se encontrará en “hola”, “hoooola”, y también en “hiedra”, pero no en “camello”.
+	Indica que el carácter que le precede puede aparecer una o varias veces. Por ejemplo, /ho+/ se encontrará en “hola” y “hoooola”, pero no en “hiedra” ni en “camello”.
?	Indica que el carácter que le precede puede aparecer ninguna o una vez. Por ejemplo, /ho?/ se encontrará en “hola” y “hiedra”, pero no en “hoooola” ni en “camello”.
.	Indica un único carácter a excepción del salto de línea. Por ejemplo, /.o/ se encontrará en “hola” pero no en “camello”.
(x)	Indica que además de buscar el valor x, se repetirá la búsqueda entre el resultado de la primera búsqueda. Por ejemplo, en la frase “hola, te espero en el hotel de holanda”, /(holan*)/ encontraría “hola”, “holan”, y “hola” (el último “hola” es parte de “holan”).

Caracteres especiales en expresiones regulares	
x|y	Indica el valor de x o el de y. Por ejemplo, /sol|viento/ encontraría "sol" en la frase: "Hace sol en Sevilla".
{n}	Indica cuántas veces exactas debe aparecer el valor que le precede (n es un entero positivo). Por ejemplo, /o{4}/ se encontraría en "hoooola" pero no en "hola".
{n,}	Indica cuántas veces como mínimo debe aparecer el carácter que le precede (n es un entero positivo). Por ejemplo, /o{2,}/ se encontraría en "hoooola" y "hoola" pero no en "hola".
{n,m}	Indica el número mínimo y máximo de veces que puede aparecer el carácter que le precede (n y m son enteros positivos). Por ejemplo, /o(1,2)/ se encontraría en "hola" y "hoola", pero no en "hoooola".
[xyz]	Indica cualquiera de los valores entre corchetes. Los elementos contenidos expresan un rango de valores, por ejemplo [abcd] podría expresarse también como [a-d].
[^xyz]	Busca cualquier valor que no aparezca entre corchetes. Los elementos contenidos expresan un rango de valores, por ejemplo [^abcd] podría expresarse también como [^a-d].
[\b]	Indica un *backspace*.
\b	Indica un delimitador de palabra, como un espacio. Por ejemplo, /\bn/ se encuentra en "nutria" pero no en "mono", y /n\b/, en "camaleón" pero no en "mono".
\B	Indica que no puede haber delimitador de palabra, como un espacio. Por ejemplo, /\Bn/ se encuentra en "mono" pero no en "nutria" ni "camaleón".
\cX	Indica un carácter de control (X es el carácter de control). Por ejemplo, /cM/ indica Ctrl+M
\d	Indica que el carácter es un dígito. También puede expresarse como /[0-9]/.Por ejemplo, /\d/ en "calle pez, n.º 9" encontraría 9.
\D	Indica que el carácter no es un dígito. /\D/ también puede expresarse como /[^0-9]/
\f	Indica salto de página (form-feed).
\n	Indica salto de línea (linefeed).
\r	Indica retorno de carro.
\s	Indica un espacio en blanco que puede ser el espacio, el tabulador, el salto de página y el salto de línea. Por tanto, es equivalente a poner [\f\n\r\t\v]
\S	Indica un único carácter diferente al espacio, al tabulador, al salto de página y al salto de línea. Por tanto, es equivalente a poner [^\f\n\r\t\v]
\t	Indica el tabulador.
\v	Indica un tabulador vertical.
\w	Indica cualquier carácter alfanumérico incluyendo el _ Es equivalente a poner [A-Za-z0-9_]
\n	*n* es un valor que hace referencia al paréntesis anterior (cuenta los paréntesis abiertos). Por ejemplo, en "manzana, naranja, pera, melocotón", la expresión /manzana(,)\snaranja\1/ encontraría "manzana, naranja".
\ooctal \xhex	Permite incluir códigos ASCII en expresiones regulares. Para valores octales y hexadecimales. o y x tomarían los valores, por ejemplo, \2Fhex.

Tabla 18

Propiedades del objeto RegExp	
Nombre	**Descripción**
$1, ..., $9	Contienen las partes de la expresión contenidas entre paréntesis. Es de solo lectura.

