Aprender JavaScript Avanzado

con 100 ejercicios prácticos

MEDIAactive

Alfaomega

marcombo
ediciones técnicas

Aprender

JavaScript Avanzado

con 100 ejercicios prácticos

Alfaomega

marcombo
ediciones técnicas

Diseño de la cubierta: *NDENU DISSENY GRÀFIC*

Datos catalográficos
MEDIAActive
Aprender JavaScript Avanzado con 100 ejercicios prácticos
Primera Edición
Alfaomega Grupo Editor, S.A. de C.V., México
ISBN: 978-607-622-621-6
Formato: 17 x 23 cm Páginas: 216

Aprender JavaScript Avanzado con 100 ejercicios prácticos
MEDIAActive
ISBN: 978-84-267-2241-6, edición en español publicada por MARCOMBO, S.A., Barcelona, España

Primera edición: Alfaomega Grupo Editor, México, febrero 2016

Pitágoras 1139, Col. Del Valle, 03100, México D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.mx**
E-mail: **atencionalcliente@alfaomega.com.mx**

ISBN: 978-607-622-621-6

Nota importante:
La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro, ni por la utilización indebida que pudiera dársele.

Edición autorizada para venta en México y todo el continente americano.

Impreso en México. Printed in Mexico.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Pitágoras 1139, Col. Del Valle, México, D.F. – C.P. 03100.
Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396
E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia,
Tels.: (57-1) 746 0102 / 210 0415 – E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – Av. Providencia 1443. Oficina 24, Santiago, Chile
Tel.: (56-2) 2235-4248 – Fax: (56-2) 2235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Paraguay 1307 P.B. Of. 11, C.P. 1057, Buenos Aires, Argentina, – Tel./Fax: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaeditor.com.ar

Presentación

APRENDER JAVASCRIPT AVANZADO CON 100 EJERCICIOS PRÁCTICOS

100 ejercicios, en este caso más teóricos, que conforman una guía de los principales elementos que forman parte del lenguaje de programación JavaScript. Si bien es imposible recoger en las páginas de este libro todas las prestaciones de estos elementos, hemos escogido las más interesantes e imprescindibles. Una vez consultados los 100 ejercicios que componen este manual, el lector será capaz de comprender por sí mismo cómo se consigue la interactividad más habitual de una página web.

LA FORMA DE APRENDER

Nuestra experiencia en el ámbito de la enseñanza nos ha llevado a diseñar este tipo de manual, en el que cada una de las funciones se ejercita mediante la realización de un ejercicio práctico. Dicho ejercicio se halla explicado paso a paso y pulsación a pulsación, a fin de no dejar ninguna duda en su proceso de ejecución. Además, lo hemos ilustrado con imágenes descriptivas de los pasos más importantes o de los resultados que deberían obtenerse y con recuadros IMPORTANTE que ofrecen información complementaria sobre cada uno de los temas tratados en los ejercicios.

Gracias a este sistema se garantiza que una vez realizados los 100 ejercicios que componen el manual, el usuario podrá desenvolverse cómodamente con el citado lenguaje de programación.

A QUIÉN VA DIRIGIDO EL MANUAL

Si se inicia usted en la práctica y el trabajo con JavaScript, encontrará en estas páginas un completo recorrido por sus principales funciones. Pero si es usted un experto en este elemento de programación, le resultará también muy útil para consultar determinados aspectos más avanzados o repasar funciones específicas que podrá localizar en el índice.

Cada ejercicio está tratado de forma independiente, por lo que no es necesario que los realice por orden (aunque así se lo recomendamos, puesto que hemos intentado agrupar aquellos ejercicios con temática común). De este modo, si necesita realizar una consulta puntual, podrá dirigirse al ejercicio en el que se trata el tema y llevarlo a cabo sobre su propio proyecto.

JAVASCRIPT

Javascript es un lenguaje de programación que permite crear programas y sitios web prácticamente idénticos a cualquier aplicación de escritorio, en cuanto a su respuesta se refiere. JavaScript aporta animaciones, interactividad y efectos visuales a un documento HTML, la base de cualquier sitio o página web.

No sólo encontramos JavaScript en las entrañas de muchos de los sitios web que visitamos diariamente. La gran mayoría de los widgets de escritorio que pueden instalarse en el ordenador han sido desarrollados con JavaScript, así como miles y miles de aplicaciones para dispositivos móviles y un amplio elenco de características de programas informáticos (como Photoshop, Dreamweaver, Acrobat, entre otros).

A diferencia de la mayoría de los libros que conforman esta colección, el que tiene usted entre las manos ha sido diseñado como guía de consulta de los principales elementos del lenguaje de programación JavaScript. Es por eso que no encontrará en estas páginas ejercicios resueltos, sino únicamente explicaciones detalladas, con ejemplos, de funciones, operadores, métodos, etc. de este lenguaje de programación tan utilizado para conseguir interactividad.

Cómo funcionan los libros "Aprender..."

El título de cada ejercicio expresa sin lugar a dudas en qué consiste éste. De esta forma, si le interesa, puede acceder directamente a la acción que desea aprender o refrescar.

Los ejercicios se han escrito sistemáticamente paso a paso, para que nunca se pierda durante su realización.

El número a la derecha de la página le indica claramente en qué ejercicio se encuentra en todo momento.

Los recuadros Importante incluyen acciones que deben hacerse para asegurarse de que realiza el ejercicio correctamente y también contienen información que es interesante que aprenda porque le facilitarán su trabajo con el programa.

En la parte inferior de todas las páginas puede seguir el ejercicio de forma gráfica y paso a paso. Los números de los pies de foto le remiten a entradas en el cuerpo de texto.

Índice

Índice

Qué es JavaScript

IMPORTANTE

Resulta de gran importancia saber que **JavaScript** no tiene nada que ver con **Java**, otro lenguaje de programación creado por la compañía SunSystems. De hecho, cabe destacar que, en sus inicios, Netscape bautizó su lenguaje de programación como **LiveScript** y que no fue hasta más adelante, por motivos de márketing, que la compañía decidió cambiar el nombre a su producto, asemejándolo al del lenguaje más exitoso en aquel momento.

JAVASCRIPT ES UN LENGUAJE DE PROGRAMACIÓN que permite crear programas y sitios web prácticamente idénticos a cualquier aplicación de escritorio, en cuanto a su respuesta se refiere. JavaScript aporta animaciones, interactividad y efectos visuales a un documento HTML, la base de cualquier sitio o página web. JavaScript forma parte de un conjunto de elementos formado por HTML y CSS: HTML aporta la estructura, CSS, el estilo y JavaScript, el funcionamiento y la interactividad con el usuario.

1. JavaScript fue creado en el año 1995 por la compañía Netscape, responsable de uno de los primeros navegadores que aparecieron en el mercado, Netscape Navigator, competencia directa de Microsoft Internet Explorer. [1] Dicha competencia provocó que el funcionamiento de JavaScript en dichos navegadores no fuera tan satisfactorio como se hubiera esperado de un lenguaje de programación tan potente. Afortunadamente, hoy en día, todos los navegadores, incluidos Firefox, [2] de Mozilla, y Safari, de Apple, [3] aceptan programas o *scripts* generados con JavaScript.
2. Teniendo en cuenta que el principal objetivo de JavaScript es la interactividad en programas y sitios web, a continuación

1

Logotipo de la compañía Netscape, creadora del navegador Navigator y del lenguaje de programación JavaScript.

2

3

veremos algunas de las aplicaciones en las cuales podemos encontrar claramente este increíble lenguaje de programación.

3. Gracias a JavaScript, es posible que una página muestre el importe total de una serie de compras, así como el IVA correspondiente aplicado, justo en el momento en que el usuario pulsa el botón **Comprar**. 4
4. Otro ejemplo sería al rellenar un formulario, en aquellos casos en que el usuario no completa los campos necesarios o lo hace de un modo no satisfactorio. En estos casos, el programa lanzará el correspondiente mensaje de error también generado con JavaScript. 5
5. Google Maps es otro claro ejemplo de cómo JavaScript permite interactuar con el usuario en tiempo real. 6
6. Pero no sólo encontramos JavaScript en las entrañas de muchos de los sitios web que visitamos diariamente. La gran mayoría de los widgets de escritorio que pueden instalarse en el ordenador han sido desarrollados con JavaScript, así como miles y miles de aplicaciones para dispositivos móviles y un amplio elenco de características de programas informáticos (como Photoshop, Dreamweaver, Acrobat, entre otros).
7. JavaScript es la llave a un mundo de interactividad prácticamente infinito y espectacular. Compruébelo usted mismo y... ¡disfrute!

001

IMPORTANTE

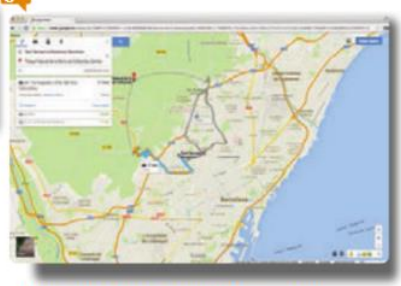
Inmediatamente después de que Netscape presentara su sorprendente lenguaje de programación JavaScript, la compañía Microsoft también se propuso atacar el mercado con su propio lenguaje de programación y lanzó jScript. En este caso, jScript estaba incluido en el navegador Internet Explorer.

Microsoft Developer

4

5

6



Google Maps utiliza JavaScript para conseguir una interacción con el usuario en tiempo real sin necesidad de cambiar de página web.

La sintaxis de JavaScript

IMPORTANTE

El uso del signo de puntuación ; (punto y coma) al finalizar cada sentencia no es obligatorio en JavaScript, aunque sí recomendable.

CUANDO HABLAMOS DE SINTAXIS DE UN lenguaje de programación nos estamos refiriendo al conjunto de reglas en que se basa la escritura del código fuente de los diferentes programas generados. La sintaxis de JavaScript es muy parecida a la de otros lenguajes de programación, como Java o C.

1. En este ejercicio trataremos de definir de forma clara y concisa en qué se basa la sintaxis básica de JavaScript. Aunque pueda parecer lo contrario, la sintaxis de JavaScript es bastante sencilla. Para empezar, podemos indicar que los intérpretes de JavaScript no tienen en cuenta los espacios en blanco ni las nuevas líneas, lo que supone que el desarrollador puede organizar el código como mejor le convenga.
2. JavaScript hace distinción entre mayúsculas y minúsculas, lo que significa que si el código no se escribe según esta distinción, el programa no funcionará según lo previsto.
3. Otro aspecto a tener en cuenta es que las variables creadas pueden almacenar diferentes tipos de datos, por lo que no es necesario definirlas al escribir el código.
4. JavaScript es un lenguaje de programación totalmente dinámico. ¿Qué significa esta afirmación? Sencillamente significa que todo cuanto se escribe en el código puede ser cambiado en cualquier momento.

1

```
<script type="text/javascript">
var hora = new Date();
var horas = hora.getHours();
 if (horas<12) {
  formatoHora = 'am';
 } else {
  formatoHora = 'pm';
 }
 horas = horas % 12;
  if (horas==0) {
   horas = 12;
  }
var min = hora.getMinutes();
 if (min<10) {
   min = '0' + min;
 }
var seg = hora.getSeconds();
 if (seg<10) {
   seg = '0' + seg;
  }
document.write("Hora actual: " + horas + ":" + min + ' ' +  formatoHora);
</script>
```

2

```
<script type="text/javascript">
function fechaCompleta(fecha,formato) {
 var diaSemana=fecha.getDay()-1;
   if (diaSemana <0){
        diaSemana=6
    }
```

La distinción entre mayúsculas y minúsculas debe realizarse y respetarse durante todo el script.

002

5. Los comentarios también forman parte de la sintaxis de JavaScript. Resultan de gran importancia porque permiten al desarrollador realizar anotaciones referentes a la escritura del código, utilizando para ello unas normas imprescindibles. Como veremos en el ejercicio siguiente, los comentarios se escriben siguiendo una puntuación básica concreta.
6. Las variables también son parte de la sintaxis de Javascript. Las variables se declaran mediante la terminología `var` e incluyen siempre valores. Los cinco tipos de valores fundamentales son booleanos, números, cadenas de caracteres, objetos plenos y matrices; todos estos tipos serán debidamente descritos en este libro.
7. El desarrollador "inventa" el nombre de las variables declaradas, siempre teniendo en cuenta dos aspectos fundamentales: que dicho nombre sólo puede contener letras, números y los signos $ y _ (guión bajo) y que el primer caracter no puede ser un número. Una declaración de variable sería la siguiente:
   `var abc;`
8. Y la asignación de valores, podría ser como sigue:
   `var abc = alfabeto;`
9. Como en las variables, las funciones adoptan un nombre único para poder ser utilizadas dentro del código. Tras este nombre, es preciso escribir los parámetros de la función entre paréntesis. Las instrucciones que describen la función se escriben entre símbolos de llaves { }.

3

```
document.write('<p>El lugar que más me ha sorprendido es <em>');
document.write(countries[countries.length-1] + '</em></p>');
countries.unshift('Islandia'); //La lista tiene ahora 8 elementos
document.write('<p>El lugar más frío es <em>');
document.write(countries[0] + '</em></p>');
countries.shift();
document.write('<p>El lugar más grande es <em>');
```

6

```
function askQuestion(pregunta) {
 var respuesta = prompt(pregunta[0],'');
  if (respuesta == pregunta[1]) {
   alert('¡Correcto!');
   puntos++;
  } else {
   alert('Lo sentimos. La respuesta corr
  }
}
```

5

```
var fechaActual = new Date();
var fechaFutura = new Date(2015,2,01);
var diferencia = fechaFutura.getTime() - fechaActual.getTime();
var totalDias = Math.round(diferencia/(1000 * 60 * 60 * 24));
document.write("¡Faltan " + totalDias + " días para la inauguración!");
```

4

```
<script type="text/javascript">
var valor1 = '5';
var valor2 = '8';
document.write(Number(valor1) + Number(valor2));
</script>
```

Comentarios, expresiones y sentencias

IMPORTANTE

Los **comentarios** insertados por el programador dentro del código pueden ser frases enteras o palabras sueltas. En cualquier caso, siempre ayudarán a aclarar partes concretas del código, sobre todo en caso de necesitar modificarlo o en el momento de su depuración.

LOS COMENTARIOS, LAS EXPRESIONES Y LAS sentencias forman parte de la sintaxis básica de JavaScript. En ocasiones existe cierta confusión entre la definición y el uso de expresiones y sentencias (la asignación de comentarios queda totalmente clara, como se verá en este ejercicio); en estas páginas trataremos de distinguir por completo ambos elementos.

1. Empezamos este ejercicio hablando de los comentarios. Los **comentarios** forman parte del código JavaScript pero no son interpretados por los navegadores web. Los programadores los utilizan sencillamente para facilitar la lectura del código en determinadas partes del mismo.
2. Existen dos tipos de comentarios de JavaScript y cada uno de ellos se representa con una puntuación distinta. El primer tipo se utiliza para comentar únicamente una línea de código y se indica con una doble barra, //.
3. El segundo tipo de comentario se utiliza para comentar más de una línea de código y requiere, en este caso, de una puntuación de apertura, /*, y otra de cierre, */.
4. El uso de comentarios en un documento de código JavaScript es ilimitado e, incluso, recomendable en aquellos casos en que se prevean confusiones en el momento de la interpretación del mismo.

Cuando se utiliza un editor de textos web, los **comentarios** se muestran de color gris, para que sea más fácil reconocerlos en el momento de la depuración o modificación.

003

5. Hablemos ahora de las expresiones y las sentencias, que, en determinados casos y tal como hemos indicado en la introducción de este ejercicio, pueden confundirse. Las **sentencias** son instrucciones que se incorporan en el código para que éste ejecute algo, por ejemplo, que realice una acción concreta.
6. Un ejemplo bien conocido de sentencia es la denominada condicional, la cual se utiliza para determinar si algo es verdadero o si existe. En función de si la condición es verdadera o no, se ejecutarán o no las acciones especificadas, ya sean funciones o expresiones. La sentencia condicional es `if`.
7. A diferencia de las sentencias, las **expresiones** son cualquier combinación de símbolos de JavaScript que representan un valor. Así, las expresiones tienen valores (de los cuales hablamos en el ejercicio anterior) y los valores y las propiedades tienen tipos.
8. Una expresión puede estar formada por diferentes tipos de elementos: operadores y operandos, valores, funciones y procedimientos.
9. Dicho esto, podemos destacar que cuando JavaScript espera la aparición de una sentencia, es posible que aparezca una expresión, pero no al contrario: no es posible escribir una sentencia cuando el programa está esperando una expresión.
10. A menudo, y debido a su construcción, se utilizan erróneamente dos tipos concretos de expresiones como si fueran sentencias: los objetos literales (expresiones) se confunden con los bloques (sentencias) y las expresiones de funciones con nombre se confunden con declaraciones de funciones, las cuales son sentencias.

IMPORTANTE

Es posible escribir **sentencias condicionales** utilizando una **expresión** con un operador condicional. Por ejemplo, éstas serían las líneas de código con la sentencia condicional `if`:

```
var premio;
if (si) {
  premio = ¡Enhorabuena!';
} else {
  premio ='Lo sentimos';
}
```

Y ésta la misma línea con el operador condicional:

```
var premio = si ?
'¡Enhorabuena!' : 'Lo sentimos';
```

(Todo se escribe en una misma línea.)

if

3

```
<script type="text/javascript">
if (bestCountry == 'Canadá') {
  document.write("<p>¡Coincido contigo!
} else if (bestCountry == 'Argentina' ||
  document.write("<p>Aún no conozco " +
} else {
  document.write("<p>¡Sí! " + bestCountr
}
</script>
```

Podemos decir que un programa o un script es una secuencia de sentencias.

Los bucles (*loops*) también son un ejemplo de sentencia en JavaScript.

4

```
var aleatorio = Math.floor(Math.random() *preguntas.length);
var aleat = preguntas[aleatorio];
```

En la descripción de estas dos variables, todo cuanto se encuentra a la derecha del signo =, el cual indica la descripción de la variable en sí misma, es una expresión.

El uso del punto y coma en JavaScript

EL USO DEL PUNTO Y COMA al final de cada sentencia no es obligatorio en el código JavaScript, aunque sí altamente recomendable. En este ejercicio le mostraremos por qué es recomendable utilizar este signo de puntuación y en qué casos precisamente no debe utilizarse.