Propiedades del objeto RegExp	
Nombre	**Descripción**
$_	Ver la propiedad *input*
$*	Ver la propiedad *multiline*
$&	Ver la propiedad *lastMatch*
$+	Ver la propiedad *lastParen*
$`	Ver la propiedad *leftContext*
$'	Ver la propiedad *rightContext*
global	Indica si se usa el indicador "g" en la expresión regular. Puedo tener los valores *true* (si se usa el indicador "g") y *false* (en caso contrario). Es de solo lectura.
ignoreCase	Indica si se usa el indicador "i" en la expresión regular. Puedo tener los valores *true* (si se usa el indicador "i") y *false* (en caso contrario). Es de solo lectura.
input	Representa el string sobre el que se aplica la expresión regular. También se llama $_. Es estática, por lo que se usa de la forma: RegExp.input.
lastIndex	Especifica el índice a partir del cual hay que aplicar la expresión regular.
lastMatch	Representa el último ítem encontrado. También se llama $&. Es estática, por lo que se usa de la forma: RegExp. lastMatch. Es de solo lectura.
lastParen	Representa el último ítem encontrado por una expresión de paréntesis. También se llama $+. Es estática, por lo que se usa de la forma: RegExp. lastParen. Es de solo lectura.
leftContext	Representa el substring que precede al último ítem encontrado. También se llama $`. Es estática, por lo que se usa de la forma: RegExp. leftContext. Es de solo lectura.
multicine	Indica si se aplicará la búsqueda en varias líneas. Sus posibles valores son *true* y *false*. También se llama $*. Es estática, por lo que se usa de la forma: RegExp. multiline.
prototype	Representa el prototipo para la clase.
rightContext	Representa el substring que sigue al último ítem encontrado. También se llama $'. Es estática, por lo que se usa de la forma: RegExp. rightContext. Es de solo lectura.
source	Representa el string que contiene el patrón para la expresión regular, y excluye las \ y los indicadores "i" y "g". Es de solo lectura.

Tabla 19

Métodos del objeto RegExp				
Nombre	**Descripción**	**Sintaxis**	**Parámetros**	**Retorno**
compile	Compila la expresión regular durante la ejecución de un script.	regexp.compile(patron, [indicadores])	patron: texto de la expresión regular. Indicadores: "g" y/o "i".	
Exec	Ejecuta la búsqueda.	regexp.exec(str) regexp(str)	str: string sobre el que se aplica la búsqueda.	Un *array* con los ítems encontra-dos.

Métodos del objeto RegExp				
Nombre	**Descripción**	**Sintaxis**	**Parámetros**	**Retorno**
test	Ejecuta la búsqueda.	regexp.test(str)	str: string sobre el que se aplica la búsqueda.	*true* si encuentra algún ítem, *false* en caso contrario.

3.2. Las *cookies*

Las *cookies* nacen con el objetivo de solucionar una limitación del protocolo HTTP 1.0. Esta viene dada por el hecho de que HTTP es un protocolo sin estado. Esto provoca que no exista forma de mantener comunicación o información del usuario a lo largo de las distintas peticiones que se realizan al servidor en una misma conexión o visita a una espacio o página web.

Las *cookies* aportan un mecanismo que permite almacenar en el equipo del cliente un conjunto pequeño de datos de tipo texto que son establecidos por el servidor web. De esta manera, en cada conexión el cliente devuelve la *cookie* con el valor almacenado al servidor que procesa el valor y actúa de forma que realiza las acciones pertinentes.

Las *cookies* aportan un mecanismo que permite almacenar un conjunto pequeño de datos en el equipo cliente.

La asignación de una *cookie* sigue la siguiente sintaxis:

```
nombre=valor [;expires=fechaGMT] [;domain=dominio] [;path=ruta] [;secure]
```

donde:

- nombre=valor: define el nombre de la *cookie* y el valor que va a almacenar.

- expires=fechaGMT: establece la fecha de caducidad de la *cookie*. Esta fecha debe establecerse en formato GMT, por lo que será útil el método toGMTString() del objeto Date. Este parámetro es opcional, de manera que cuando una *cookie* no tiene establecida una fecha de caducidad esta se destruye cuando el usuario cierra el navegador, en este caso se dice que la *cookie* es de sesión. Las *cookies* que tienen establecida una fecha de caducidad se conocen como *cookies* persistentes.