1. Con el fin de evitar posibles errores detectados por el navegador web en el momento de la interpretación, las diferentes instrucciones contenidas en nuestros scripts deben separarse adecuadamente. ¿Cómo? Mediante un punto y coma al final de cada línea. 1
2. Sin embargo, existen ciertas excepciones a esta norma del punto y coma. En concreto, las sentencias que finalizan con un bloque no deben llevar punto y coma al final de las mismas. Y todavía podemos concretar más esta excepción: sólo los bucles (como `for` y `while`), los conectores (como `if`, `switch` o `try`) y las declaraciones de funciones que finalizan con un bloque no se cierran con un punto y coma.
3. Éste sería un ejemplo de una sentencia con un bucle `for` que no finaliza con un punto y coma: 2

```
for (var num=11; num<=20; num++) {
 document.write('<h3>Número ' + num + '<br></h3>');
}
```

1

```
document.write('<p>El lugar más bonito es <em>');
document.write(countries[3] + '</em></p>');
document.write('<p>El lugar con más historia es <em>');
document.write(countries[0] + '</em></p>');
document.write('<p>El lugar que más me ha sorprendido es <em>');
document.write(countries[countries.length-1] + '</em></p>');
countries.unshift('Islandia'); //La lista tiene ahora 8 elementos
document.write('<p>El lugar más frío es <em>');
document.write(countries[0] + '</em></p>');
countries.shift();
document.write('<p>El lugar más grande es <em>');
document.write(countries[4] + '</em></p>');
countries.pop();
document.write('<p>El lugar más fascinante es <em>');
document.write(countries[countries.length-1] + '</em></p>');
countries.splice(3,0, 'Austria', 'México', 'Panamá');
document.write('<p>El lugar más imperial es <em>');
document.write(countries[3] + '</em></p>');
countries.splice(6,1);
document.write('<p>El lugar en que más me he cansado es <em>');
document.write(countries[6] + '</em></p>');
countries.splice(1,2, 'Canadá', 'Egipto');
```

2

```
<script type="text/javascript">
  if (numFav) {
   document.write('<p>El número ' + numFav + ' es un buen
  }
  for (var num=11; num<=20; num++) {
   document.write('<h3>Número ' + num + '<br></h3>');
  }
</script>
```

for

004

4. Vea que la sentencia finaliza con la llave de cierre, y no con este signo y un punto y coma. Seguidamente, incluimos un ejemplo de una sentencia con un conector condicional como `if` que tampoco finaliza con un punto y coma:

```
if (bestCountry == 'Canadá') {
document.write("<p>¡Coincido contigo! China también</p>");
}
```

5. Por último, un ejemplo de la tercera de las excepciones listadas en el punto 2, es decir, una declaración de función, serían las líneas siguientes:

```
function printToday() {
 var hoy = new Date(); document.write(hoy.getDate());
}
```

6. Sin embargo, sepa que la inserción de un punto y coma en estos casos concretos no conlleva un error de sintaxis, debido a que dicho signo se considera una sentencia vacía y las sentencias vacías pueden aparecer en el punto en que el código prevé encontrar una sentencia.

7. JavaScript cuenta con un mecanismo que se ocupa de insertar un punto y coma allá donde prevé que es necesario. Este mecanismo se denomina ASI, siglas del término en inglés *Automatic Semicolon Insertion*, y aunque a primera vista puede resultar muy útil, en ocasiones produce resultados inesperados. Por esta razón, lo mejor y más recomendable es utilizar el punto y coma nosotros mismos allí dónde sea necesario.

IMPORTANTE

Básicamente, el mecanismo ASI funciona al determinar el punto en que una sentencia finaliza y le agrega un punto y coma.

3

```
<script type="text/javascript">
if (bestCountry == 'Canadá') {
  document.write("<p>¡Coincido contigo! Canadá también es mi país favorito</p>");
} else if (bestCountry == 'Argentina' || bestCountry == 'Portugal') {
  document.write("<p>Aún no conozco " + bestCountry + ". Si vuelves algún día, avísame ;-)</p>");
} else {
  document.write("<p>Sí! " + bestCountry + " también es muy bonito</p>");
}
</script>
```

}

4

```
function printToday() {
var hoy = new Date(); document.write(hoy.getDate());
}
```

Datos primitivos

COMO LA MAYORÍA DE LENGUAJES DE programación, Javascript cuenta con los tipos de datos habituales de este tipo de lenguajes: booleanos, cadenas de caracteres, números, entre otros. Estos datos se confunden en ocasiones con otros elementos propios también de este lenguaje: los objetos. En éste y en el siguiente ejercicio, trataremos de describir y distinguir ambos tipos de elementos.

1. Como hemos indicado en la introducción, JavaScript cuenta con los siguientes tipos de datos: booleanos, números, cadenas de caracteres, matrices y los tipos especiales `null` y `undefined`. Todos estos datos, que se denominan primitivos, tienen propiedades, las cuales, a su vez, tienen un nombre y un valor.
2. Los datos primitivos tienen unas características que los identifican. Veamos cuáles son, aunque sepa que cada uno de ellos será tratado ampliamente más adelante en este libro.
3. El tipo de dato booleano (en inglés, *boolean*) acepta sólo dos valores:

   ```
   true
   false
   ```

4. Y normalmente se utilizan en sentencias condicionales.

1

Debe saber que en realidad los datos primitivos no tienen propiedades, sino que las toman prestadas de los constructores que los envuelven. Hablaremos de dichos constructores más adelante en este libro. Imagen: aprenderaprogramar.com.

2

```
<script type="text/javascript">
var hora = new Date();
var horas = hora.getHours();
 if (horas<12) {
  formatoHora = 'am';
 } else {
  formatoHora = 'pm';
 }
 horas = horas % 12;
  if (horas==0) {
   horas = 12;
  }
var min = hora.getMinutes();
 if (min<10) {
   min = '0' + min;
 }
var seg = hora.getSeconds();
 if (seg<10) {
   seg = '0' + seg;
 }
```

005

5. Los tipos numéricos (en inglés, *number*) representan cualquier número entero o de punto flotante, es decir, con decimales (`23.7`). 3

```
var valor1 = 5;
var valor2 = 2.9;
```

6. Por su parte, las cadenas de caracteres (en inglés, *string*) almacenan, como su nombre indica, una secuencia de caracteres, teniendo en cuenta que dicha secuencia puede estar vacía. Las cadenas de caracteres se representan en JavaScript entre comillas, ya sea simples o dobles. Por ejemplo: 4

```
var numFav = prompt('¿Cuál es tu número favorito?');
```

7. Da igual el tipo de comillas que utilice, aunque sí es importante que sea coherente dentro del documento con este uso. Existen dos tipos de datos primitivos más que se utilizan par indicar la falta de información. Se trata de los valores `undefined` y `null`. El primero significa que no existe ningún valor, ni tipo de dato ni objeto, en la sentencia, 5 y el segundo, que no existe ningún objeto.

8. De forma común, todos los datos primitivos tienen la característica de que contienen una comparación entre sus valores, que sus propiedades no pueden ser ni modificadas, ni alteradas ni eliminadas y, por último, que son predeterminados en JavaScript, lo que significa que el usuario no puede crear sus propios datos primitivos.

3

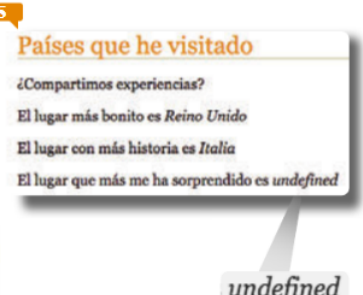
```
<script type="text/javascript">
var valor1 = '5';
var valor2 = '8';
document.write(Number(valor1) + Number(valor2));
</script>
```

Normalmente, el tipo de datos `Number` no utiliza comillas, como en el caso de las cadenas de caracteres. En este ejemplo, las comillas se utilizan porque los valores numéricos dependen del elemento `Number`, el cual lo utiliza para realizar un cálculo matemático.

4

```
<script type="text/javascript">
    do {
    var numFav = prompt('¿Cuál es tu número favorito?');
    numFav = parseInt(numFav);
    } while (isNaN(numFav));
</script>
```

5



Objetos en JavaScript

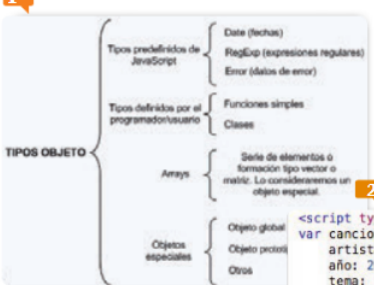
COMO HEMOS INDICADO EN EL EJERCICIO anterior, JavaScript cuenta con datos primitivos y con objetos. Todo cuanto no puede situarse dentro de alguno de los tipos de datos descritos en las páginas anteriores puede considerarse un objeto.

1. Existen tres tipos de objetos en JavaScript, considerados más comunes, que se mueven con los denominados constructores: los objetos simples, las matrices y las expresiones regulares. Los objetos tiene propiedades y valores y, a diferencia de los datos primitivos, sí pueden ser creados por el programador. Veamos paso a paso cada uno de estos tipos de objetos. 1
2. El constructor `Object` envuelve los objetos simples, los cuales se representan mediante nombres o datos. Un ejemplo de un objeto simple sería el siguiente:

```
{
 nombre: 'Mónica',
 apellido: 'González'
}
```

3. Este objeto tiene dos propiedades (`nombre` y `apellido`) y cada una de ellas, un valor (`'Mónica'` y `'González'`). Como hemos indicado, es el programador quien decide el nombre de los objetos, siendo consecuente en su uso en todo el script. 2

1



Fuente de esta imagen: aprenderaprogramar.com

Ésta sería la descripción de un pequeño script que cuenta con una variable que contiene un objeto con varias propiedades y valores.

2

```
<script type="text/javascript">
var cancion = {
    artista: "Daniel Mariano",
    año: 2013,
    tema: 'Todos somos todos',
    getInfo: function () {
      return this.tema + ':' + this.artista + ', ' + this.año +'.';
    }
}
alert(cancion.getInfo());
</script>
```

006

4. El segundo tipo destacable de objetos son las matrices, en inglés *arrays*. Una matriz es, sencillamente, una lista de elementos. Las matrices son la mejor forma de almacenar más de un valor en un mismo lugar, puesto que de otro modo sería necesaria crear una variable para cada elemento de la lista. Las matrices se envuelven dentro del constructor `Array` y tienen el aspecto siguiente: 3

```
[ 'lunes', 'martes', 'miércoles', 'jueves', 'viernes']
```

5. Todos los elementos de una matriz corresponden a un número en un índice, siendo el primero de estos números el 0; esto significa que, en nuestro ejemplo, el elemento `'lunes'` es el número 0, el elemento `'martes'`, el 1, y así sucesivamente. 4

6. El tercer tipo de objetos de JavaScript es el de las expresiones regulares, regidas por el constructor `RegExp`. Las expresiones regulares están formadas por caracteres simples, `/abc/`, o por una combinación de caracteres simples y especiales, `/a*b_c/`. Una expresión regular sirve para localizar patrones en una cadena de texto, para comprobar que una cadena tiene una determinada estructura o para verificar que una dirección de correo electrónico está escrita correctamente. 5

7. Las características que comparten estos tres tipos de objetos son las siguientes: cada objeto tiene su propia identidad y, por tanto, pueden compararse; es posible modificar las propiedades de los objetos, añadir de nuevas y eliminar las sobrantes, y los constructores pueden ser considerados como implementaciones de tipos personalizados.

IMPORTANTE

Existe un listado de caracteres especiales que pueden utilizarse en expresiones regulares. Tenga en cuenta que cada uno de estos caracteres se utiliza en combinación con caracteres simples y tienen un significado concreto. Los caracteres especiales que se pueden utilizar en JavaScript para generar expresiones regulares son los siguientes:
`< $ ^ . * + ? [ ] >`

La imagen 5 muestra un ejemplo de código en el cual se utiliza una expresión del tipo `RegExp, /@/`, para verificar que un campo de e-mail cuenta con una dirección de correo electrónico válida, es decir, con el símbolo @.

3

```
<script type="text/javascript">
    var countries = ['Italia',
                     'Francia',
                     'Dinamarca',
                     'Reino Unido',
                     'Alemania',
                     'Rusia',
                     'Grecia',
                     'China'
                    ];
```

5

```
{
  var texto=document.expresiones.data.value;
  var ejemplo= texto.match(/@/);
  if (ejemplo=="@") {
    document.expresiones.resultado.value="Si";
  }
  else {
    document.expresiones.resultado.value="No";
  }
}
```

4

```
countries.splice(3,0, 'Austria', 'México', 'Panamá');
document.write('<p>El lugar más imperial es <em>');
document.write(countries[3] + '</em></p>');
countries.splice(6,1);
document.write('<p>El lugar en que más me he cansado es <em>');
document.write(countries[6] + '</em></p>');
countries.splice(1,2, 'Canadá', 'Egipto');
```

Los elementos que forman una matriz se gestionan mediante valores de índice, siempre recordando que el primer elemento corresponde al valor 0, no 1.

Valores y objetos indefinidos

EN UN EJERCICIO ANTERIOR HEMOS TENIDO la oportunidad de mencionar dos tipos de valores que se utilizan en JavaScript para indicar la falta de información en un script. Se trata de los valores (o, también podríamos denominarlos, "no-valores") `null` y `undefined`.

1. En este ejercicio trataremos con todo detalle los dos "no valores" que JavaScript utiliza para indicarnos que falta información en el script: `null` y `undefined`. `Undefined` significa que no hay ni un valor primitivo ni un objeto, y podemos encontrarlo en variables sin inicializar, en una falta de parámetros o en una omisión de propiedades. Por su parte, `null` significa que no hay ningún objeto, y podemos encontrarlo en aquellas partes del script en que se espera la existencia de un objeto, sea del tipo que sea.
2. Así, podemos resumir esta descripción diciendo que `undefined` indica la no existencia y `null`, el vacío.
3. A continuación veremos en ejemplos los casos en que se generarán los valores `undefined` y `null`. Empezaremos por las variables no inicializadas comparadas con otras que sí lo han sido:

```
var caso1 = 452;
var caso2 = '452';
var caso3;
```

1

```
> (function() {})();
undefined
> (function() {})() == null;
true
> (function() {})() == undefined;
true
> (function() {})() === null;
false
> (function() {})() === undefined;
true
>
```

2

```
undefined = no existencia
null = vacío
```

3

```
var caso1 = 452;
var caso2 = '452';
var caso3;
```

Con esta comparativa puede ver claramente que `undefined` es un valor, aunque sea "sin valor".

007

4. La variable `caso1` contiene datos del tipo `Number`; la variable `caso2` contiene datos del tipo `String` mientras que la variable `caso3` no ha sido inicializada y es del tipo `undefined`.
5. El caso de la falta de parámetros también produce variables de este tipo. Un ejemplo sería el siguiente: [4]

```
var caso1 = { };
```

6. En efecto, la variable `caso1` sí ha sido inicializada pero no muestra ningún parámetro, por tanto, es del tipo `undefined`.
7. En JavaScript, el valor `null` se considera un valor especial, puesto que en sí mismo es un valor aunque indica la falta de contenido. En nuestras variables de ejemplo, el valor `null` sería como sigue: [5]

```
var caso1 = null;
```

8. Quizás se estará preguntando por qué JavaScript dispone de estos "no-valores" entre sus elementos. La razón se remonta a los inicios de este lenguaje de programación. JavaScript adoptó el procedimiento de dividir los valores en primitivos y objetos, así como de utilizar el valor para indicar la ausencia de objetos con el valor `null`.
9. Así, el valor `null` se convirtió en 0 (cero) si daba como resultado un número: [6]

```
Number(null)//El resultado sería 0
5 + null // El resultado sería 5
```

10. Y se inventó un nuevo no-valor, undefined, para aquellos casos en que el resultado no debiera ser un número, es decir, para variables sin inicializar y para la omisión de propiedades: [7]

```
Number(undefined) // El resultado sería NaN (Not a Number)
5 + undefined // El resultado sería NaN
```

4

```
var caso1 = { }; // es undefined
```

5

```
var caso1 = null; // es null porque no existe
```

6

```
Number(null)//El resultado sería 0
5 + null // El resultado sería 5
```

7

```
Number(undefined) // El resultado sería NaN (Not a Number)
5 + undefined // El resultado sería NaN
```

Comprobar valores indefinidos o nulos

EN ESTE EJERCICIO LE MOSTRAREMOS CÓMO puede comprobar si los valores `undefined` y `null` existen en un script por separado o bien si uno de los dos valores existe.

1. Empezaremos por el valor `null`. Si lo que necesita saber es si el valor de una variable es el mencionado no-valor, es decir, si no tiene ningún valor asociado, sencillamente es preciso comparar esa variable con el valor `null`. El modo de hacerlo pasa por el uso del comparador === (estrictamente igual): 1

```
if(caso1===null) {
}

//Siendo "caso1" el nombre de la variable que estamos analizando
```

2. En el caso en que la variable devuelva, efectivamente, el valor `null`, será preciso realizar cuanto se requiera para que el script funcione correctamente. 2

3. También podemos comprobar de la existencia de una variable `undefined` utilizando el mismo comparador que para el valor `null`: 3

```
if(caso1===undefined) {
}
```

1

```
if(caso1===null) {

} //Siendo "caso1" el nombre de la variable que estamos analizando
```

2

undefined==false	false
null==false	false
undefined==0	false
null==0	false
undefined==''	false
null==''	false
false==''	true
''==0	true
false==0	true
null==undefined	true

3

```
if(caso1===undefined) {

}
```

En ambos casos (1 y 3), el término `caso1` correspondería al nombre de una variable ficticia.

4. A continuación, le mostramos un ejemplo más gráfico de la comprobación de un objeto indefinido: [4]

```
if (typeof(caso1) === "undefined") {
  alert("caso1 no está definido.");
}
```

5. Sin embargo, no es ésta la única forma de realizar esta comprobación. El operador `Typeof` es otro modo de llevarla a cabo, aunque normalmente, y debido a su sencillez, se utilizará el comparador `===`.

6. `Typeof` es un operador que devuelve una cadena o string que describe el tipo de dato que corresponde con el objeto, ya sea una variable, una función..., que se escribe a continuación. Más concretamente, el operador `typeof` distingue los valores primitivos de los objetos y, en el caso de las primitivos, determina el tipo al que pertenecen: "number", "string", "boolean", "object", "function" y "undefined". [5]

7. El operador `typeof` de JavaScript también puede interpretar el valor `null`, y en este caso lo hace como un tipo Object, no como un tipo `null`. Este comportamiento, que puede provocar confusiones, se mantiene por compatibilidad con versiones anteriores del lenguaje.

8. La propiedad `SomeObject.prop` también se utiliza para comparar un valor no definido en null. ¿Cómo? Verificando que esta comparación es `true` si la propiedad `SomeObject.prop` es `null` o si dicha propiedad no existe. [6]

IMPORTANTE

Si el uso y la aplicación del operador Typeof no le ha quedado demasiado claro, no se preocupe; más adelante en este libro dedicaremos un ejercicio entero a dicho operador.

[4]

```
if (typeof(caso1) === "undefined") {
  alert("caso1 no está definido.");
}
```

[5]

```
var declarada;
var resultado = (declarada === undefined);
// true

var resultado = (typeof declarada === 'undefined');
// true

var resultado = (typeof noDeclarada === 'undefined')
// true
```

[6]

```
someObject.prop == null;
```

La propiedad `SomeObject.prop` también se utiliza para comparar un valor no definido en `null` comprobando que esta comparación es `true` si la propiedad `SomeObject.prop` es `null` o si dicha propiedad no existe.

Objetos envolventes para primitivos

LOS TRES TIPOS DE DATOS PRIMITIVOS en JavaScript, booleano, numérico y cadena de caracteres, tiene su correspondiente constructor: `Boolean`, `Number` y `String`. Los valores primitivos son contenidos o envueltos (*wrap*) en las instancias, también denominadas objetos envolventes (en inglés, *wraper objects*).

1. En este ejercicio veremos los detalles de los objetos que envuelven los primitivos. Para empezar diremos que los constructores crean objetos que son prácticamente incompatibles con los valores primitivos que envuelven.
2. Si estos constructores se utilizan como funciones, la conversión de los valores se lleva a cabo en sus correspondientes tipos de primitivos.
3. Aunque los expertos recomiendan este método de conversión, lo cierto es que entre los programadores de JavaScript se tiende a evitar los objetos envolventes. ¿Por qué? Porque la conversión no es un proceso necesario, puesto que casi todo cuanto pueden hacer los objetos también pueden hacerlo los primitivos.
4. Los objetos envolventes son distintos a los datos primitivos. Si tenemos en cuenta que las instancias envolventes son objetos,

1

```
typeof new String('caso1') // convierte a 'object'
new String('caso1') === 'caso1' // convierte a 'false'
```

2

```
String(caso1) // convierte en caso1
```

3

```
var 1 = new String('caso1');
var 2 = new String('caso1');
a == b
false
```

podemos deducir que no existe ningún modo de comparar objetos en JavaScript, ni mediante operadores de igualdad como `==`.

5. Si lo que necesita es añadir propiedades a un valor primitivo, será preciso envolver la primitiva y añadir la propiedad al objeto envolvente. Con este proceso lo que estamos haciendo es desenvolver el valor para poder trabajar con él. Vea un ejemplo de este proceso en las siguientes líneas:

```
new Boolean (true)
new Number(258)
new String('caso1')
```

6. Así los valores primitivos han sido envueltos con el correspondiente constructor (`Boolean`, `Number` y `String`). Para desenvolverlos, se utiliza el método `valueOf()`, el cual está presente en todos los objetos:

```
new Boolean (true).valueOf()
true
new Number(258).valueOf()
258
new String('caso1').valueOf()
'caso1'
```

7. Tal y como veremos con detalle en un ejercicio anterior, la conversión de objetos envolventes en primitivas genera números y texto (o cadenas de caracteres), pero no datos booleanos.

8. Por último, cabe destacar que los datos primitivos adquieren sus métodos de los correspondientes objetos envolventes.

4

```
new Boolean (true)
new Number(258)
new String('caso1')
// Así envolvemos los datos primitivos
con sus correspondientes constructores
```

Recuerde que todo cuanto escribimos tras la doble barra // no es interpretado por el navegador web.

5

```
new Boolean (true).valueOf()
true
new Number(258).valueOf()
258
new String('caso1').valueOf()
'caso1'
// Así desenvolvemos los primitivos
```

Operadores de asignación

IMPORTANTE

El resultado de las operaciones es aquello que permite a un programa modificar su manera de comportarse según los datos que tenga para trabajar. Las operaciones pueden ser más o menos sencillas y pueden realizarse con distintos tipos de operandos.

LOS OPERADORES SIRVEN PARA REALIZAR LOS cálculos y las operaciones definidos en un script. Todos los operadores convierten sus operandos en los tipos correspondientes, aunque hay que advertir que la mayoría de los operadores sólo funcionan con valores primitivos, lo que implica que los objetos se convierten en primitivos antes de ser manipulados.

1. En este ejercicio conocerá cuáles son los operadores de asignación, cómo y para qué se utilizan. Como presentación, diremos que los operadores de asignación son los que se utilizan para asignar valores a las variables. El operador de asignación más utilizado es =. A la derecha de este operador se escriben los valores finales, mientras que a su izquierda se inserta el nombre de la variable en la cual se almacenará el dato. Veamos un ejemplo básico:

```
var caso1 = 100;
```

2. Podemos definir una asignación como una expresión que evalúa un valor asignado. Esta definición permite en JavaScript encadenar asignaciones. Un ejemplo de ello sería lo siguiente:

```
caso1 = caso2 = 100;
```

3. Esta sentencia permite asignar el valor `100` tanto al valor `caso1` como al valor `caso2`. Aunque el valor = es el más utilizado en-

```
puntos = 0;
```

1

```
var puntos = 0;
var preguntas = [
 ['¿Cuál es la capital de Austria?', "Viena"],
 ['¿Cuál es la capital de Noruega?', "Oslo"],
 ['¿Qué lengua se habla en Madagascar?', "Malgache"],
 ['¿Cómo se llaman los habitantes de Panamá?', "Panameños"]
];
```

2

```
caso1 = caso2 = 100;
//Los operadores de asignación
permiten encadenar valores
```

El operador = se denomina operador de asignación simple.

tre los operadores de asignación, es preciso saber que existen otros; de hecho, se trata de combinaciones entre el operador = y otros operadores matemáticos. Algunas de estas combinaciones entre operadores de asignación son `+=`, `-=`, `*=` y `/=`. Todos ellos realizan la operación que indica el operador matemático entre el valor de la izquierda y el de la derecha y ubica el resultado en la parte izquierda de la sentencia. Veamos unos ejemplos de cada uno de ellos sobre un único caso:

```
caso1 = 103;
caso1 += 300;
caso1 -= 200;
caso1 *= 3;
caso1 /=2;
```

4. Cada uno de los operadores se basa en el valor de la variable `caso1` y va realizando la operación que indica el operador con el valor de la derecha; el resultado se sitúa en la parte izquierda de la sentencia siguiente. Existe otro operador de asignación combinado con el operador básico =; se trata del operador %= el cual permite calcular el resto de una operación y se asigna como resultado de la variable.
5. Además de asignar valores a variables, los operadores de asignación también pueden asignarse a propiedades, eventos o accesos a indizadores.
6. Los operadores de asignación son asociativos por la derecha. ¿Qué significa esto? Sencillamente, que las operaciones se agrupan de derecha a izquierda. Por ejemplo, una expresión de la forma `caso1 = caso2 = caso3` se evalúa como `caso1 = (caso2 = caso3)`.

3

```
caso1 = 103;
//La variable caso1 tiene el valor 103
caso1 += 300;
// Se suma el valor 300 al valor 103;
caso1 -= 200;
// Se resta 200 al resultado obtenido
en la sentencia anterior
caso1 *= 3;
// Se multiplica por 3 el resultado
obtenido en la sentencia anterior
caso1 /=2;
// Se divide entre 2 el resultado
obtenido en la sentencia anterior
```

Este tipo de operadores de asignación se denominan compuestos.

4

```
caso1 = caso2 = caso3
caso1 =(caso2 = caso3)
//Ambas sentencias son iguales
```

Operadores de igualdad

JAVASCRIPT DISPONE DE DOS TIPOS DE operadores para determinar la igualdad entre valores. Los operadores denominados de igualdad y de desigualdad estricta (=== y !==) y los denominados de igualdad y de desigualdad normal (== y !=). Por las razones que se verán en este ejercicio, los expertos recomiendan utilizar los operadores estrictos y evitar, en la medida de lo posible, los normales.

1. En este ejercicio trataremos de describir y comprobar cómo se utilizan los distintos operadores de igualdad existentes en JavaScript. Empezaremos por los operadores estrictos, tanto de igualdad como de desigualdad.
2. El operador de igualdad estricta se representa con los signos === (3 signos de igualdad) y el de desigualdad estricta con la combinación de signos !== (1 signo de exclamación cerrado y 2 signos de igualdad).
3. Estos operadores consideran sólo aquellos valores que son del mismo tipo para ser iguales. Los valores de diferentes tipos no son nunca estrictamente iguales. Si los dos valores escritos en la sentencia son del mismo tipo, entonces el resultado es estrictamente igual.

1

```
var caso1 = '125';
  if (caso1 === 125)
    document.write('Estos valores son iguales');
// En este caso, los dos valores son iguales
y la sentencia condicional daría false como resultado
porque son datos de distinto tipo
```

Los operadores de igualdad, sean del tipo que sean, no se pueden personalizar en JavaScript.

2

En sentencias condicionales, las siguientes comparaciones de igualdad estrictas darían como resultado el elemento incluido en cada comentario.

```
1 === 1            // es estrictamente igual, es decir, true
1 === "1"          // es false
null === undefined // es false
1 === true         // es false
"" === false       // es false
'\n' === false     // es false
"" === 0           // es false
'\n' === 0         // es false
```

011

4. Los comparadores de igualdad estrictos pueden comparar dos números, pares de datos booleanos, pares de datos de cadenas de texto, objetos... En todos estos casos, la igualdad está garantizada. [2]
5. En cuanto al operador de desigualdad estricta (`!==`), éste equivale a la negación de una comparación de igualdad estricta: [3]

```
caso1 !== caso2;
!(caso1 === caso2)
```

6. ¿Qué comportamiento tienen los operadores de igualdad normales? Estos comparadores (`==` y `!=`) realizan primero una conversión a un mismo tipo de dato para comprobar si ambos son iguales. [4] Veamos en qué consisten estas conversiones.
7. Si la sentencia contiene una cadena de texto y un número, el operador de igualdad convierte la cadena de texto en número para saber si ambos datos son iguales; esta comparación se realiza mediante un comparador de igualdad estricto.
8. Si la sentencia contiene un dato booleano y otro no booleano, el operador convierte el primero en un número y los compara de forma normal (`==` o `!=`).
9. En el caso en que la sentencia contenga un objeto y un número o una cadena de texto, el operador intentará convertir el objeto en un dato primitivo y realizará la comparación normal.
10. Es por este proceso de conversión que es altamente recomendable realizar las comparaciones mediante los operadores de igualdad estrictos.

IMPORTANTE

La conversión entre tipos de datos conlleva principalmente dos problemas: el primero de ellos es que las reglas de conversión en general pueden producir resultados inesperados y el segundo, que, debido a la indulgencia del operador, los errores de tipo pueden no resultar evidentes nunca.

3

```
caso1 !== caso2;
!(caso1 === caso2)
//Ambas sentencias son iguales
```

En el caso de la comparación `NaN == NaN`, el resultado es siempre false, puesto que el elemento `NaN` no es número y no se puede comparar.

4

```
1 == 1              // por lógica, es true
1 == "1"            // en este caso, se convierte
                       la cadena de texto a número
                       y da como resultado true
null == undefined   // true
1 == true           // true
"" == false         // true
'\n' == false       // true, el elemento '\n'
                       se interpreta como una cadena vacía
"" == 0             // true
'\n' == 0           // true
```

Operadores de comparación

LOS OPERADORES DE COMPARACIÓN COMPARAN SUS operandos, los cuales pueden ser números, cadenas de texto, lógicos u objetos, y devuelven como resultado un valor lógico según si la comparación establecida es o no verdadera. De hecho, todos los operadores de igualdad estudiados en el ejercicio anterior forman parte de este grupo de operadores. Además de los operadores de igualdad, los de comparación son los siguientes: > , >= , < y <=.

1. En este ejercicio, le mostraremos cómo se utilizan los operadores de comparación en JavaScript. Como se ha indicado en la introducción, son cuatro, además de los operadores de igualdad tratados en el ejercicio anterior, los operadores de comparación:

```
> (mayor que)
>= (mayor o igual que)
< (menor que)
<= (menor o igual que)
```

2. Los cuatro operadores de comparación funcionan tanto para números como para cadenas de texto. Sin embargo, cabe señalar que en el caso de las cadenas de texto, el uso de este tipo de operadores no es demasiado recomendable puesto que hacen distinción entre mayúsculas y minúsculas y no gestionan características como, por ejemplo, los acentos.

1

```
== // igual
!= // distinto
=== // estrictamente igual
!== // estrictamente distinto
> // mayor que
>= // mayor o igual que
< //menor que
<= //menor o igual que
```

2

```
caso2 > caso1
// la variable caso1 tiene el valor 3
   y la variable caso2, el valor 10
"120" > 111
```

012

3. El operador > devuelve el valor `true` siempre y cuando el operando izquierdo sea mayor que el derecho. Vea como ejemplo las siguientes sentencias: 2

```
caso2 > caso1
// la variable caso1 tiene el valor 3
   y la variable caso2, el valor 10
"120" > 111
```

4. El operador >= devuelve el valor `true` en el caso en que el operando izquierdo sea igual o mayor que el derecho. Un par de ejemplos del uso de estos operadores serían las siguientes sentencias: 3

```
caso2 >= caso1
// la variable caso1 tiene el valor 3
   y la variable caso2, el valor 10
caso1 >= 10
```

5. El siguiente operador de comparación es `<` que permite devolver el valor true si el operando izquierdo es menor que el derecho. Siguiendo con nuestros ejemplos con las variables `caso1` y `caso2`, mire estas sentencias: 4

```
caso1 < caso2
// la variable caso1 tiene el valor 3
   y la variable caso2, el valor 10
"120" < "111"
```

6. El último de los operadores de comparación que trataremos en este ejercicio es <=, que devuelve true en el caso en que el operando de la izquierda sea menor o igual que el de la derecha. Vea un claro ejemplo en las líneas siguientes: 5

```
caso1 <= caso2
// la variable caso1 tiene el valor
   y la variable caso2, el valor 10
caso2 <= 10
```

3

```
caso2 >= caso1
// la variable caso1 tiene el valor 3
   y la variable caso2, el valor 10
caso1 >= 10
```

4

```
caso1 < caso2
// la variable caso1 tiene el valor 3
   y la variable caso2, el valor 10
"120" < "111"
```

5

```
caso1 <= caso2
// la variable caso1 tiene el valor
   y la variable caso2, el valor 10
caso2 <= 10
```

El caso del operador +

IMPORTANTE

El operador + examina, ante todo, los operandos situados a izquierda y a derecha. Si uno de estos operandos es una cadena de texto o caracteres, entonces el otro también es convertido en cadena de texto y ambos, concatenados.

EL OPERADOR + SE COMPORTA DE maneras distintas según si los operandos con los que trabaja son números, textos u otro tipo de datos. Así, actuará sencillamente como un operador aritmético cuando se trata de números o bien servirá para concatenar elementos cuando trabaja con textos u otros datos.

1. Debido a sus comportamientos especiales en determinados momentos, hemos creído conveniente dedicar un ejercicio al operador + (más). Debemos tener en cuenta que JavaScript estudia y determina en cada momento con qué tipo de datos está tratando, y es por eso que puede decidir cómo debe comportarse, por ejemplo, el operador +.
2. El operador + realiza una suma cuando el programa interpreta que ambos operandos son números: 1

```
valor1 = 5
valor2 = 8
resultado =  10 + valor2;
```

3. En este caso, JavaScript interpreta que la variable resultado almacenará el valor resultante de la suma 10 + 8. En ocasiones, puede ocurrir que en lugar de interpretar los valores como numéricos, los interprete como cadenas de caracteres o como cadenas y números. Es en estos casos que el operador + actúa de concatenador, es decir, realiza combinaciones entre valores en lugar de sumarlos. 2

1

```
valor1 = 5
valor2 = 8
resultado =  10 + valor2;
//El resultado será 18.
```

2

```
<script type="text/javascript">
        var valor1 = '5';
        var valor2 = '8';
        document.write(valor1 + valor2);
// En este caso, el resultado sería 58,
puesto que JavaScript interpreta que se
desea combinar dos cadenas de texto.
</script>
```

En este caso, ni el uso de las comillas permite interpretar a JavaScript que se trata de números, sino que los lee como cadenas de caracteres.

4. El ejemplo anterior, el de la concatenación de textos, puede ser solventado mediante dos procedimientos: eliminando las comillas incluidas en la descripción de las variables o bien utilizando el método `Number()`. El primer procedimiento quedaría como sigue: 2

```
var valor1 = 5;
var valor2 = 8;
document.write(valor1 + valor2);
```

5. Y el resultado sería el esperado, es decir, 13. El empleo del método `Number()` resulta también del todo eficaz. ¿Cómo funciona este método? Muy sencillo. Permitiendo que el programa convierta en número cualquier cadena de caracteres. El modo en que se utiliza este método es el siguiente: 4

```
var valor1 = '5';
var valor2 = '8';
document.write(Number(valor1) + Number(valor2));
```

6. Como ve, en este caso es necesario recuperar las comillas en la definición de las variables para que el método `Number()` realice correctamente la conversión entre datos. El signo + realiza así la suma de los dos valores. Dicho esto podemos acabar diciendo que el operador + realiza una suma de valores cuando, al tratarse de operandos distintos, puede convertirlos a ambos en valores numéricos. En cualquier otro caso, el operador + realizará una concatenación de valores.

3

```
<script type="text/javascript">
        var valor1 = 5;
        var valor2 = 8;
        document.write(valor1 + valor2);
// En este caso, el resultado sería 13,
puesto que JavaScript interpreta que se
desea sumar dos cifras.
</script>
```

Para poder distinguir en su editor de textos web si está trabajando con valores numéricos o textuales, sepa que los números se muestran de color azul y los caracteres, de color rojo.

4

```
var valor1 = '5';
var valor2 = '8';
document.write(Number(valor1) + Number(valor2));
// El método Number() convierte texto en números.
// Ahora el resultado es 13, una suma de valores.
```

Operadores lógicos

LOS DATOS BOOLEANOS DISPONEN DE OPERADORES específicos que producen resultados de cada uno de estos tipos. Los operadores booleanos son && (AND) y || (OR) y el no lógico ! (NOT).

1. En este ejercicio trataremos los operadores lógicos y los numéricos. Los operadores lógicos son lo que producen valores booleanos y se asignan a operandos de este tipo. Se utilizan para realizar operaciones lógicas, es decir, aquellas que producen resultados verdaderos o falsos (`true` o `false`).
2. El primero de estos operadores, conocido como binario, es `&&`, es decir, AND. En una sentencia que confronte dos operandos, dicha operación producirá el primero de los operandos siempre y cuando este valor pueda convertirse en `false`; si no, dará como resultado el otro operando. Si los dos operandos son verdaderos, el valor producido es `true` y si no, producirá `false`.

```
if (numFav < 20 && numFav > 1) {
  document.write("<p>¡Perfecto!</p>");
}
```

3. Éste es un ejemplo en que se utiliza el operador lógico `&&` para comparar dos valores. Si el valor de la variable denominada `numFav` es menor que 20 y también mayor que 1, entonces la

1

```
if (x==2 && y!=3){
   */la variable x vale 2
   y la variable y es distinta de 3/*
}
```

En este ejemplo, se evalúan dos comprobaciones mediante un operador lógico. En la primera de estas comprobaciones, `x==2` dará como resultado `true` siempre y cuando la variable denominada `x` valga `2`. En la segunda comprobación, `y!=3` dará como resultado `true` si la variable denominada `y` tiene un valor distinto de `3`.

&&

2

```
if (numFav < 20 && numFav > 1) {
    document.write("<p>¡Perfecto!</p>");
}
```

operación producirá el valor `true` y, consecuentemente, aparecerá escrito en pantalla (`document.write`) el mensaje indicado. Pasemos ahora al segundo operador lógico que queremos tratar, el operador `||`, es decir, OR. En una sentencia que confronte dos operandos, el operador producirá como resultado `true` en el caso en que uno de los dos operandos sea verdadero; en el caso en que ambos valores sean falsos, entonces el resultado será `false`. 3

```
if (numFav==7 || numFav==9) {
 document.write("<p>¡El mío también!</p>");
}
```

4. El ejemplo sería idéntico al anterior aunque sólo debe cumplirse una de las dos condiciones mostradas en los dos operandos. Existe un tercer operador booleano que es no lógico y se representa con el signo `!`. Este operador sólo cuenta con un operando, el cual debe poder convertirse en verdadero para que el operador produzca un resultado `false`. Si no es así, el resultado que devolverá será true. Un ejemplo sencillo de este operador es el siguiente: 4

```
if (x==2 && y!=3)
```

5. En el cual como puede verse se utilizan dos de los operadores aquí mencionados: el operador `&&` y el operador negativo `!`. Es el segundo operando el que contiene este operador. En los ejercicios siguientes trataremos con todo detalle otros tipos de operadores de JavaScript: los numéricos y otros denominados especiales.

||

3

```
if (numFav==7 || numFav==9) {
    document.write("<p>¡El mío también!</p>");
}
*/ En este caso, si el valor de la variable numFav
es 7 o 9, una de las dos cifras, el operador devolverá
un valor verdadero y se imprimirá en pantalla el mensaje
indicado /*
```

4

```
if (x==2 && y!=3){
```

Operadores numéricos y especiales

LOS OPERADORES NUMÉRICOS O ARITMÉTICOS, IGUAL que ocurre con los booleanos, trabajan con operandos del tipo Number y, por tanto, producen resultados sólo de este tipo. Los operadores numéricos son los aritméticos en todas sus combinaciones (`+ - * / % ++ --`). Por su parte, existen tres tipos de operadores conocidos como especiales: el operador condicional (`?:`) y el operador coma (`,`).