- domain=dominio: establece el dominio que ha asignado la *cookie*, de modo que esta solo se devolverá ante una petición de este, por ejemplo: domain=www.uoc.edu implica que la *cookie* solo se devuelve al servidor www.uoc.edu cuando se establece una conexión con este. Si no se estable-

ce ningún valor es el propio navegador quien establece el valor del dominio que ha creado la *cookie*.

- path=ruta: establece una ruta específica del dominio sobre la que se devolverá la *cookie*. Si no se establece, la ruta por defecto asignada por el navegador es la ruta desde la que se ha creado la *cookie*.

- secure: si se indica este valor, la *cookie* solo se devuelve cuando se ha establecido una comunicación segura mediante HTTPS. Si no se asigna este valor, el navegador devuelve la *cookie* en conexiones no seguras HTTP.

Por tanto, el funcionamiento básico es el siguiente: un usuario se conecta a un sitio web, el navegador a partir de la URL de este revisa su conjunto de *cookies* buscando una que coincida con el dominio y la ruta. Si existe una o más *cookies* con estas características, el navegador envía al servidor la *cookie* o, en caso de varias, las *cookies* separadas por el carácter punto y coma:

```
nombre=Victor; email=vriospazos@uoc.edu
```

Para que las *cookies* funcionen, es necesario tener habilitadas estas en el navegador. Algunos usuarios las consideran un mecanismo de invasión de la intimidad, ya que se pueden establecer *cookies* persistentes que lleven un seguimiento de ciertas acciones en el proceso de navegación por los usuarios.

Nota

Es posible comprobar si las *cookies* están habilitadas. Se puede encontrar más información en:

https://developer.mozilla.org/es/docs/Web/API/Navigator.cookieEnabled

y en

http://www.w3schools.com/jsref/prop_nav_cookieenabled.asp

3.2.1. Manejo de *cookies*

En JavaScript es posible trabajar con *cookies* a partir de la propiedad *cookie* del objeto Document. El funcionamiento es muy simple: solo es necesario asignar a esta propiedad una cadena que represente a la *cookie*, de manera que el navegador analice la cadena como *cookie* y la añada a su lista de *cookies*.

```
document.cookie="nombre=Victor; expires=Sun, 14-Dec-2010 08:00:00 GMT; path=/ficheros";
```

En la anterior asignación, se crea una *cookie* persistente con fecha de caducidad del 14/12/10; en este caso se asigna una ruta que se utilizará conjuntamente con el dominio por defecto de la página que ha creado la *cookie*.

Este mecanismo de dominio/ruta provoca que una *cookie* solo pueda ser recuperada por el dominio/ruta que se indica, impidiendo de este modo el acceso a la información almacenada desde otros dominios.

En JavaScript se trabaja con *cookies* a partir de la propiedad *cookie* del objeto Document.

El análisis que realiza el navegador sobre la cadena asignada a document.cookie comprueba que el nombre y el valor no contengan caracteres como espacios en blanco, comas, eñes, acentos y otros caracteres.

Para solucionar este problema, se utilizan las funciones escape() y unescape() que realizan la conversión de cadenas en cadenas URL que el validador del navegador da por buenas.

De esta forma se utilizará el método escape() para convertir una cadena en formato URL antes de almacenarla en la *cookie*, y cuando la *cookie* se recupere se utilizará la función unescape() sobre el valor de la *cookie*.

3.2.2. Escritura, lectura y eliminación de *cookies*

La escritura de *cookies* es muy sencilla, solo es necesario asignar a la propiedad *cookie* una cadena en la que se especifique el nombre, el valor y los atributos de caducidad, dominio, ruta y seguridad que se deseen aplicar.