1. En un ejercicio anterior tuvimos un primer contacto con los operadores aritméticos básicos (`+ - * /`). En este ejercicio trataremos otro tipo de operador aritmético o numérico: se trata de la combinación `++` y `--`, que puede situarse delante o detrás de un valor numérico incluido en una variable.
2. La posición de cada uno de estos operandos puede ayudarle a recordar si el resultado devuelto se situará antes o después de la suma (`++`) o la resta (`--`). En el caso en que el operando esté situado detrás del operador `++`, el resultado va antes del incremento. Y viceversa.
3. Veamos esta descripción en ejemplos. Los operadores `++` y `--` situados delante del nombre de la variable devuelven el valor actual de dicha variable más o menos 1. Estas líneas son una muestra de este uso:

```
puntos++;
```

1

```
function a uestion(pregunta) {
 var resp ta = prompt(pregunta[0],'');
  if (resp sta == pregunta[1]) {
   alert(' orrecto!');
   puntos++;
  } else {
   alert('Lo sentimos. La respuesta c ecta es ' + pregunta[1]);
  }
}
function cuest() {
 for (var i=0; i<preguntas.length; i++) {
  askQuestion(aleat);
  aleatorio = Math.floor(Math.random() *preguntas.length);
  aleat = preguntas[aleatorio];
 }
```

En estas dos funciones puede ver que se utilizan en determinados puntos de las distintas sentencias diversos operadores aritméticos, entre los cuales se encuentra el operador `++` detrás del nombre de la variable `puntos`.

2

```
caso1 = 15;
++caso1
16
// ++ suma 1 al valor 15

caso1 = 15;
--caso1
14
// -- resta 1 al valor 15
```

Estos ejemplos son básicos para ayudar a comprender con sencillez el uso de estos operadores.

```
caso1 = 15;
++caso1
16
```

4. Y también:

```
caso1 = 15;
--caso1
14
```

5. Como hemos indicado, la posición de estos operadores puede cambiar respecto a la variable que acompañan. En este caso, el incremento o el decremento se realiza antes y después se devuelve el resultado. Veamos un ejemplo básico de ello:

```
caso1 = 15;
caso1++
15
caso1 = 16
```

6. Pasemos ahora a tratar brevemente otro tipo de operadores, que se consideran especiales debido a su uso concreto en determinados scripts. El operador condicional `?:` sirve para realizar expresiones condicionales, sin tener en cuenta el grado de complejidad de las mismas. Si la condición escrita es verdadera, dará como resultados el valor `if_true`; de otro modo, si la condición no es cierta, el resultado será `if_false`.

7. El operador , (coma) es para las expresiones lo que el punto y coma es para las sentencia. Su procedimiento consiste en evaluar los dos operandos a los que acompaña y devuelve como resultado el valor `right`.

015

IMPORTANTE

Más adelante en este libro volveremos a dedicarnos a estos tipos de operadores para profundizar mejor en su uso y funcionamiento en JavaScript.

3

```
caso1 = 15;
caso1++
15
caso1 = 16
```

```
caso1 = 15;
caso1--
15
caso1 = 14
```

5

```
var caso1 = 0;
var caso2 = (caso1++, 10);

// Lo que significa que:

caso1 = 1
caso2 = 10
```

Debido a que puede llevar a confusión en el caso de scripts muy largos, es recomendable no utilizar el operador , y optar por la separación habitual de sentencias, es decir, con un ;

4

```
var ahora = new Date();
var saludo = "Buenas" + ((ahora.getHours() > 17) ? " tardes." : " dia.");
*/ La variable ahora contiene la hora actual.
   Si ésta pasa de las cinco de la tarde
   se generará el mensaje "Buenas tardes"/*
```

El operador condicional `?:` se puede utilizar como forma abreviada de una sentencia condicional del tipo `if ... else`.

Comprobar datos en JavaScript

IMPORTANTE

En muchas ocasiones, resulta recomendable utilizar otras funciones para cambiar los tipos de datos de una variable antes de utilizar el operador `typeof`. Una de estas funciones es `parseInt()`.

COMO HEMOS PODIDO COMPROBAR HASTA EL momento, el uso de operadores concretos y de operaciones realizadas depende del tipo de datos con el que se asocien dichos operadores. La comprobación de datos se puede llevar a cabo con un operador especial: el operador `typeof`, el cual nos envía un texto que contiene el tipo de datos objeto de nuestra comprobación.

1. Debido a su utilidad, hemos creído conveniente dedicar un ejercicio completo al operador `typeof`, aunque en una lección anterior (008) ya tuvimos la ocasión de nombrarlo y describirlo brevemente. 1
2. `Typeof` es un operador que devuelve una cadena o string que describe el tipo de dato que corresponde con el objeto, ya sea una variable, una función..., que se escribe a continuación. Más concretamente, el operador `typeof` distingue los valores primitivos de los objetos y, en el caso de las primitivos, determina el tipo al que pertenecen: "number", "string", "boolean", "object", "function" y "undefined". 2
3. En el caso de los datos booleanos, como recordará, los valores que se manejan son `true` y `false`.

1

```
if (typeof(caso1) === "undefined") {
  alert("caso1 no está definido.");
}
```

Tal y como se trató en un ejercicio anterior, el uso de typeof para categorizar si un dato es un `undefined` o `null` es una de las soluciones para llevar a cabo esta comprobación.

2

```
var declarada;
var resultado = (declarada === undefined);
// true

var resultado = (typeof declarada === 'undefined');
// true

var resultado = (typeof noDeclarada === 'undefined')
// true
```

016

4. ¿Cómo se utiliza en un script el operador `typeof`? A continuación le mostramos un ejemplo para cada uno de los tipos de datos más básicos: [4]

```
var booleano = false;
alert("Tipo de dato: " + typeof booleano)

var numero = 107;
alert("Tipo de dato: " + typeof numero)

var caracteres = "Barcelona";
alert("Tipo de dato: " + typeof caracteres)

var hoy = newDate();
alert("Tipo de dato: " + typeof hoy)
```

5. Los resultados que devolverá el operador `typeof` para cada uno de estos tipos será `Boolean`, `Number`, `String` y `Object` (`Object` corresponde a una función, elemento distinto a los datos primitivos).
6. Los paréntesis en el uso del operador `typeof` son totalmente opcionales; aplíquelos cuando crea que pueden ayudar a la comprensión y la lectura del script. [3]
7. A modo informativo diremos que los números, tanto enteros como decimales, siempre son del tipo `Number`.
8. Además de distinguir entre datos primitivos y objetos, `typeof` también permite comprobar métodos (`parseInt()`), valores de propiedades (`window.length` o `document.lastModified`) y objetos predefinidos (`Math` u `Object`). [5]

3

```
var booleano = false;
alert("Tipo de dato: " + typeof booleano)
//Typeof devolverá el valor Boolean

var numero = 107;
alert("Tipo de dato: " + typeof numero)
// Typeoff devolverá el valor Number

var caracteres = "Barcelona";
alert("Tipo de dato: " + typeof caracteres)
//Typeof devolverá el valor String

var hoy = newDate();
alert("Tipo de dato: " + typeof hoy)
// Typeof devolverá el valor Object
```

4

```
typeof caracteres

// es lo mismo que:

typeof (caracteres)
```

5

```
typeof document.lastModified
// El valor devuelto será 'string'

typeof window.length
// El valor devuelto será 'number'

typeof eval
// El valor devuelto será 'function'

typeof parseInt
// El valor devuelto será 'function'

typeof Math
// El valor devuelto será 'object'
```

El operador 'instanceof'

ASÍ COMO EL OPERADOR `typeof` SE utiliza en JavaScript para distinguir entre los diferentes datos primitivos y los objetos, existe otro operador que permite comprobar si un objeto es una instancia o un constructor (de los constructores hablaremos con todo detalle más adelante en este libro): se trata del operador `instanceof`.

1. En este ejercicio describiremos el operador `instanceof`, el cual, como ya hemos adelantado en la introducción de este ejercicio, se utiliza para saber si un objeto es una instancia o un constructor. Por si todavía no sabe qué es un constructor diremos que son funciones con nombre que ayudan en la producción de objetos que son parecidos en algún aspecto. Los constructores serán tratados con todo detalle en un ejercicio posterior de este libro.
2. El uso del operador `instanceof` se recomienda en aquellos casos en que se desea comprobar el tipo de objeto con que se está trabajando en tiempo de ejecución.
3. El modo en que se utiliza este operador en la sintaxis de JavaScript necesita el nombre del objeto cuyo tipo de desea comparar y el tipo de objeto en cuestión. Lo que debe tener en cuenta es que como operando derecho, es decir, como tipo de objeto, debe declarar un objeto, no su descripción. Con ello queremos decir que, por ejemplo, en un objeto tipo String, deberá indicar la palabra 'String' en lugar de la cadena de texto correspondiente.

1

```
function constructores() {
var caso1 = new Coche('Diesel');
var caso2 = new Coche('Gasolina');
alert ('El vehículo 1 soporta ' + caso1.getlitros() + ' litros\n');
alert ('El vehículo 2 soporta ' + caso2.getlitros() + ' litros\n');
}
```

Ejemplo de constructor en JavaScript.

2

```
nombre instanceof tipo
```

4. A continuación, mostramos unos ejemplos de script en los cuales se utiliza el operador `instanceof`. Con el primero de ellos podremos comprobar si un dato es un objeto del tipo Date: 3

```
hoy = new Date(2015, 02, 29)
if (hoy instanceof Date) {
   document.write('¡Hoy es tu día!');
}
```

5. `hoy` es el nombre del objeto que se desea comprobar y `Date`, el tipo de objeto. Si el resultado es `true`, es decir, si `hoy` es un objeto `Date`, se ejecutará la función indicada a continuación (`document.write`).

6. Veamos un segundo ejemplo del uso del operador `instanceof`. En esta ocasión, lo utilizaremos para comprobar si dos objetos del tipo String y Number son del tipo Object: 4

```
texto = new String()
cifra = new Number()
texto instanceof String == true
texto instanceof Object == true
texto instanceof Date == false
cifra instanceof Date  == true
cifra instanceof Object == true
cifra instanceof String  == false
```

7. Como puede ver, los valores que devuelve el operador `instanceof` en este caso son valores booleanos del tipo `true` y `false`. Otro caso en el cual se podría utilizar el operador que estamos tratando es la comprobación de que un tipo de objeto creado por nosotros es de un tipo concreto y del tipo Object. 5

3

```
hoy = new Date(2015, 02, 29)
if (hoy instanceof Date) {
   document.write('¡Hoy es tu día!');
}
// Si la variable hoy es un objeto del tipo Da
se imprimirá en pantalla el mensaje indicado.
```

4

```
texto = new String()
cifra = new Number()
texto instanceof String == true
texto instanceof Object == true
texto instanceof Date == false
cifra instanceof Date  == true
cifra instanceof Object == true
cifra instanceof String  == false
```

5

```
function Prenda(tipo, tejido, color) {
   this.tipo = tipo
   this.tejido = tejido
   this.color= color
}
miPrenda = new Prenda("Falda", "Lana", "Azul")
a = miPrenda instanceof Prenda == true
b = miPrenda instanceof Object == true
```

El valor situado a la derecha del operador == es el que devolverá el operador `instanceof` al evaluar los tipos de datos indicados en cada caso.

Operadores de objeto

AUNQUE SERÁN TRATADOS POR SEPARADO Y con todo detalle más adelante en este libro, en este ejercicio introduciremos los denominados operadores de objeto. Como su nombre indica, se trata de una serie de operadores que trabajan únicamente con objetos, nunca con otro tipo de datos.

1. Son tres los operadores que trabajan sobre objetos:

   `new`: este operador crea una instancia de un objeto.

   `delete`: este operador elimina propiedades de un objeto.

   `in`: este operador comprueba que un objeto disponga de una propiedad ya definida.

2. Veamos uno a uno en qué consisten estos operadores de objeto y cómo se utilizan. El operador `new` se utiliza para crear una instancia de un objeto, tanto si éste ha sido generado de forma personalizada o bien si es de uno de los tipos predefinidos: Array, Boolean, Date, Function, Number, etc. 1

3. El operador `new` va siempre seguido del nombre de una función, la cual se utiliza para inicializar el nuevo objeto creado. Los parámetros que se necesitan son, por un lado el constructor, es decir, una función que especifica el tipo de instancia del objeto y por otro, los argumentos, es decir, una lista de valores con los que el constructor será llamado: 2

new Date()

1

```
<p><script type="text/javascript">
 var hoy = new Date();
 var textoFecha = fechaCompleta(hoy, '#DIASEMANA#, #DIA# de #MES# de #YEAR#');
 document.write(textoFecha);
</script></p>
<p><script type="text/javascript">
var fechaActual = new Date();
var fechaFutura = new Date(2015,2,01);
var diferencia = fechaFutura.getTime() - fechaActual.getTime();
var totalDias = Math.round(diferencia/(1000 * 60 * 60 * 24));
document.write("¡Faltan " + totalDias + " días para la inauguración!");
</script></p>
```

2

```
new constructor[([argumentos])]
```

```
function Prenda(tipo, tejido, color) {
  this.tipo = tipo;
  this.tejido = tejido;
  this.color= color;
}
var prenda1 = new Prenda("Falda", "Lana", "Azul");
```

4. El segundo de los operadores de objeto que deseamos describir en este ejercicio es `delete`, con el cual es posible eliminar propiedades de un objeto, así como suprimir uno o varios elementos de una matriz (*array*). A continuación puede consultar un ejemplo que ilustra esta segunda opción, la de eliminar elementos de una lista:

```
var lugares = ['Italia', 'Francia', 'Dinamarca', 'Reino Unido', 'Alemania', 'Rusia', 'Grecia', 'China' ];
delete lugares[3];
```

5. Si tenemos en cuenta que el primer elemento de una matriz tiene asignado el valor 0, en este caso se suprimirá de la lista el elemento Reino Unido.

6. Por último, el operador de objeto `in` devuelve un valor verdadero (true) siempre y cuando la propiedad especifica en la sentencia se encuentre en el objeto indicado. Los parámetros que intervienen en el uso de este operador son `prop`, que es una cadena o expresión numérica que representa una propiedad o el índice de una matriz, y `objectName`, que contiene el nombre de un objeto. En la imagen 5 puede ver un ejemplo de código en que se emplea el operador de objeto `in`.

3

```
function Prenda(tipo, tejido, color) {
   this.tipo = tipo;
   this.tejido = tejido;
   this.color= color;
}
var prenda1 = new Prenda("Falda", "Lana", "Azul");
var prenda2 = new Prenda("Jersey", "Angora", "Rosa");
var prenda3 = new Prenda("Pantalón", "Tejano", "Gris");
//Podemos crear tantas variables del objeto Prenda
como necesitemos
```

4

```
var lugares = ['Italia', 'Francia', 'Dinamarca', 'Reino Unido',
'Alemania', 'Rusia', 'Grecia', 'China' ];
delete lugares[3];
document.write ("element 3: " + lugares[3]);
document.write ("<br />");
document.write ("array: " + lugares);
```

```
"tejido" in Prenda
```

5

```
var Prenda {tipo: "Jersey", tejido: "Angora", color: "Rosa"};
"tipo" in Prenda == true
"tejido" in Prenda == true
```

Conocer los tipos de datos booleanos

BÁSICAMENTE, LOS TIPOS DE DATOS BOOLEANOS son true y false. En este ejercicio vamos a tratar con todo detalle este tipo de datos, que, por otro lado, ya hemos mencionado en más de una ocasión en ejercicios anteriores.

1. En JavaScript, las variables booleanas son aquellas que pueden almacenar información dual, es decir, del tipo Sí/No, Verdadero/Falso, Cumple/No cumple, etc. A este tipo de variable siempre se le asigna como contenido `true` o `false`.
2. Debido a que `true` y `false` están consideradas como palabras clave en JavaScript, no está permitido utilizarlas como nombre para nuestras variables. Si en alguna ocasión, por despiste, nombra una variable `false`, por ejemplo, el navegador interpretará que hay un error en el código y simplemente no ejecutará la función definida. 1
3. Nunca sitúe entre comillas los valores true y false; si lo hace, el tipo de dato se convertirá en texto ("String") 2 y dejará de ser booleano: 2

1
```
var true = new Prenda("Falda", "Lana", "Azul");
var false = new Prenda("Jersey", "Angora", "Rosa");
var false = new Prenda("Pantalón", "Tejano", "Gris");
//Nunca deben utilizarse los nombres true y false
para designar una variable.
```

Tenga en cuenta que si nombra a una variable como `true` o `false`, el intérprete de JavaScript no lanzará ningún mensaje de error; únicamente no se ejecutará la instrucción detallada.

2
```
var caso1 = "true"; typeof caso1 == "string"
var caso1 = "false"; typeof caso1 == "string"
//Si podemos entre comillas los valores true y false
tendremos una cadena de caracteres en lugar de
un valor booleano.
```

3
```
var caso1 = true; typeof caso1 == "boolean"
var caso1 = false; typeof caso1 == "boolean"
// Utilizamos el operador typeof para conocer
 el tipo de dato definido en la variable caso1.
```

```
var caso1 = true; typeof caso1 == "boolean"
var caso1 = false; typeof b == "boolean"
```

4. Las instrucciones `if/else` son las más adecuadas para comprobar si se cumplen ciertas condiciones y, para mostrar los resultados de estas comprobaciones se suelen utilizar datos de tipo booleanos:

```
var hora = new Date();
var horas = hora.getHours();
 if (horas<12) {
  formatoHora = 'am';
 } else {
  formatoHora = 'pm';
 }
```

5. En este código se utilizará `true` o `false` para decidir si el formato de la hora debe mostrarse con 'am' o con 'pm'. De ello dependerá si la hora es anterior a las 12 (`true`, entonces 'am') o después (`false`, entonces 'pm').

6. ¿Cuándo una variable devuelve `true` y cuándo lo hace `false`? A grandes rasgos, podemos decir que devuelve `true` cuando recibe un valor entre comillas o bien cualquier número que no sea 0. Y devuelve `false` cuando se omite un valor al constructor o bien si este valor es una cadena vacía, el valor 0 o, como hemos indicado anteriormente, la palabra `false` sin comillas.

4

```
function askQuestion(pregunta) {
 var respuesta = prompt(pregunta[0],'');
  if (respuesta == pregunta[1]) {
   alert('¡Correcto!');
   puntos++;
  } else {
   alert('Lo sentimos. La respuesta correcta es ' + pregunta[
  }
}
```

5

```
var hora = new Date();
var horas = hora.getHours();
 if (horas<12) {
  formatoHora = 'am';
 } else {
  formatoHora = 'pm';
 }
```

6

```
var caso1 = new Boolean("8")
document.write(caso1 + "<br>")

var caso2 = new Boolean(8)
document.write(caso2 + "<br>")

//En ambos casos, se devuelve true
```

7

```
var caso1 = new Boolean(false)
document.write(caso1 + "<br>")

var caso2 = new Boolean(0)
document.write(caso2 + "<br>")
//En ambos casos, se devuelve false
```

Convertir valores en booleanos

CADA UNO DE LOS VALORES DE JavaScript (números, booleanos, texto, objetos y los valores undefined y null) pueden ser convertidos a booleanos. Como se verá en este ejercicio, a cada uno de estos valores le corresponde un valor booleano concreto.

1. Para poder convertir valores de JavaScript en datos booleanos, utilizaremos el objeto 'Boolean' como un objeto envolvente. Si no recuerda qué son los objetos envolventes, no dude en recuperar el ejercicio 9 de este libro.
2. El único parámetro que precisa es proceso de conversión de este tipo de datos es el de valor. Se tratará en este caso del valor inicial del objeto booleano. Si no existe ningún valor o éste es 0, -0, null, false, NaN, undefined o una cadena vacía, el valor inicial del objeto será `false`. El resto de valores crean un objeto con el valor inicial `true`. La tabla incluida en la imagen 1 indica el valor inicial y qué valor booleano le corresponde tras su conversión.
3. Cualquier objeto cuyo valor no sea undefined o null, incluyendo un objeto Boolean cuyo valor sea false, da como resultado true cuando se pasa a una sentencia condicional. ¿Qué significa esta afirmación? Lo entenderá fácilmente en el siguiente ejemplo:

```
x = new Boolean(false);
if (x)
//En este caso, el resultado es true.
```

1

undefined	false
null	false
dato booleano	no se puede convertir
Número 0 o NaN	false
Cualquier otro número	true
Cadena de caracteres entre comillas	false
Otra cadena de caracteres	true
Objeto	Siempre true

2

```
x = new Boolean(false);
if (x)
//En este caso, el resultado es true.
```

020

4. JavaScript dispone de tres métodos para convertir manualmente datos a booleanos. El primero de estos métodos, y el más defendido por los desarrolladores, pasa por invocar el objeto Boolean como una función y no como un constructor: `Boolean(valor)`. Un ejemplo de sentencias que siguen este método sería el siguiente: 3

```
Boolean(undefined) == false
Boolean(null) == false
Boolean(0) == false
Boolean(1) == true
Boolean(2) == true
Boolean('') == false
Boolean('abc') == true
```

5. El segundo método para convertir manualmente valores a booleanos es utilizando el operador condicional, incluido dentro del grupo de los operadores especiales de JavaScript: `? :`. Como recordará, este operador sirve para realizar expresiones condicionales, sin tener en cuenta el grado de complejidad de las mismas. Si la condición escrita es verdadera, dará como resultado el valor `true`; de otro modo, si la condición no es cierta, el resultado será `false`. 4

6. El tercer método que permite aplicar JavaScript es utilizar una doble negación delante del valor que deseamos convertir. Esta doble negación se consigue con los símbolos `!!`.

7. No queremos terminar sin advertir que no se deben confundir los datos primitivos booleanos `true` y `false` con los valores `true` y `false` del objeto Boolean. Del mismo modo, no debe utilizar un objeto Boolean en lugar de una primitiva booleano (`true` o `false`).

3

```
Boolean(undefined) // devuelve false
Boolean(null) // devuelve false
Boolean(0) // devuelve false
Boolean(1) // devuelve true
Boolean(2) // devuelve true
Boolean('') // devuelve false
Boolean('abc') // devuelve true
```

4

```
var ahora = new Date();
var saludo = "Buenas" + ((ahora.getHours() > 17) ? " tardes." : " dia.");
*/ La variable ahora contiene la hora actual.
   Si ésta pasa de las cinco de la tarde
   se generará el mensaje "Buenas tardes"/*
```

Los operadores || y !

EN EL EJERCICIO 14 DE ESTE libro tratamos de forma breve y descriptiva los tres operadores lógicos de JavaScript: && (AND), || (OR) y ! (NOT). En este ejercicio vamos a profundizar en dos de ellos para mostrar el modo en que pueden comportarse en sus scripts.

1. Empezaremos este ejercicio hablando del operador lógico `||` (OR) y recordando su descripción: se utiliza, como el resto de operadores lógicos, para tomar decisiones en un script y se basan en los operandos booleanos `true` y `false`. En una sentencia que confronte dos operandos, el operador `||` producirá como resultado `true` en el caso en que uno de los dos operandos sea verdadero; en el caso en que ambos valores sean falsos, entonces el resultado será `false`.
2. Otro modo de gestionar el operador OR es el siguiente: Si el primer operando puede convertirse en `true`, la comparación devuelve `true`. De no ser así, el resultado será el segundo operando.
3. En ocasiones, una sentencia puede tener como valor (ya sea un parámetro o el resultado de una función) un valor nulo (undefined o null) o un valor real. Si deseamos proporcionar

1

```
if (numFav==7 || numFav==9) {
    document.write("<p>¡El mío también!</p>");
}
*/ En este caso, si el valor de la variable numFav
es 7 o 9, una de las dos cifras, el operador devolverá
un valor verdadero y se imprimirá en pantalla el mensaje
indicado /*
```

2

```
true || false // devuelve true
true || 'def' // devuelve true
'lunes' || 'def' // devuelve lunes
'' || 'def' // devuelve 'def'
```

un valor predeterminado que sustituya a estos valores, es posible utilizar el operador ||. Un ejemplo resumido de uno de estos casos es el siguiente: 3

```
function guardarPrenda(prenda) {
 prenda = prenda || ''; ...
}
```

4. De hecho, éste es el uso más habitual de este operador lógico. Otro caso en que puede utilizarse el operador || como valor predeterminado es para una propiedad. Imagine que en una sentencia falta una propiedad de un objeto determinado: 4

```
setJuego(options.nombre || 'Sin nombre');
```

5. El operador || se ocupa todavía de un tercer caso en aquellas situaciones en que falta un valor. Este operador puede utilizarse como un valor predeterminado en el caso de pérdida del resultado de una función. Vea en la imagen 5 un ejemplo resumido de código en que se utiliza el operador OR como valor predeterminado por un resultado de una función. 5

6. Hablemos ahora del operador NOT (!), el cual, como recordará, convierte su operando en booleano y después lo niega. Estos son algunos ejemplos gráficos de lo que hace este operador no lógico:

```
!true
// devuelve false
!43
// devuelve false
!''
//devuelve true
!{}
//devuelve false
```

3

```
function guardarPrenda(prenda) {
 prenda = prenda || ''; ...
}
*/ El parámetro que aquí hemos denominado prenda
es opcional y puede ser sustituido por una cadena
de caracteres vacía.
```

4

```
setJuego(options.nombre || 'Sin nombre');
// La propiedad nombre del objeto options
es opcional. En el caso en que no aparezca,
se podrá utilizar el valor 'Sin nombre'.
```

5

```
function contar(regex, str) {
 return (str.match(regex) || []).length;
}
```

Trabajar con números

LOS NÚMEROS SON UNA PARTE IMPORTANTE de la programación puesto que permiten gestionar tareas tales como calcular costes o distancias entre dos puntos. JavaScript ofrece diferentes maneras de trabajar con ellos, además de muchas particularidades a tener en cuenta.

1. En cuanto a números se refiere, JavaScript cuenta con la particularidad de que, aunque los programadores no lo vean, trata todos los números del mismo modo: como si tuvieran un punto decimal. Sin embargo, si no existe ningún valor detrás de este punto, dicho signo no se visualiza. 1
2. La mayoría de los intérpretes de JavaScript distinguen y optimizan entre decimales y números enteros. Más adelante en este libro se tratarán más detalladamente cada uno de estos tipos de números.
3. Los números en JavaScript son de doble valor, o, lo que es lo mismo, son de 64 bits, y se basan en el estándar que utilizan la mayoría de los lenguajes de programación: el IEEE Standard for Floating-Point Arithmetic (IEEE 754). Si desea echar un vistazo a este largo documento, no dude en acceder a la siguiente dirección web: **http://www.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF**. 2

1

```
3.000
// Éste valor es 3 en JavaScript
```

Tenga en cuenta que este lenguaje de programación no utiliza la coma decimal, sino el punto.

2

Lecture Notes on the Status of
IEEE Standard 754 for Binary Floating-Point Arithmetic

Prof. W. Kahan
Elect. Eng. & Computer Science
University of California
Berkeley CA 94720-1776

Introduction:

022

4. JavaScript trabaja con diferentes tipos de números. Estos tipos pueden ser enteros, decimales o hexadecimales. Veamos un ejemplo de cada uno de ellos:

```
número entero == 275
número decimal == 14.528
número hexadecimal == 0xFF
```

5. Los números hexadecimales tienen su correspondencia en números enteros, que son en los que se basa JavaScript para trabajar: 3

```
número hexadecimal == 0xFF == 255
```

6. Además de estos tipos, JavaScript trabaja con números exponenciales. Estos se representan en el código como `eX`, una abreviatura que significa "Multiplica por 10^x": 4

```
7e2 == 700
7e-2 == 0.07
0.7e2 == 70
```

7. Al trabajar con números, y más concretamente, con números, métodos y propiedades, es preciso distinguir claramente el punto que da acceso a la propiedad del punto decimal. Así, supongamos que desea llamar la propiedad `toString()` para el número 238. Según como desee tratar este número, el punto que separará el valor de la propiedad será insertado de distintos modos: 5

```
238..toString()
238 .toString()
// En este caso, es preciso dejar un espacio entre el punto y el número
238.0.toString()
(238).toString()
```

3

```
número entero == 275
número decimal == 14.528
número hexadecimal == 0xFF
// número hexadecimal == 0xFF == 255
```

4

```
7e2 == 700
7e-2 == 0.07
0.7e2 == 70
//Tenga en cuenta que el operador ==
lo hemos situado para generar el resultado
de las multiplicaciones con el exponencial
```

5

```
238..toString()
238 .toString()
// En este caso, es preciso dejar
un espacio entre el punto y el número
238.0.toString()
(238).toString()
```

Convertir datos a números

IGUAL QUE JAVASCRIPT CUENTA CON DISTINTOS métodos para convertir valores en booleanos, tal y como vimos en un ejercicio anterior, este lenguaje de programación también dispone de procesos de conversión de datos a números.

1. En este ejercicio veremos, por un lado, en qué tipo de valor numérico pueden convertirse distintos datos de JavaScript, y por otro, cuál es la mejor forma para realizar manualmente esta conversión. Para empezar, consulte la tabla incluida en la imagen 1 para comprobar en qué valor numérico puede convertirse cada tipo de dato.
2. Como ha podido ver en la tabla, las cadenas de caracteres vacías se convierten en 0; sin embargo, debe saber que ésta no es la única posibilidad, puesto que la opción `NaN` (Not a Number) también es una buena solución.
3. Como hemos indicado anteriormente, existen distintas formas de convertir manualmente prácticamente cualquier tipo de dato de JavaScript en un valor numérico. La mejor de las opciones es el método `Number()`, invocado como función y no como constructor, puesto que su uso es mucho más descripti-

1

undefined	`NaN`
null	`0`
dato booleano	Si es `false` == 0 Si es `true` == 1
Cualquier número	No hay conversión
Cadena de caracteres	El mismo número pero sin comillas. En el caso de una cadena vacía, el número será `0`
Objeto	Utilizando el método `ToPrimitive(value, Number)`, convierte la primitiva obtenida

En el caso de las cadenas de caracteres (String), normalmente aparecerá el valor numérico original entre comillas, `'25.894'`; el resultado será `25.894`.

2

```
Number(null)//El resultado sería 0
5 + null // El resultado sería 5


Number(undefined) // El resultado sería NaN (Not a Number)
5 + undefined // El resultado sería NaN
```

vo. Éste sería un ejemplo del método `Number()` para convertir una cadena de caracteres en un valor numérico: 3

```
var caso1 = new Number("casa")
document.write(caso1 + "<br>")
//El resultado de la conversión es NaN
```

4. El valor `"casa"` es un texto, lo que supone que no se puede convertir en ninguna cifra en concreto, sino en el elemento `NaN`. Veamos otros ejemplos: 4

```
var caso1 = new Number()
document.write(caso1 + "<br>")
//El resultado de la conversión es 0
```

5. Aquí no existe ningún valor definido para el método `Number()`, por lo que se convierte en un 0.

```
var caso1 = new Number("9876")
document.write(caso1 + "<br>")
//El resultado de la conversión es 9876

var caso2 = new Number("1258P")
document.write(caso1 + "<br>")
//El resultado de la conversión es NaN
```

6. Vea la diferencia entre estos dos valores numéricos. En el primer caso, el resultado es exactamente el mismo que el que se pretende convertir, puesto que se trata de un número. 5 Sin embargo, el caso de la segunda variable da como resultado de la conversión de nuevo el elemento `NaN`. La razón es la letra P insertada como quinto caracter en la cadena, la cual provoca que el número inicial deje de comportarse como tal. 6

023

IMPORTANTE

En el próximo ejercicio trataremos otras formas de convertir datos de JavaScript en números, formas distintas al método `Number()`, por otro lado el más recomendable para llevar a cabo conversiones de este tipo.

3

```
var caso1 = new Number("casa")
document.write(caso1 + "<br>")
//El resultado de la conversión es NaN
```

Como el valor entre paréntesis es una cadena de texto, JavaScript realiza la conversión en el elemento `NaN`, que, como ya sabe, significa Not a Number.

4

```
var caso1 = new Number()
document.write(caso1 + "<br>")
//El resultado de la conversión es 0
```

En este caso, recuerde que el resultado de la conversión sería el mismo si hubiera comillas dentro del paréntesis (`''`).

5

```
var caso1 = new Number("9876")
document.write(caso1 + "<br>")
//El resultado de la conversión es 9876
```

6

```
var caso2 = new Number("1258P")
document.write(caso1 + "<br>")
//El resultado de la conversión es NaN
```

La función parseFloat()

PARSEFLOAT() ES UNA FUNCIÓN GLOBAL QUE permite convertir cadenas de texto (String) en valores numéricos. Aunque la función más recomendable para llevar a cabo esta operación de conversión es Number(), descrita en el ejercicio anterior, creemos conveniente dedicar un ejercicio a esta otra conocida función.

1. En este ejercicio conocerá con todo detalle cómo se utiliza la función `parseFloat()` para convertir cadenas de texto en valores numéricos. Básicamente, esta función convierte su argumento, es decir, la cadena de texto, e intenta devolver un número.
2. Como recordará, JavaScript trata todos los números como si fueran decimales, como si llevaran un punto decimal. Si la función encuentra un caracter distinto a los signos + y -, un número del 0 al 9, un punto decimal o un número exponencial, el resultado de la conversión será sólo el valor que aparece antes del punto. En el caso en que sea imposible convertir en número el primer caracter de la cadena, entonces el resultado de la operación será `NaN`
3. En la imagen 2 le mostramos unos ejemplos en los cuales la función `parseFloat()` convierte diferentes argumentos en un mismo número: 8.57.

1

```
parseFloat("FF2");
// En este caso, el resultado
de la conversión es NaN puesto
que se trata de un valor numérico
hexadecimal.
```

2

```
parseFloat("8.57");
parseFloat("857e-2");
parseFloat("0.0857E+2"); v
var caso1 = "8.57"; parseFloat(caso1);
parseFloat("8.57abcgdft");
```

024

4. Realmente, usted puede utilizar la función que prefiere cuando necesite realizar las conversiones numéricas, `parseFloat()` o `Number()`. Sin embargo, debe tener en cuenta que el resultado de estas conversiones puede ser distinto si se realiza con un método o con otro. Veamos algunas diferencias entre ambos.

5. Algunos de los valores que `Number()` convierte en un valor numérico, `parseFloat()` lo hace en el elemento `NaN`. Por ejemplo, con ciertos valores booleanos: 3

```
parseFloat(false) // el resultado es NaN
Number(false) // el resultado es 0
parseFloat(true) // el resultado es NaN
Number(true) // el resultado es 1
```

6. También con el valor `null` el resultado es distinto: 4

```
parseFloat(null) // el resultado es NaN
Number(null) // el resultado es 0
```

7. `parseFloat()` convierte las cadenas vacías en `NaN`, mientras que `Number()` lo hace en 0. Por su parte, en valores numéricos combinados con otros dígitos, `parseFloat()` da como resultado de la conversión el número indicado en el argumento hasta el dígito no numérico; `Number()` considera que esta combinación de dígitos no es un número y, por tanto, el resultado de la conversión es `NaN`.

```
parseFloat(' ') // el resultado es NaN
Number(' ') // el resultado es 0
parseFloat('2.548@') // el resultado es 2.548
Number('2.548@') // el resultado es NaN
```

3

```
parseFloat(false) // el resultado es NaN
Number(false) // el resultado es 0
parseFloat(true) // el resultado es NaN
Number(true) // el resultado es 1
```

El resultado de la función `parseFloat()` será el mismo si el argumento se escribe entre comillas (`'true'`).

5

```
parseFloat(' ') // el resultado es NaN
Number(' ') // el resultado es 0
parseFloat('2.548@') // el resultado es 2.548
Number('2.548@') // el resultado es NaN
```

4

```
parseFloat(null) // el resultado es NaN
Number(null) // el resultado es 0
```

Las funciones parseInt() e isNaN()

LA FUNCIÓN PARSEINT(), EN COLABORACIÓN CON la función `isNaN()`, permite validar si la introducción de datos requerida es numérica. De hecho, `parseInt()` actúa de forma muy parecida a la función descrita en el ejercicio anterior, `parseFloat()`.

1. En este ejercicio describiremos con todo detalles el uso y funcionamiento de las funciones `parseInt()` e `isNan()`. Empezaremos estas líneas analizando el comportamiento de la función `parseInt()`.
2. `parseInt()` convierte los datos de una cadena de caracteres en valores numéricos enteros. A diferencia de la función `parseFloat()`, ésta contiene dos argumentos en su sintaxis: el primero es la cadena de texto en sí misma y el segundo, opcional, es una base especificada: 1

   ```
   parseInt(cadena [, base])
   ```
3. ¿Cómo funciona? `parseInt()` convierte la cadena de caracteres y trata de devolver como resultado un número entero basado en una base especificada de forma opcional en el segundo argumento de la sintaxis.
4. La base especificada como segundo argumento de la función `parseInt()` permite que si dicha base es 10 la conversión dé como resultado un número decimal; si la base es 8, el resultado será un octal, y si es 16, el resultado será hexadecimal. En el caso en que esta base sea un número mayor que 10, las letras del alfabeto indican valores mayores que 9. 2

1

```
parseInt("34")
parseInt("34", 2)
```

La primera definición de la función `parseInt()` cuenta con el argumento imprescindible para que la conversión pueda llevarse a cabo; la segunda, muestra también la base especificada opcional sobre la cual debe realizarse en este caso la conversión.