En el siguiente ejemplo, se muestra una función que recibe como parámetros el nombre, el valor y el número de días durante los que la *cookie* estará activa como parámetro opcional, de manera que si este no se pasa, la *cookie* será de sesión y, por tanto, se eliminará cuando el usuario cierre el navegador.

```
function asignaCookie(nombre,valor,dias){
   if (typeof(dias) == "undefined"){
      //Si no se pasa el parámetro dias la cookie será de sesión
      document.cookie = nombre + "=" + valor;
   } else {
      //Se crea un objeto Date al que se le asigna la fecha actual
      //y se le a ade los dias de caducidad transformados en
      //milisegundos
      var caduca = new Date;
      caduca.setTime(fechaCaduca.getTime() + dias*24*3600000);
      document.cookie = nombre + "=" + valor +";expires=" + caduca.toGMTString();
   }
}
```

La lectura de *cookies* se realiza examinando la cadena almacenada en la propiedad *cookie* del objeto Document, pero se debe tener en cuenta que esta cadena está formada por tantos pares nombre=valor como *cookies* con distinto valor se han establecido en el documento actual. Por ejemplo, si se realiza la asignación:

```
document.cookie= "nombre=Victor";
document.cookie="email=vriospazos@uoc.edu"
```

Cada una de las anteriores cadenas se añade a la *cookie*, de modo que el valor de document.cookie será el siguiente:

```
"nombre=Victor; email=vriospazos@uoc.edu"
```

Normalmente solo es necesario recuperar o trabajar con alguna *cookie* en concreto, por ejemplo se necesita la cadena que indica el correo electrónico, pero la *cookie* contiene todas las cadenas almacenadas.

Una *cookie* almacena el conjunto de cadenas que se van asignando a la propiedad *cookie* del objeto Document.

Esto hace necesario implementar un mecanismo que recupere cada cadena por separado a partir de un análisis de la cadena devuelta por document.cookie.

El siguiente ejemplo utiliza un *array* asociativo para almacenar los nombres y los valores de cada uno de los componentes de la *cookie*:

```
//Se crea el objeto que contendrá el array asociativo cookies[nombre] = valor
var cookies = new Object();

//Se define la función que analiza la cadena y crea el array a partir de la cadena
//document.cookie, comprobando ciertas situaciones especiales
function extraeCookies(){

   //Variables que almacenaran las cadenas nombre y valor
   var nombre, valor;

   //Variables que controlarán los límites que marcan la posición de las
   //distintas cookies en la cadena
   var inicio, medio, fin;

   //El siguiente bucle comprueba si existe alguna entrada en el array
   //asociativo, de forma que si es así se crea una nueva instancia del
   //objeto cookies para eliminarla
   for (name in cookies){
      cookies = new Object();
      break;
   }
inicio = 0;

   //Se realiza un bucle que captura en cada paso el nombre y valor de
   //cada cookie de la cadena y lo asigna al array asociativo
   while (inicio< document.cookie.length){
      //la variable medio almacena la posición del próximo carácter "="
```

```
        medio =  document.cookie.indexOf('=', inicio);

        //la variable fin almacena la posición del próximo carácter ";"
        fin = document.cookie.indexOf(';',inicio);

        //El siguiente if comprueba si fin adquiere el valor -1 que indica
        //que no se ha encontrado ningún carácter ";" lo cual indica que
        //se ha llegado a la última cookie y por tanto se asigna a la
        //variable fin la longitud de la cadena
        if (fin == -1) {
           fin = document.cookie.length;
        }

        //El siguiente if realiza dos comprobaciones, en primer lugar si
        //medio es mayor que final o medio es -1 (que indica que no se
        //ha encontrado ningún carácter "=") la cookie tiene nombre pero
        //no valor asignado
        //En otro caso el nombre de la cookie se encuentra entre los
        //caracteres situados entre inicio y medio y el valor de la cookie
        //entre los caracteres situados entre medio+1 y final
        if ( (medio > fin) || (medio == -1) ) {
           nombre = document.cookie.substring(inicio, fin);
           valor = "";
        } else {
           nombre = document.cookie.substring(inicio,medio);
           valor= document.cookie.substring(medio+1,fin);
        }

        //Una vez recuperado el nombre y el valor se asigna al array
        //asociativo aplicando la función de conversión unescape()
           cookies[nombre] = unescape(valor);

           //En el siguiente paso del bucle while, la variable inicio adquiere
           //el valor fin + 2 saltando de esta forma el punto y coma y el
           //espacio que separa las distintas cookies en la cadena
           inicio = fin + 2;
     }
  }
```

La eliminación de una *cookie* se realiza asignando una fecha de caducidad del pasado, por ejemplo se puede utilizar la fecha “01-Jan-1970 00:00:01 GMT”.