2

```
parseInt("58") // La conversión da 58
parseInt("101011",2) // La conversión da 43
parseInt("34",8) // La conversión da 28
parseInt("3F",16) // La conversión da 63
```

025

5. En el caso en que la función no encuentre ningún número en dicha base, ignora el caracter encontrado y los situados a su derecha y devuelve como resultado el valor entero convertido hasta el punto. 3
6. Ahora bien, si el primer caracter no puede ser convertido a un número según la base especificada, entonces el resultado de la conversión es `NaN` puesto que `parseInt()` sólo trabaja con números enteros. 4
7. Y es en estos casos, en aquellos en que `parseInt()` recibe como resultado de la conversión el valor `NaN`, que puede entrar en acción la función `isNaN()`. Esta función da como resultado un valor booleano (`true` o `false`) dependiendo del resultado que reciba por parte de `parseInt()`. Si lo que recibe es un número, `isNaN()` devolverá `false`, mientras que si recibe el valor `NaN` devolverá el valor `true`. Según el valor devuelto, la acción descrita en el código se ejecutará o no.
8. En la imagen 5 puede ver un fragmento de código en el cual se utilizan las funciones parseInt() e isNaN() de forma complementaria. En este caso, se está diseñando un pequeño cuestionario para el usuario, en el cual se pregunta por su número favorito. Gracias a estas dos funciones, el script lanzará un mensaje de advertencia ("¡Esto no es ningún número!") si aquello que escribe el usuario no es un número. Dése cuenta también que el script es una sentencia condicional; habitualmente las funciones descritas en este ejercicio se ejecutarán en este tipo de funciones.

3

```
parseInt("6.95") // La conversión da 6
parseInt("32Jh9k") // La conversión da 32
```

4

```
parseInt("abc") // La conversión es NaN
```

5

```
<script type="text/javascript">
    var numFav = prompt('¿Cuál es tu número favorito?');
    numFav = parseInt(numFav);
    if (isNaN(numFav)) {
      numFav = prompt('¡Esto no es ningún número!');
    }
</script>
```

Tenga en cuenta que la función `isNaN()` también puede ser complementaria de la función `parseFloat()`, descrita en el ejercicio anterior.

Valores numéricos especiales

JAVASCRIPT TIENE DOS TIPOS DE VALORES numéricos especiales: los que indican valores de error, NaN e Infinity, y dos tipos de ceros, +0 y -0. Respecto a estos dos últimos, es cierto que no es habitual verlos en código JavaScript, sin embargo, debe saber que este lenguaje de programación almacena el signo y la magnitud de un número de forma separada.

1. En este ejercicio trataremos los valores numéricos considerados especiales en JavaScript. Aunque sobre el primero de ellos, `NaN`, ya hemos tenido ocasión de hablar en alguna ocasión, empezaremos por éste.
2. Aunque no lo parezca, el valor `NaN`, acrónimo de Not a Number, es un valor numérico, y así puede comprobarse, por ejemplo, aplicando el método `typeof()`: 1

```
typeof NaN == 'Number'
```

3. ¿En qué situaciones se produce como resultado el valor NaN? Principalmente cuando se produce un valor erróneo. Por ejemplo, cuando un valor numérico no puede ser considerado como tal: 2

```
Number('abc') // Se convierte en NaN
Number('undefined') // Se convierte en NaN
```

4. Otras situaciones en que el valor NaN se produce como resultado de una conversión es en operaciones matemáticas falli-

1

```
typeof NaN == 'Number'
```

Paradójicamente, el valor NaN es un valor numérico.

2

```
Number('abc') // Se convierte en NaN
Number('undefined') // Se convierte en NaN
```

Recuerde que la función `Number()` es la más indicada para convertir valores numéricos. Sin embargo, las cadenas de texto no pueden considerarse como número, igual que el valor `undefined`.

das y en aquellos casos en que NaN es uno de los operandos de la sentencia: [3]

```
Math.acos(2) // Se convierte en NaN
NaN + 3 // Se convierte en NaN
```

5. Hablemos ahora del valor numérico `Infinity`. Este valor se utiliza para indicar que existe un error numérico, el cual puede deberse a dos causas: que el número generado sea demasiado largo y, por tanto, no puede ser representado, o bien que se esté intentando realizar una división entre cero. [4]

   ```
   Math.pow(2, 1024) // El resultado es Infinity
   ```

   pero

   ```
   Math.pow(2,1023) //El resultado es 8.98846567431158e+307
   ```

6. En el caso de que los dos operandos de un resta o un división sean el valor `Infinity`, se conseguirá el valor de error `NaN`; en cambio, si se pretende sumar o multiplicar dos operandos `Infinity`, se obtendrá como resultado el valor `Infinity`. [5]

7. Por último, y para terminar con los valores numéricos especiales, destacaremos que JavaScript mantiene separadamente la magnitud y el signo de cualquier número, incluido el 0. Es por esta razón que el 0 puede ser positivo (+0) y negativo (-0). Cuando escribimos un 0, estamos suponiendo que es +0 y, de hecho, no existe diferencia alguna entre ambos ceros. [6]

IMPORTANTE

Podemos afirmar que el valor `Infinity` es más grande y más pequeño que cualquier otro valor numérico.

3

```
Math.acos(2) // Se convierte en NaN
Math.log(-1) // Se convierte en NaN
Math.sqrt(-1) // Se convierte en NaN

25 / NaN // Se convierte en NaN
NaN + 3 // Se convierte en NaN
```

4

```
Math.pow(2, 1024) // El resultado es Infinity
3/ 0 // El resultado es Infinity
3 / -0 El resultado es -Infinity
```

El exponente de un número debe estar entre -1023 y 1024. En el caso en que el exponente sea demasiado pequeño, el número se convierte en 0; si es demasiado grande, entonces se convierte en `Infinity`.

5

```
Infinity - Infinity // El resultado es NaN
Infinity / Infinity // El resultado es NaN
Infinity + Infinity // El resultado es Infinity
Infinity * Infinity // El resultado es Infinity
```

6

```
-0 // Es igual a 0
+0 === -0 // Da como resultado true
-0 < +0 // Da como resultado false
+0 < -0 // Da como resultado false
```

Cómo se representan los números

LOS NÚMEROS EN JAVASCRIPT TIENEN UNA precisión de 64 bits. La representación interna de los valores numéricos está basada en el estándar IEEE 754. La distribución de los 64 bits de cada número se realiza entre el signo, el exponente y la fracción, como se detalla en este ejercicio.

1. Vamos a empezar este ejercicio declarando la fórmula mediante la cual el valor de cualquier número se contabiliza. Es la siguiente:

   $(-1)^{signo}$ x %1.**fracción** x $2^{exponente}$

2. Esta fórmula está directamente relacionada con el modo en que se distribuyen los 64 bits de un número en JavaScript. En la tabla incluida como imagen 1 puede comprobar gráficamente esta distribución.

3. El signo % tiene como significado que el número central está escrito en lenguaje binario: un 1 seguido de un punto binario, seguido de una fracción binaria (o número natural). Esta explicación puede parecer un tanto complicada por lo que a continuación veremos algunos ejemplos para aclararla:

   +0 (signo=0; fracción=0; exponente= -1023)
   -0 (signo=1; fracción=0; exponente= -1023)
   1 $=(-1)^0$x%1.0x2^0 (signo=0; fracción=0; exponente= 0)
   2 $=(-1)^0$x%1.0x2^1
   3 $=(-1)^0$x%1.1x2^0 (signo=0; fracción=2^{51}; exponente= 0)
   0.5 $=(-1)^0$x%1.0x2^{-1}
   -1 $=(-1)^1$x%1.0x2^0

1

Signo	Exponente [-1023, 1024]	Fracción
1 bit	11 bits	52 bits
bit 63	bits 62-52	bits 51-0

4. Quizás alguna de las afirmaciones detalladas en las líneas anteriores no le ha quedado demasiado clara. Por si necesita una ayuda, vamos a tratar de explicar las líneas que contienen la información para los dos ceros y para el 3.
5. Empezamos por los dos ceros, el positivo (+0) y el negativo (-0). Dado que la fracción se encuentra prefijada por el valor 1, no es posible representar el 0 con ello. Por esta razón, JavaScript codifica un cero mediante la fracción 0 (fracción = 0) y el exponencial especial -1023. [2]
6. Si tenemos en cuenta que en JavaScript existen dos ceros, uno positivo y otro negativo, entonces el signo puede ser tanto + como -. (Si necesita recordar algún aspecto relacionado con los dos ceros de JavaScript, recupere la lección anterior.)
7. En cuanto al valor 3 de la lista codificada, cabe destacar que el bit 53 es el más alto, el más importante, de la fracción. Este bit es 1. [3]
8. Terminaremos este ejercicio hablando de los exponentes especiales. Ya vimos en el ejercicio anterior que el exponente de un número debe estar comprendido entre el -1023 y el 1024. Los exponentes especiales son precisamente estos dos valores, no los valores comprendidos entre ellos.
9. ¿Cuándo se utilizan cada uno de ellos? 1024 indica errores derivados de valores como `NaN` e `Infinity`, estudiados en el ejercicio anterior, mientras que -1023 se utiliza para indicar un cero (cuando la fracción = 0) y valores pequeños por debajo del cero (cuando la fracción no es 0).

2

```
+0 (signo=0; fracción=0; exponente= -1023)
-0 (signo=1; fracción=0; exponente= -1023)
```

3

```
3  =(-1)0x%1.1x20  (signo=0; fracción=251; exponente= 0)
```

Errores de redondeo

AUNQUE EN JAVASCRIPT LOS NÚMEROS ESTÁN representados internamente por números de punto flotante binarios, la realidad es que dichos valores se escriben como números de punto flotante decimales. Para poder comprender por qué esta diferencia produce imprecisiones, es necesario recordar que las fracciones pueden ser representadas de ambos modos. En este ejercicio vamos a intentar describir esta representación y comprender por qué en muchas ocasiones se producen errores de redondeo en este lenguaje de programación.

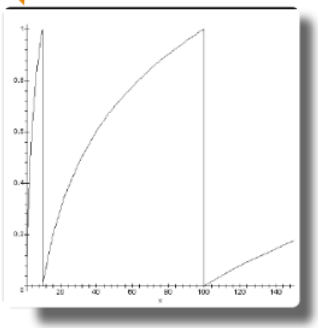
1. Para empezar diremos que en el sistema decimal, las fracciones contienen una mantisa, representada por la letra *m*, dividida por un múltiplo de 10. Recuerde que la mantisa es la diferencia entre un número y su parte entera, es decir, su parte fraccionaria. La fórmula que representa el sistema decimal sería la siguiente:
2. $\frac{m}{10^e}$
3. Como puede comprobar en esta fórmula, el denominador siempre es un 10, lo que conlleva a que fracciones del tipo 1/3 no puedan ser representadas como un número de punto flotante decimal (no es posible obtener un 3 en el denominador, sólo 10).

1

$$\frac{m}{10^e}$$

El valor *m* representa la mantisa de la fracción. Por ejemplo, en el número decimal 37,2548, la parte entera es 37 y la mantisa, 0,2548.

2



Este es el gráfico de la función mantisa.

4. Por su parte, los números de punto flotante binarios sólo pueden contener valor múltiplos de 2 como denominador. Dicho esto, podemos deducir que habrá unos números de punto de flotante decimales que podrán ser representados en el sistema binario y otros, no. En la imagen 3 puede ver unos ejemplos de fracciones con múltiples de 2 que pueden ser representados en el sistema binario. 3
5. Las fracciones que contienen un número distinto a 2 como denominador no pueden ser representadas de forma precisa como número binario. 4
6. Para trabajar estos errores de redondeo, es recomendable no comparar directamente números enteros, sino utilizar el valor denominado épsilon estándar. Este valor es exactamente el 2^{-53} y su representación en el lenguaje de programación es la siguiente: 5

```
var EPSILON = Math.pow(2, -53);
 function epsEqu(x, y) {
  return Math.abs(x - y) < EPSILON;
}
```

7. Otra forma de obtener resultados correctos en los casos en que una comparación convencional podría ser inadecuada es mediante el método `epsEqu()`:

```
0.1 + 0.2 === 0.3 // Da como resultado false
epsEqu(0.1+0.2, 0.3) // Da como resultado true
```

3

- $0.5_{dec} = \frac{5}{10} = \frac{1}{2} = 0.1_{bin}$
- $0.75_{dec} = \frac{75}{100} = \frac{3}{4} = 0.11_{bin}$
- $0.125_{dec} = \frac{125}{1000} = \frac{1}{8} = 0.001_{bin}$

4

- $0.1_{dec} = \frac{1}{10} = \frac{1}{2 \times 5}$
- $0.2_{dec} = \frac{2}{10} = \frac{1}{5}$

5

```
var EPSILON = Math.pow(2, -53);
 function epsEqu(x, y) {
  return Math.abs(x - y) < EPSILON;
}
```

El valor estándar `epsilon` está considerado de doble precisión.

6

```
0.1 + 0.2 === 0.3
// Da como resultado false

epsEqu(0.1+0.2, 0.3)
// Da como resultado true
```

Números enteros

EN JAVASCRIPT, LOS NÚMEROS ENTEROS SE representan internamente de dos maneras: como un número muy pequeño sin fracción decimal (motores JavaScript) y mediante operadores enteros (especificación ECMASCRIPT).

1. En este ejercicio veremos cómo trabajan los números enteros en JavaScript y qué características tienen. Para empezar, debe saber que existen una serie de rangos de números enteros en JavaScript que se consideran importantes. El primero de estos rangos es el que va del valor 2^{-53} al valor 2^{53} y los números incluidos en él se denominan números enteros seguros.
2. ¿En qué se basan los números enteros seguros? En su representación matemática en JavaScript. Se considera que los números enteros incluidos en el rango de 2^{-53} a 2^{53} son seguros porque los números enteros matemáticos coinciden exactamente con sus representaciones en JavaScript.
3. Todos aquellos números que se encuentran fuera de este rango se consideran no seguros, puesto que dos o más números enteros matemáticos se representan con el mismo número entero en JavaScript. ¿Qué significa esto? Significa que cualquier número después del 2^{53} JavaScript sólo puede representar los números enteros matemáticos cada dos.

IMPORTANTE

Como se apuntó en el primer ejercicio de este curso, JavaScript es un lenguaje de programación creado por Netscape, el cual se vió muy pronto amenazado por otros lenguajes de programación creados por otras compañías. Con el fin de evitar intromisiones graves, Netscape solicitó a la organización de estándares Ecma International que alojará su lenguaje de programación como el estándar. Como ya existía otro lenguaje con el nombre Java (el creado por Sun), se decidió bautizar el estándar de Netscape como ECMAScript. A finales de 2014, se estaba finalizando la versión 6 de dicho estándar.

1

```
        arreglo[i]=parseInt(prompt("Digite numero"));
        document.write(arreglo[i] + " ,");
        arreglo[i]=arreglo[i]+1;
}
document.write("<br>")

document.write ("Los numeros modificados son: ");
for(i=0;i<5;i++)
{
        document.write(arreglo[i] + " ,");
}
```

Es muy importante saber cómo funcionan los números en JavaScript. De esta manera no podremos evitar la aparición de errores pero sí localizar de dónde proceden.

4. Los índices de valores en matrices también son rangos de números enteros importantes en JavaScript. Las características que tienen los números incluidos en este rango son las siguientes: son números de 32 bits sin signo (ni + ni -); tiene una longitud máxima de 2^{32} a 1, y el rango de valores es 0, 2^{32} a 1, sin incluir la longitud máxima.
5. Los operandos bit a bit también se consideran de gran importancia en Javascript. Estos operandos incluyen el operador >>>, de 32 bits, sin signo y con un rango de 0 a 2^{32} y el resto de operadores bit a bit, es decir, los de 32 bits, con signo (+ o -) y dentro del rango -2^{31} - 2^{31}.
6. JavaScript sólo puede trabajar con números enteros mayores de 53 bits y puede representarlos como números decimales. La tabla en la imagen 2 muestra cómo se lleva a cabo esta representación. 2
7. Si hablamos de los números de 54 bits, el dígito menos importante es siempre 0; para los números de 55 bits, los dos dígitos menos importantes son también 0, y así sucesivamente. Vea los ejemplos siguientes para confirmar estos datos: 3

```
Math.pow(2, 53) - 1 = 9007199254740991  //Correcto
Math.pow(2, 53) = 9007199254740992  //Correcto
Math.pow(2, 53) + 1 = 9007199254740992  // No se puede representar
Math.pow(2, 53) + 2  = 9007199254740994  //Correcto
```

2

1 bit	0	
1 bit	1	%1 x 2^0
2 bits	2 a 3	%1.f_{51} x 2^1
3 bits	4 a 7 = 2^2-(2^3 a 1)	%1.$f_{51}f_{50}$ x 2^2
4 bits	2^3-(2^4 a 1)	%1.$f_{51}f_{50}f_{49}$ x 2^3
...	...	...
53 bits	2^{52}-(2^{53} a 1)	%1.f_{51}...f_0 x 2^{52}

3

```
Math.pow(2, 53) - 1 = 9007199254740991
//Correcto
Math.pow(2, 53) = 9007199254740992
//Correcto
Math.pow(2, 53) + 1 = 9007199254740992
// No se puede representar
Math.pow(2, 53) + 2  = 9007199254740994
//Correcto
```

Números enteros seguros

IMPORTANTE

Todos aquellos números que se encuentran fuera del rango del valor 2^{-53} al valor 2^{53} se consideran no seguros, puesto que dos o más números enteros matemáticos se representan con el mismo número entero en JavaScript. ¿Qué significa esto? Significa que cualquier número después del 2^{53} JavaScript sólo puede representar los números enteros matemáticos cada dos.

EN EL EJERCICIO ANTERIOR HEMOS TENIDO la oportunidad de empezar a conocer los denominados números enteros seguros de JavaScript, uno de los rangos de números enteros más importantes de este lenguaje de programación. Es por esta razón que queremos dedicarles por completo un ejercicio y entender un poco más cómo trabajan y en qué consisten.

1. Como hemos indicado en el ejercicio anterior, los números enteros seguros son los números incluidos en el rango que va del valor 2^{-53} al valor 2^{53}. Este tipo de números se basan en su representación matemática en JavaScript. Se considera que los números enteros incluidos en este rango de valores son seguros porque los números enteros matemáticos coinciden exactamente con sus representaciones en JavaScript.
2. Un número entero seguro de JavaScript es el que inequívocamente representa un entero matemático. ECMAScript 6 proporciona un par de constantes para definir un número entero seguro, [1] así como una función concreta para determinar si un número entero es o no seguro. [2]
3. ¿Y cuál es el significado y el modo de proceder de esta función? Teniendo en cuenta que el valor a comprobar es `n`, la función comprueba si dicho valor es un número y si es entero. Si el valor `n` es ambas cosas, será seguro siempre y cuando sea mayor o igual que `MIN_SAFE_INTEGER` y menor o igual que el valor en `MAX_SAFE_INTEGER`.

1

```
Number.MAX_SAFE_INTEGER = Math.pow(2, 53)-1;
Number.MIN_SAFE_INTEGER = -Number.MAX_SAFE_INTEGER;
```

2

```
Number.isSafeInteger = function (n) {
  return (typeof n === 'number' &&
     Math.round(n) === n &&
     Number.MIN_SAFE_INTEGER <= n &&
     n <= Number.MAX_SAFE_INTEGER);
}
```

4. Además de saber cuándo estamos ante un número entero seguro o no seguro, también es posible saber cuándo se obtienen resultados correctos o incorrectos derivados de cálculos aritméticos. Por ejemplo, si ve una operación como [3]

```
201254851 + 5 = 201254855
```

5. Ya puede comprobar a simple vista que es incorrecta; se trata de dos operandos seguros (`201254851` y `5`) y un resultado no seguro (`201254855`). La verificación la realizamos como sigue: [4]

```
Number.isSafeInteger(201254851) == true
Number.isSafeInteger(5) == true
Number.isSafeInteger(201254855) == false
```

6. Veamos un caso contrario, es decir, uno en que uno de los operandos no es seguro aunque el resultado sí lo es: [5]

```
9007199254740995 -10 = 9007199254740986
```

7. Como veremos a continuación, el primer operando (`9007199254740995`) no es seguro pero el segundo (`10`) y el resultado (`9007199254740986`) sí lo son: [6]

```
Number.isSafeInteger(9007199254740995) == false
Number.isSafeInteger(10) == true
Number.isSafeInteger(9007199254740986) == true
```

8. Con todo esto podemos confirmar que el resultado de aplicar un operador entero (`op`) es siempre correcto, siempre y cuando tanto los operandos como el resultado sean seguros.