El único problema aparece en el caso de que la *cookie* no tenga un valor asignado; estos casos deben de ser tratados de forma especial como se observa en el siguiente ejemplo:

```
  function eliminaCookie(nombre){
```

```
    //Se actualiza la cookie modificando la fecha de caducidad y asignando
    //un valor cualquiera, en el ejemplo borrada
    document.cookie = nombre + "=borrada; expires=Thu, 01-Jan-1970 00:00:01 GMT";
    //Se actualiza la cookie sin valor indicando una fecha de caducidad
    //anterior
    document.cookie = nombre + "; expires=Thu, 01-Jan-1970 00:00:01 GMT";
}
```

Sobre la anterior función, cabe comentar que en el caso de que la *cookie* se haya creado con información sobre el dominio y la ruta, es necesario introducir esta información en la cadena que actualiza o borra la *cookie*.

Las anteriores funciones proporcionan mecanismos suficientes para el control de las *cookies*, aunque estas pueden ser modificadas dependiendo de las necesidades del programador.

A continuación se van a crear dos *cookies*, con nombre "nombre" y "email" con los valores "Victor" y "vriospazos@uoc.edu" respectivamente:

```
asignaCookie("nombre","Victor");
asignaCookie("email","vriospazos@uoc.edu");
```

Se obtiene el *array* asociativo con las *cookies* almacenadas llamando a la función extraeCookies() y se pueden utilizar estas:

```
extraeCookies();
var nom = cookies["nombre"];
var corr = cookies["email"];
```

Se eliminan las dos *cookies* utilizando la función eliminaCookie():

```
eliminaCookie("nombre");
eliminaCookie("email");
```

3.2.3. Principales usos de las *cookies*

Actualmente se utilizan las *cookies* en los siguientes casos:

- Se utilizan *cookies* para almacenar el estado del usuario, adaptando la presentación o el contenido de la página basándose en las preferencias de este.

- Redireccionar el acceso a una página distinta cuando el usuario cumple ciertas condiciones, por ejemplo, un usuario registrado pasará directamente a las páginas de contenido, mientras que si es la primera vez que el usuario accede se le dirige a la página de registro.

- Al igual que en el caso anterior, un usuario que accede por primera vez se le puede abrir una ventana de información inicial, de manera que esta solo se abre la primera vez que accede a la página.

3.2.4. Limitaciones

Cada navegador impone un conjunto de limitaciones sobre el tamaño y el número de *cookies* que se pueden establecer. Aunque la recomendación RFC 2109 establece unas mínimas limitaciones:

- por lo menos 300 *cookies*;
- mínimo de 4096 bytes por *cookie* (de acuerdo con el tamaño de los caracteres que componen la *cookie* no terminal en la descripción de la sintaxis del encabezado Set-Cookie), y
- al menos 20 *cookies* por nombre de *host* o dominio único.

Evidentemente se trata de limitaciones aproximadas, ya que cada navegador aplica sus propias limitaciones.

3.3. Web Storage

Web Storage aparece con HTML5 y se presenta como alternativa a las *cookies* para almacenar información de las páginas web en el lado del cliente. A diferencia de las *cookies*, que solo permiten almacenar hasta 4 KB de información, con Web Storage podemos almacenar hasta 5 MB; esta capacidad viene definida por el navegador, de modo que dependiendo del navegador que utilicemos esta capacidad puede ser mayor.

A diferencia de las *cookies*, los datos almacenados con Web Storage no se envían automáticamente al servidor, lo que reduce el peso de cada petición al servidor; por este mismo motivo la información almacenada solo puede ser accedida desde la parte del cliente, es decir, no existe un mecanismo para acceder directamente desde un lenguaje de programación del lado del servidor como podría ser PHP.

Existen dos tipos de almacenamiento: localStorage y sessionStorage. La diferencia entre ambos es que el primero conserva la información permanentemente hasta que se limpie la caché del navegador y en el segundo la información permanece hasta que se cierra la sesión, o lo que es lo mismo hasta que se cierra la pestaña que contiene nuestra web.