[3]

```
201254851 + 5 = 201254855
// El resultado correcto sería
201254851 + 5 = 201254856
```

[4]

```
Number.isSafeInteger(201254851) == true
Number.isSafeInteger(5) == true
Number.isSafeInteger(201254855) == false
```

[5]

```
9007199254740995 -10 = 9007199254740986
```

[6]

```
Number.isSafeInteger(201254851) == true
Number.isSafeInteger(5) == true
Number.isSafeInteger(201254855) == false
```

Convertir datos a números enteros

IMPORTANTE

Según los expertos desarrolladores, el método de conversión con las diferentes funciones `Math()` es el más adecuado para realizar este cálculo; el uso de la función `ToInteger()` y de los operadores bit a bit tienen aplicaciones concretas, y el método con la función `parseInt()` es adecuado para convertir cadenas de texto.

COMO YA HEMOS INDICADO EN MÁS de una ocasión en estos últimos ejercicios dedicados a los números, en Javascript todos los números son decimales. En concreto, los números enteros son números decimales sin fracción. JavaScript permite convertir cualquier número en un número entero, y pone para ello a disposición del usuario diferentes procedimientos.

1. En este ejercicio le mostraremos los diferentes procedimientos o métodos mediante los cuales es posible convertir un número `n` en un número entero. Estos métodos son 4: las funciones `Math.floor()`, `Math.ceil()` y `Math.round()`; 1 la función `ToInteger()`; el uso de operadores bit a bit binarios, y la función global `parseInt()`, que tratamos en el ejercicio 25. Veamos las principales características de cada uno de estos métodos.
2. Las mencionadas variantes de la función `Math()` son la mejor opción para convertir un número en un entero. Así, la función `Math.floor()` convierte su argumento en el número entero inferior más cercano; 2 la función `Math.ceil()` convierte su argumento en el número entero superior más cercano, 3 y, por último, la función `Math.round()` convierte su argumento en el número entero más cercano. 4

1

```
var preguntas = [
  ['¿Cuál es la capital de Austria?', "Viena"],
  ['¿Cuál es la capital de Noruega?', "Oslo"],
  ['¿Qué lengua se habla en Madagascar?', "Malgache"],
  ['¿Cómo se llaman los habitantes de Panamá?', "Panameños"]
];
var aleatorio = Math.floor(Math.random() *preguntas.length);
var aleat = preguntas[aleatorio];
```

Existen muchísimas aplicaciones de la función `Math()` en Javascript. En este ejemplo, se ha utilizado para que una serie de preguntas aparezcan en la página de forma aleatoria.

2

```
Math.floor(6.8) // El resultado es 6
Math.floor(-6.8) // El resultado es -7
```

3

```
Math.ceil(6.2) // El resultado es 7
Math.ceil(-6.2) // El resultado es -6
```

4

```
Math.round(6.2) // El resultado es 6
Math.round(6.5) // El resultado es 7
Math.round(6.8) // El resultado es 7
```

`Math.round(x)` es lo mismo que `Math.ceil(x * 0.5)`.

3. El segundo de los métodos para convertir un número en un entero es la función `ToInteger()`, la cual elimina la fracción en un número decimal.[5] La definición del resultado de esta función, según la especificación de ECMAScript, es la siguiente:

```
sign(número) x floor(abs(number))
```

4. Y la operación en JavaScript podría resumirse en las siguientes líneas:[6]

```
function ToInteger(x) {
 x = Number(x);
 return x < 0 ? Math.ceil(x) : Math.floor(x);
}
```

5. Los operadores bit a bit binarios convierten uno de los operandos existentes en un número entero de 32 bits; este entero será posteriormente manipulado para producir un resultado que también sea un entero de este tipo. Los operadores que pueden utilizarse son OR (`|`)[7] y los operadores `shift`.[8]

6. El último de los procedimientos para convertir un dato en un número entero es la función `parseInt()`. El uso de esta función, sin embargo, se reduce a los cadenas de texto (*strings*), lo que significa que puede convertir estos elementos en números pero no se basará nunca en un valor numérico para realizar una conversión a entero.

7

```
function ToInt32(x) {
  return x | 0;
}
```

8

```
function ToInt32(x) {
  return x << 0;
}
// Esta función convierte
el valor x en un número entero
con signo de 32 bits.
```

5

```
ToInteger(6.2) // El resultado es 6
ToInteger(6.5) // El resultado es 6
ToInteger(6.8) // El resultado es 6

ToInteger(-6.2) // El resultado es -6
ToInteger(-6.5) // El resultado es -6
ToInteger(-6.8) // // El resultado es -
```

La función `ToInteger()` es una operación interna de ECMAScript (vea el recuadro IMPORTANTE del ejercicio 29 para recordar la información de este nombre).

6

```
function ToInteger(x) {
 x = Number(x);
 return x < 0 ? Math.ceil(x) : Math.floor(x);
}
// Si el número es negativo, se puede
utilizar la función Math.ceil(),
evitando así la operación sign.
```

Operadores aritméticos

LOS OPERADORES ARITMÉTICOS SON AQUELLOS QUE se utilizan para operaciones con números. En este ejercicio realizaremos un repaso a los operadores aritméticos vistos hasta el momento y añadiremos otros que todavía no han sido definidos en estas páginas.

1. El operador + suma valores numéricos y concatena números y cadenas de caracteres:

```
7.2 + 1.3 == 8.5
14 + ' visitas' == 14 visitas
```

2. Los operadores - * y / realizan una resta, una multiplicación y una división de los valores indicados. En el caso del operador %, es preciso recordar que el valor que devuelve la operación como resultado tendrá el mismo signo (+ o -) que el signo que muestre el primer operando.

3. Dada la característica del operador % referente al signo que acompaña al resultado, existe una función concreta q no es correcta. Dicha función es la siguiente:

```
function isOdd(n) {
 return n % 2 === 1;
}
console.log(isOdd(-5)); // false
console.log(isOdd(-4)); // false
```

1

```
7.2 + 1.3 == 8.5
// Suma de valores numéricos
14 + ' visitas' == 14 visitas
// Concatenación de número y texto
```

2

```
9 %7 == 2
-9 % 7 == -2
// El signo del valor
obtenido como resultado
coincide con el del
primer operando
```

3

```
return n % 2
function isOdd(n) {
 return n % 2 === 1;
}
console.log(isOdd(-5));
// El resultado es false
console.log(isOdd(-4));
// El resultado es false
```

4. La función escrita correctamente sería la siguiente: 4

```
function isOdd(n) {
 return Math.abs(n % 2) === 1;
}
console.log(isOdd(-5)); // true
console.log(isOdd(-4)); // false
```

5. Los operadores - y + se utilizan delante de un valor numérico; el signo - niega el argumenta y el signo + lo mantiene. En el caso en que el operador + acompañe a un valor no numérico, entonces lo convierte en número.

6. Existen unas combinaciones de operadores que acompañan variables: ++ y --. Dependiendo de si se sitúan delante o detrás del valor de la variable, estos realizan una función u otra. Si los signos van delante incrementan o disminuyen el valor de la variable en 1: 5

```
var caso1 = 250;
++caso1 == 251 // caso1 = 251
--caso1 == 249 // caso1 = 249
```

7. Al contrario, si los signos se sitúan detrás del valor de la variable, entonces incrementa o disminuye este valor en 1 y lo devuelve:

```
var caso1 = 250;
caso1++ == 250 // caso1 = 251
caso1-- == 250 // caso1 = 249
```

8. Como puede comprobar, la posición del signo le ayuda a saber cuándo debe devolverse el resultado, antes o después de incrementarse o disminuirse el valor de la variable.

5

```
if (numFav) {
 document.write('<p>El número ' + numFav + ' es un buen número</p>');
}
for (var num=11; num<=20; num++) {
 document.write('<h3>Número ' + num + '<br></h3>');
}
```

Math.abs(n % 2)

4

```
function isOdd(n) {
 return Math.abs(n % 2) === 1;
}
console.log(isOdd(-5));
// El resultado es true
console.log(isOdd(-4));
// El resultado es false
```

6

```
var caso1 = 250;
caso1++ == 250 // caso1 = 251
caso1-- == 250 // caso1 = 249
```

Operadores bit a bit

LOS OPERADORES BIT A BIT DE JavaScript convierten sus operandos en números enteros de 32 bits y producen un resultado del mismo tipo, es decir, un número entero de 32 bits. En este ejercicio trataremos algunos aspectos de este tipo concreto de operadores, los cuales se pueden utilizar para procesar protocolos binarios, algoritmos especiales, etc.

1. Los números enteros de 32 bits no cuentan con ningún signo visible. Sin embargo, es posible codificar números negativos. Aunque el límite entre los números positivos y negativos es fluido, es preciso decidir el signo que debe llevar el valor en el momento de convertirlo de o a un número de JavaScript.
2. Así, podemos separar los números enteros de 32 bits en dos grupos:

   Si el bit más elevado es 0, el número es cero o positivo,
   Si el bit más elevado es 1, el número es negativo
3. Según esta separación de números, podemos afirmar que un valor como 4294967295, si se interpreta como un número entero de 32 bits, pasa a ser -1 cuando se convierte en un número de Javascript. 1

1

```
ToInt32(4294967295)
//Se convierte en -1 en JavaScript
```

La función `ToInt32()` elimina la fracción de un valor y aplica el módulo 2^{32}.

2

```
parseInt('11001010', 2) & parseInt('1111', 2)).toString(2)
// El resultado es 1010
(parseInt('11001010', 2) | parseInt('1111', 2)).toString(2)
// El resultado es 11001111
(parseInt('11001010', 2) ^ parseInt('1111', 2)).toString(2)
// El resultado es 11000101
```

El operador bit a bit `&` se lee AND; `|` se lee OR, y `^` se lee Xor o eXclusive Or.

4. JavaScript tiene tres operadores bit a bit binarios: & | y ^. A continuación se muestra el uso de cada uno de los operadores con la función parseInt(), que, como recordará, permite convertir datos en un valor numérico: [2]

```
parseInt('11001010', 2) & parseInt('1111', 2)).toString(2)
// El resultado es 1010

(parseInt('11001010', 2) | parseInt('1111', 2)).toString(2)
// El resultado es 11001111

(parseInt('11001010', 2) ^ parseInt('1111', 2)).toString(2)
// El resultado es 11000101
```

5. Tenga en cuenta que, aunque los operadores & y | existen dobles (&& y ||), el operador ^^ no existe. Si alguna vez se encuentra con esta operador doble, sepa que el modo en que debe funcionar es el siguiente: [3]

```
caso1 ^^ caso2 === (caso1 && !caso2) || (!caso1 && caso2)
```

6. Para terminar este ejercicio, hablaremos de los tres tipos de operadores de desplazamiento de bits en JavaScript: << >> >>>. El primero, <<, desplaza hacia la izquierda la representación binaria del primer operando el número de bits representado en el segundo y rellena con ceros por la derecha. [4] El operador >> desplaza hacia la derecha la representación binaria del primer operando el número de bits representado en el segundo y descarta los bits que sobran por la derecha. [5] El tercer tipo, >>>, desplaza hacia la derecha la representación binaria del primer operando el número de bits representado en el segundo, descarta los bits que sobran por la derecha y rellena con ceros por la izquierda. [6]

3

```
caso1 ^^ caso2 === (caso1 && !caso2) || (!caso1 && caso2)
// La aplicación doble del operador ^ no existe.
```

4

```
9<<2 == 36
// 1001 movido 2 bits a la iquierda
resulta 100100, que es 36.
```

5

```
9>>2 == 2
// 1001 movido 2 bits a la derecha
resulta 10, que es 2.
-9>>2 == -3
// El signo se preserva.
```

6

```
19>>>2 == 4
// 10011 movido 2 bits a la derecha
resulta 100, que es 4.
Para números no negativos, los dos
tipos de desplazamiento a la derecha
dan el mismo resultado.
```

Propiedades del objeto Number

IMPORTANTE

Recuerde que el objeto `Number` tiene dos usos principales: si el argumento del objeto no puede ser convertido en un valor numérico, devuelve el valor `NaN` (Not a Number); en el caso de trabajar en un contexto de no constructor (por ejemplo, sin el operador `new`), el objeto `Number` se puede utilizar para realizar conversiones de tipo de datos.

EL OBJETO NUMBER DE JAVASCRIPT ES un objeto envolvente que permite trabajar con valores numéricos. Los objetos del tipo Number se crean a partir del constructor Number(), el cual tiene una serie de propiedades que definiremos con todo detalle en este ejercicio.

1. El objeto Number dispone, entre otras, de las siguientes propiedades: `Number.MAX_VALUE`, `Number.MIN_VALUE`, `Number.NaN`, `Number.NEGATIVE_INFINITY` Y `Number.POSITIVE_INFINITY`. Veamos con ejemplos en qué consisten cada una de ellas.
2. La propiedad `Number.MAX_VALUE` significa el número positivo más alto que se puede representar. Esta propiedad tiene un valor aproximado de 1.79E+308; cualquier otro valor más alto que éste es representado con el valor `INFINITY`. `MAX_VALUE` es una propiedad estática del objeto Number, por lo que siempre quedará representada en la cadena `Number.MAX_VALUE`. 1
3. Como puede imaginar, la propiedad `Number.MIN_VALUE` significa el número positivo más pequeño que se puede representar; es el número positivo más cercano al 0. `MIN_VALUE` tiene el valor aproximado de 5e-324 y todos aquellos valores más pequeños a éste se convierten en 0. Igual que la propiedad `MAX_VALUE`, `MIN_VALUE` es una propiedad estática del objeto Number, por lo que siempre quedará representada en la cadena `Number.MAX_VALUE`. 2

1

```
if (num1 * num2 <= Number.MAX_VALUE) {
  func1();
} else {
  func2();
}
```

En este script de ejemplo, se multiplican los valores contenidos en `num1` y `num2`. Si el resultado de la multiplicación es menor o igual que el valor de `MAX_VALUE`, se ejecutará la función1; si no, se ejecutará la función2.

2

```
if (num1 / num2 >= Number.MIN_VALUE) {
  func1();
} else {
  func2();
}
```

En este script de ejemplo, se dividen los valores contenidos en `num1` y `num2`. Si el resultado de la división es mayor o igual que el valor de `MIN_VALUE`, se ejecutará la función1; si no, se ejecutará la función2.

4. La tercera propiedad del objeto Number que deseamos destacar en este ejercicio es `Number.NaN`. (Si necesita recordar algún aspecto acerca del valor `NaN`, no dude en recuperar el ejercicio 25 de este libro.) La propiedad Number.NaN representa exactamente el valor NaN, es decir, Not a Number. De hecho, equivale a este valor. Para acceder a esta propiedad estática del objeto Number, no es preciso crear un objeto de este tipo.

5. La propiedad `Number.NEGATIVE_INFINITY` representa el valor negativo Infinity (`-Infinity`). El valor de esta propiedad es el mismo que el valor negativo de la propiedad global `Infinity` de objetos. Resulta importante saber que esta propiedad se comporta de forma distinta que el valor matemático infinity: 3

   Cualquier valor positivo x `negative_infinity` = `negative_infinity`
   Cualquier valor negativo x `negative_infinity` = `positive_infinity`
   0 x `negative_infinity` = `NaN`
   NaN x `Negative_infinity` = `NaN`
   Cualquier número / `negative_infinity` = 0
   `negative_infinity` / cualquier valor negativo = `positive_infinity`
   `negative_infinity` / cualquier valor positivo = `negative_infinity`
   `negative_infinity` / `negative_infinity` o `positive_infinity` = `NaN`

6. Por último, la propiedad `Number.POSITIVE_INFINITY` representa el valor positivo Infinity. El valor de esta propiedad es el mismo que el valor negativo de la propiedad global `Infinity` de objetos. La propiedad `Number.POSITIVE_INFINITY` se utiliza para indicar una condición de error que devuelve un número finito en caso de éxito. Esta propiedad también se comporta de forma distinta que el valor matemático `infinity`. 4

034

IMPORTANTE

`Number.prototype` también es una propiedad del objeto Number, la cual representa el prototipo para el constructor `Number`. Debido a que esta propiedad dispone de diferentes métodos, será tratada en exclusiva en el ejercicio siguiente.

3

```
var num1 = (-Number.MAX_VALUE) * 4;

if (num1 == Number.NEGATIVE_INFINITY) {
  num1 = returnFinite();
}
```

En este ejemplo, se ha creado una variable, `num1`, que contiene un valor más pequeño que el valor mínimo. Al ejecutarse la sentencia condicional `if`, `num1` tiene el valor `-Infinity`, por lo que se convierte en un valor más gestionable antes de continuar con el script.

4

```
var num2 = Number.MAX_VALUE * 2;

if (num2 == Number.POSITIVE_INFINITY) {
  num2 = returnFinite();
}
```

En este ejemplo, se ha creado una variable, `num2`, que contiene un valor mayor que el valor máximo. Al ejecutarse la sentencia condicional `if`, `num1` tiene el valor `Infinity`, por lo que se convierte en un valor más gestionable antes de continuar con el script.

Métodos de números primitivos

TODOS LOS MÉTODOS DE LOS NÚMEROS primitivos en JavaScript se almacenan en la propiedad `Number.prototype`. Esta propiedad representa el prototipo para el constructor `Number`. Todas las instancias del objeto Number son heredadas de la propiedad `Number.prototype`. El objeto prototipo del constructor Number puede ser modificado para que afecte a todas las instancias del objeto.