Tabla 20. Soporte de Web Storage en los diferentes navegadores de internet

IE	Firefox	Chrome	Safari	Opera	Safari iOS	Android
8+	3,5+	5+	4+	10,5+	2+	2+

La manera de archivar y acceder a los datos es igual en ambos casos, así que nos vamos a centrar en localStorage.

Podemos saber si el navegador del usuario es compatible con Web Storage utilizando el código siguiente:

```
<script>
   //Podemos utilizar el objeto localStorage o el sessionStorage
   if(localStorage){
      //Existe localStorage y podemos hacer almacenamiento
      alert("El navegador es compatible con localStorage ");
   } else {
      //No existe localStorage
      alert("El navegador no es compatible con localStorage ");
   }
<script>
```

3.3.1. Utilizar Web Storage

Almacenar un valor

Cuando vayamos a almacenar datos utilizando Web Storage, hemos de tener en cuenta un par de cosas: los datos solo podrán ser cadenas de texto y cada dato que almacenemos está relacionado con la clave que le hayamos asignado.

Existen tres maneras diferentes de almacenar un valor:

- Mediante la inserción del par clave-valor:

```
localStorage.setItem('clave','valor');
```

- Mediante la inserción de un nuevo elemento como si localStorage fuese una tabla asociativa:

```
localStorage['clave'] = 'valor';
```

- Tratando a localStorage como un objeto:

```
localStorage.clave = 'valor';
```

Recuperar un valor

Del mismo modo que para la asignación de valores teníamos tres maneras distintas de hacerlo, para recuperar un valor también las tenemos:

- Utilizando el método getItem():

```
valor = localStorage.getItem('clave');
```

- Tratando de acceder al valor utilizando la clave como si localStorage fuese una tabla asociativa:

```
valor = localStorage['clave'];
```

- Tratando a localStorage como un objeto y utilizando la clave como si fuese una propiedad de este:

```
valor = localStorage.clave';
```

Eliminar valores

Podemos eliminar un determinado valor almacenado utilizando el método removeItem del siguiente modo:

```
localStorage.removeItem('clave');
```

Si lo que deseamos es eliminar todos los valores almacenados, podemos hacerlo con el método clear:

```
localStorage.clear();
```

Otras operaciones

Dado que localStorage se comporta como una tabla asociativa, podemos consultar cuántos elementos tenemos almacenados de la siguiente manera:

```
var total = localStorage.length;
```

También podemos obtener el nombre de una determinada clave usando *localStorage.key(i)*, donde *i* es la posición en la que se encuentra la clave.

Con este método, si nos interesa, podemos recorrer todos los elementos almacenados en localStorage. Por ejemplo, con este código escribimos todas las claves y sus valores correspondientes:

```
<script>
   for (var i=0; i < localStorage.length; i++) {
```

```
        document.write("La clave "+localStorage.key(i)+" almacena el valor
         "+localStorage[localStorage.key(i)]);
    }
</script>
```

Si queremos hacer pruebas en local con Web Storage, es posible que en algunos navegadores, como Internet Explorer, sea necesario ejecutar el documento HTML utilizando un servidor web local.

Web recomendada

En la revista *Mosaic* podéis ver algunos ejemplos en funcionamiento: http://mosaic.uoc.edu/2014/02/11/web-storage/

Actividades

1. Describid tres ejemplos de objetos del mundo real:

- Para cada uno de ellos definid la clase a la que pertenecen.
- Asignad a cada clase un identificador descriptivo adecuado.
- Enumerad varios atributos y operaciones para cada una de las clases.

2. Cread una clase para cada uno de los objetos planteados en el ejercicio anterior utilizando el lenguaje Javascript.

3. Cread una página web en la que se creen por lo menos dos objetos de las clases anteriores y se utilicen sus métodos y propiedades.

4. Cread un reloj donde cuya actualización sea manual a partir de un clic en un botón por parte del usuario de la web.

5. Utilizando el objeto *string*, cread dos funciones:

- encripta(texto): que sustituye cada uno de os caracteres de la cadena por otros, de forma que el texto resultante quedará ininteligible.
- desencripta(texto): que realiza la sustitución inversa a la de la función anterior.

Utiliza las dos funciones anteriores en una página web, donde, a partir de un texto introducido por el usuario, este se encripta o de desencripta.