1. En este ejercicio veremos algunos de los métodos que se almacenan en la propiedad `number.prototype`. En concreto, hablaremos de los métodos `Number.prototype.toExponential()`, `Number.prototype.toFixed()`, `Number.prototype.toPrecision()`, `Number.prototype.toString()` y `Number.prototype.valueOf()`.
2. Empecemos. El método `Number.prototype.toExponential()` devuelve una cadena de texto que representa el objeto Number en notación exponencial con un dígito antes del punto decimal, y redondeado al dígito del parámetro `fractionDigits` después del punto decimal. El parámetro `fractionDigits` es opcional y contiene un número entero que especifica el número de dígitos después del punto decimal. 1
3. El método `Number.prototype.toFixed()` devuelve una representación sin exponente del número, redondeado a los dígitos del parámetro opcional `fractionDigits`. Este parámetro es el número de dígitos que aparece después del punto decimal y debe ser un valor comprendido entre el 0 y el 20, ambos incluidos. Si este valor no aparece, se utilizará el valor 0. 2

1 Si el parámetro `fractionDigits` se omite, el número de dígitos después del punto decimal se establece por defecto en el número de dígitos necesarios para representar un valor único.

```
(numObj.toExponential());  // La cadena devuelta es 7.71234e+1
(numObj.toExponential(4)); // La cadena devuelta es 7.7123e+1
(numObj.toExponential(2)); // La cadena devuelta es 7.71e+1
(77.1234.toExponential()); // La cadena devuelta es 7.71234e+1
(77 .toExponential());     // La cadena devuelta es 7.7e+1
```

2

```
numObj.toFixed();       // La cadena devuelta es '12346'
numObj.toFixed(1);      // La cadena devuelta es '12345.7'
                           (con redondeo)
numObj.toFixed(6);      // La cadena devuelta es '12345.678900'
                           (se añaden ceros)
(1.23e+20).toFixed(2);  // La cadena devuelta es '123000000000000000000.00'
```

4. Pasemos ahora al método `Number.prototype.toPrecision()`. Este método devuelve una cadena que representa el objeto Number en la precisión especificada. Como en el caso del parámetro `fractionDigits` en los dos métodos descritos anteriormente, el parámetro `precision` es opcional en este método; dicho parámetro especifica el número de dígitos significativos. En el caso en el parámetro `precision` no exista, se utilizará directamente el método `toString()`, que se describe a continuación. 3

5. El método `Number.prototype.toString()`, por su parte, devuelve una cadena que representa el objeto Number especificado. El objeto `Number` anula el método `toString()` del objeto `Object`. En el caso de los objetos de tipo Number, el método `toString()` devuelve una cadena que representa el objeto que se especifica en el argumento opcional `radix`. Este argumento es un número entero entre el 2 y el 36 que especifica la base que debe utilizarse para representar valores numéricos. Si el parámetro `radix` no se especifica, se recomienda utilizar como base el valor 10. 4

6. El último método que trataremos en este ejercicio es `Number.prototype.valueOf()`, el cual devuelve el valor primitivo envolvente del objeto Number. Cabe destacar que este método normalmente no se utiliza de forma explícita en el código de JavaScript, sino que es el lenguaje mismo el que lo aplica internamente. 5

3

```
var numObj = 5.123456;

numObj.toPrecision(); // Devuelve 5.123456
numObj.toPrecision(5); // Devuelve 5.1235
numObj.toPrecision(2); // Devuelve 5.1
numObj.toPrecision(1); // Devuelve 5
```

4

```
var num1 = 10;

num1.toString(); // Devuelve '10'
(17).toString();  // Devuelve '17'
```

Si el valor numérico es negativo, el signo – se mantiene, así como si el valor del parámetro `radix` es 2. En este caso, la cadena devuelta es la representación binaria en positivo del número precedido del signo – (menos).

5

```
var num1 = numObj.valueOf();
console.log(num1); // La cadena devuelta es 10
console.log(typeof num1); // La cadena devuelta es number
```

```
var numObj = new Number(10);
console.log(typeof numObj); // object
```

Algunas funciones numéricas

EN EJERCICIOS ANTERIORES, HEMOS PODIDO DESCUBRIR cómo se utilizan algunas funciones numéricas. En concreto, hemos tenido ocasión de tratar las funciones `isNaN()`, `parseInt()` y `parseFloat()`, entre otras. En este ejercicio, y para terminar el bloque de lecciones dedicadas al objeto Number, hablaremos de otras tres conocidas funciones: `Number.isFinite()`, `Number.isInteger()` y `Number.isSafeInteger()`.

1. La función numérica `isFinite()` comprueba si el valor contenido en el objeto Number es un número real, es decir, no es ni `Infinity` ni `NaN`. El único parámetro que presenta esa función es `value`, el cual contendrá el valor que se desea comprobar.
2. Es preciso tener en cuenta que no hay que confundir esta función con la global del mismo nombre, la cual, si es necesario, convierte el parámetro en un número antes de realizar la comprobación. La función numérica `isFinite()` no realiza forzosamente esta conversión, lo que significa que sólo los valores del tipo Number y finitos devuelven `true`.
3. La función numérica `Number.isInteger()` determina si el valor comprobado es un número entero, eliminando la fracción en un número decimal. Como y se vio en el ejercicio 31 de este libro, la definición del resultado de esta función,

1

```
Number.isFinite(Infinity);  // Devuelve false
Number.isFinite(NaN);       // Devuelve false
Number.isFinite(-Infinity); // Devuelve false
```

2

```
Number.isFinite(0);  // Devuelve true
Number.isFinite(2e64); // Devuelve true
```

3

```
Number.isFinite('0'); // Devuelve false
// Con la función global isFinite ('0')
el resultado habría sido true
```

según la especificación de ECMAScript, es `sign(número) x floor(abs(number))`, y la operación en JavaScript podría resumirse en las siguientes líneas: 4

```
function ToInteger(x) {
 x = Number(x);
 return x < 0 ? Math.ceil(x) : Math.floor(x);
}
```

4. El único parámetro que precisa esta función es el del valor que se desea comprobar. Si el valor de destino es un número entero, la función devuelve `true`, mientras que si no es un número entero o es del tipo `NaN` o `Infinity`, devuelve `false`. Puede ver algunos ejemplos del tratamiento de los valores según sean estos en la imagen 5. 5
5. Por último, la función `Number.isSafeInteger()` determina si un valor es un número entero seguro. ¿Qué entendemos por número entero seguro? Son aquellos números enteros que se encuentran entre el valor $-(2^{53}-1)$ y el valor $2^{53}-1$, ambos inclusives. Como ejemplo diremos que un número entero seguro es 2^{53}-1, puesto que puede ser representado con exactitud y, además, ningún otro número entero puede redondearlo.
6. El único parámetro que presenta esta función es el denominado `testValue`, que representa el valor que debe ser comprobado. 6

4

```
function ToInteger(x) {
 x = Number(x);
 return x < 0 ? Math.ceil(x) : Math.floor(x);
}
```

5

```
Number.isInteger(0.1); // Devuelve false
Number.isInteger(1); // Devuelve true
Number.isInteger(Math.PI); // Devuelve false
Number.isInteger(-100000); // Devuelve true
Number.isInteger(NaN); // Devuelve false
Number.isInteger(0); // Devuelve true
Number.isInteger("10"); // Devuelve false
```

6

```
Number.isSafeInteger(3);  // Devuelve true
Number.isSafeInteger(Math.pow(2, 53)); // Devuelve false
Number.isSafeInteger(Math.pow(2, 53) - 1);  // Devuelve true
Number.isSafeInteger(NaN); // Devuelve false
Number.isSafeInteger(Infinity); // Devuelve false
Number.isSafeInteger('3'); // Devuelve false
Number.isSafeInteger(3.1); // Devuelve false
Number.isSafeInteger(3.0); // Devuelve true
```

Cadenas de caracteres

LAS CADENAS DE CARACTERES (*STRINGS*, EN inglés) son secuencias de caracteres de JavaScript. Cada uno de estos caracteres es una unidad de código UTF-16 de 16 bits, lo que significa que un único caracter Unicode está representado por uno o dos caracteres de JavaScript. Esto es relevante para contar caracteres o para dividir cadenas de texto.

1. Con este ejercicio empezamos una serie dedicados a los *strings* de JavaScript. Las variables de tipo texto tienen propiedades y métodos. Como recordará, las propiedades son las características de la cadena de texto y los métodos, sus funcionalidades. 1
2. Las cadenas literales, es decir, aquellas que aparecerán de alguna manera en los documentos de destino, se delimitan con comillas. Dichas comillas pueden ser simples o dobles; usted decide cuál de ellas utilizar, aunque sea siempre coherente. Es decir, si abre una cadena de texto con comillas simples, no la cierre con comillas dobles.
3. En referencia al uso de las comillas para delimitar cadenas de texto en JavaScript, podemos considerar que en el mundo de la programación se suelen reservar las comillas dobles para el lenguaje HTML 2 y las comillas simples, para JavaScript. 3

1
```
var preguntas = [
 ['¿Cuál es la capital de Austria?', "Viena"],
 ['¿Cuál es la capital de Noruega?', "Oslo"],
 ['¿Qué lengua se habla en Madagascar?', "Malgache"],
 ['¿Cómo se llaman los habitantes de Panamá?', "Panameños"]
];
```

2
```
</head>
<body id="Areacentral">
<div id="container">
  <div id="cabecera"><img src="../images
  <div id="contentWrap">
    <div id="principal">
      <h1>Bienvenidos a mi sitio web</h1>
```

3
```
var mensaje = 'Has contestado correctamente '
mensaje += ' de ' +preguntas.length;
mensaje += ' preguntas';
var respuestas = document.getElementById('resu
respuestas.innerHTML = mensaje;
}
```

037

4. Paradójicamente, las dobles comillas también se utilizan de forma exclusiva para cadenas de caracteres en algunos lenguajes de programación, como C++ y Java, por ejemplo. Por lo tanto, en un código base multilenguaje, estará permitido utilizar las comillas dobles para los strings. 4

5. Como última consideración acerca del uso de las comillas en las cadenas de texto en JavaScript diremos que según JSON, es preciso utilizar las comillas dobles. JSON (siglas de *JavaScript Object Notation*) 5 es un texto sin formato para el almacenamiento de datos. Con el tiempo, JSON se ha ido haciendo popular, sobre todo como formato para el intercambio de datos para servicios web, para archivos de configuración, entre otros. 6 Resulta interesante saber que ECMAScript dispone de una librería para convertir una cadena de texto en formato JSON a un valor de JavaScript, y viceversa:

```
JSON.parse('{}'); // Devuelve {}
JSON.parse('true'); // Devuelve true
JSON.parse('"foo"'); // Devuelve "foo"
JSON.parse('[1, 5, "false"]'); // Devuelve [1, 5, "false"]
JSON.parse('null'); // Devuelve null
```

6. A parte de utilizar adecuadamente las comillas, el uso de sangrías es altamente recomendable; gracias a ello, el código aparecerá más limpio y claro a los ojos de quien lo lea. Utilice siempre una estructura coherente. 7

5

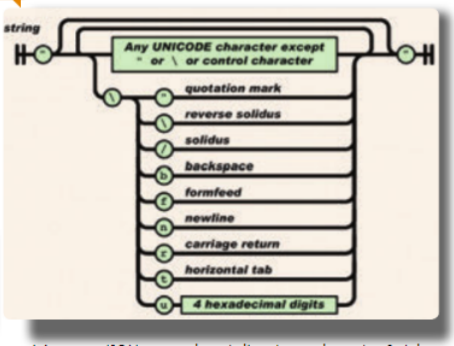
Logotipo de JSON.

4

```
string mensaje;
mensaje = "Hola";
cout << mensaje;
```

El uso de las comillas dobles para *strings* está reservado para lenguajes de programación como C++ y Java; JavaScript adquiere esta característica de estos dos lenguajes.

6



Así es como JSON representa cada tipo de caracter en JavaScript. (Fuente: http://www.json.org)

7

```
function fechaCompleta(fecha,formato)
 var diaSemana=fecha.getDay()-1;
   if (diaSemana <0){
        diaSemana=6
    }
```

Las sangrías son tan importantes como las comillas en JavaScript.

Secuencias de escape en strings

LAS SECUENCIAS DE ESCAPE SON PROPORCIONADAS por JavaScript para escribir caracteres que no se pueden introducir directamente. Las secuencias de escape pueden incluirse dentro de las cadenas de caracteres o strings.

1. En este ejercicio trataremos de describir en qué consisten las secuencias de escape en JavaScript. La mayoría de los caracteres en cadenas de texto literales se representan a ellos mismos, es decir, no tienen otro significado. 1 Sin embargo, ¿qué ocurre si deseamos insertar un caracter que no podemos escribir directamente? Hablamos, por ejemplo, de un espacio o un retroceso en un texto.
2. Es en estos casos que Javascript permite utilizar el signo \ (denominado en inglés *backslash*), para indicar que deseamos escapar un caracter y habilitar ciertas características especiales. La barra diagonal inversa informa al intérprete de JavaScript que el caracter que viene a continuación es un caracter especial. 2
3. ¿Y si necesitamos escribir en una cadena de texto del código una barra diagonal inversa, sin que ésta habilite la secuencia de escape? En estos casos debe saber que deberá utilizar una barra diagonal inversa doble (\\). 3

1

```
document.write("<p>Aún no conozco " + bestCountry + ". Si vuelves algún día, avísame ;-)</p>");
else {
document.write("<p>¡Sí! " + bestCountry + " también es muy bonito</p>");
```

2

```
texto1="\u00A9 Mediaactive"
// El significado de esta secuencia
de escape es el signo de copyright
seguido del texto Mediaactive
```

La secuencia de escape \u00A9 representa el símbolo de copyright.

3

```
document.write('La ruta de la imagen es C:\\proyecto\\imagenes\\imagen.gif.');
```

4. Los caracteres Unicode pueden ser especificados con la siguiente secuencia de escape:

   `\uhhhh`

5. El caracter `h` de esta secuencia representa un número hexadecimal de cuatro dígitos, teniendo en cuenta que una secuencia de escape Unicode puede representar cualquier caracter de 16 bits. Aun así, debe saber que existen secuencias de escape de un único caracter que representan, por ejemplo, caracteres de tabulación (`\t`) o de retroceso (`\b`).

6. Las secuencias de escape se dividen en categorías, concretamente en dos: espacios en blanco y terminadores de línea. A continuación le facilitamos en una tabla las secuencias de escape que pertenecen a la categoría de espacios en blanco:

\u0009	\t	Tabulador
\u000B	\v	Tabulador vertical
\u000C	\f	Avance de página
\u0020		Espacio
\u00A0		Espacio de no separación
\uFEFF		Marca de orden de bytes

8. Por su parte, incluimos las secuencias de escape que pertenecen a la categoría de terminadores de línea en la tabla siguiente:

\u000A	\n	Nueva línea
\u000D	\r	Retorno de carro
\u2028		Separador de línea
\u2029		Separador de párrafo

4

```
alert( ' Contrase\u00F1a err\u00F3nea ' );
// El texto que mostraría el cuadro de alerta
es "Contraseña Errónea"
```

5

```
alert(  ' Contrase\u00F1a err\u00F3nea\u0021 \n Int\u00E9ntalo de nuevo. ' );
// En este ejemplo, además de la cadena de texto "Contraseña errónea",
conseguimos un signo de exclamación con la secuencia de escape \u0021,
un salto de línea con \n y el texto "Inténtalo de nuevo."
```

Convertir valores en strings

COMO YA HEMOS APUNTADO EN MÁS de una ocasión, en JavaScript existen cinco tipos de datos distintos que pueden contener valores: cadenas de texto, números, booleanos, objetos y funciones. A su vez, existen en este lenguaje tres tipos de objetos: objetos (Object), fechas (Date) y matrices (Arrays). Y también existen dos tipos de datos que no pueden contener valores: null y undefined. Es posible convertir prácticamente cualquier valor en una cadena de caracteres, y en este ejercicio veremos distintas formas de hacerlo.

1. De forma predeterminada, los distintos tipos de valores existentes en JavaScript se convierten en cadenas de texto o strings del siguiente modo: los valores `undefined` y `null` adquieren el mismo valor, `'undefined'` y `'null'`; lo mismo ocurre con los valores booleanos `true` y `false`, que se convierten en `'true'` y `'false'`; los valores numéricos se convierten en el mismo número pero como cadena de texto, es decir, entrecomillado. 1
2. El caso de los objetos es un tanto diferente, por el hecho de que en primer lugar son convertidos en valores primitivos, los cuales, según el resultado, serán convertidos en string de un modo u otro.
3. Cabe advertir que las cadenas de texto no pueden convertirse en ellas mismas; en este caso, la conversión no es posible.

1

undefined	'undefined'
null	'null'
valor booleano	'true' o 'false'
valor numérico (por.ejemplo, 25)	'25'
objeto	ToPrimitive()
cadena de texto	No se puede convertir

039

4. ¿Cómo se convierte un valor en una cadena de texto o string? Existen en Javascript tres modos de llevar a cabo esta conversión de valores: utilizando el método `String(valor)`, llamado como función y no como constructor, mediante el método `''+valor` y siguiendo el método `.toString()`.

5. Veamos unos ejemplos para cada uno de estos métodos. El método `String(value)` necesita como parámetro únicamente el valor que se desea convertir: 2

```
String(true) == 'true'
String(85102) == '85102'
String(['lunes', 'martes', 'miercoles']) == 'lunes,martes,miercoles'
```

6. En este caso, `String()` es llamado como función, y no como constructor; en dicho caso, el método utilizado sería `new String()`. El método `.toString()` convierte exclusivamente valores numéricos en cadenas de texto para ser tratadas como tal. Este método tiene un parámetro opcional, correspondiente a la base sobre la cual debe realizarse la conversión. Esta base (radix) puede ser 0, 2, 8 o 16: 3

```
var num = 15;
var a = num.toString() == 15;
var b = num.toString(2) == 1111;
var c = num.toString(8) == 17;
var d = num.toString(16) == f;
```

7. El método de conversión `.toString()` es compatible con los principales navegadores web: Google Chrome, Internet Explorer, Firefox, Safari y Opera. 4

2

```
String(true)
// Se convierte en 'true'
String(85102)
// Se convierte en '85102'
String([ 'lunes', 'martes', 'miercoles' ])
// Se convierte en 'lunes,martes,miercoles'
```

3

```
var num = 15;
var a = num.toString() // Se convierte en 15;
var b = num.toString(2)  // Se convierte en 1111;
var c = num.toString(8)  // Se convierte en 17;
var d = num.toString(16)  // Se convierte en f;
```

Tenga en cuenta que el método `value.toString()` no funciona con los valores `undefined` y `null`.

Comparar cadenas de caracteres

IMPORTANTE

Recuerde que JavaScript dispone de dos tipos de operadores para determinar la igualdad entre valores. Los operadores denominados de igualdad y de desigualdad estricta (`===` y `!==`) y los denominados de igualdad y de desigualdad normal (`==` y `!=`). (Recupere el ejercicio 11 de este libro si necesita recordar esta información.)

EN UN EJERCICIO ANTERIOR PUDIMOS CONOCER un tipo de operadores denominados de comparación: <, <=, > y >=. Los operadores de comparación comparan sus operandos, los cuales pueden ser números, cadenas de texto, lógicos u objetos, y devuelven como resultado un valor lógico según si la comparación establecida es o no verdadera. Javascript dispone de otro método para realizar comparaciones entre valores: se trata en este caso del método `String.prototype.localeCompare()`.

1. En este ejercicio, recordaremos los aspectos más básicos de los operadores de comparación en Javascript e introduciremos un método distinto para comparar valores en este lenguaje de programación. Como se ha indicado en la introducción, los operadores de comparación son cuatro:

```
> (mayor que)
>= (mayor o igual que)
< (menor que)
<= (menor o igual que)
```

2. Aunque los operadores de igualdad también se consideran operadores de comparación. [1] Este tipo de operadores tienen una serie de características muy importantes a tener en cuenta. La primera de ellas es que distinguen entre mayúsculas y minúsculas. Para determinar si una letra es mayor o menor que otra, las mayúsculas se consideran menores que las minúsculas y las primeras letras del alfabeto son menores que las últimas: [2]

1

```
== // igual
!= // distinto
=== // estrictamente igual
!== // estrictamente distinto
> // mayor que
>= // mayor o igual que
< //menor que
<= //menor o igual que
```

2

```
'B' > 'A'  == true
// B es menor que A
'B' > 'a'  == false
// B no es menor que a
```

La distinción entre mayúsculas y minúsculas en las cadenas de caracteres es fundamental para obtener los resultados esperados.

```
'B' > 'A'  == true
'B' > 'a'  == false
```

3. La segunda de las características a tener en cuenta es que no permite el uso de acentos ni de diéresis. Esto significa que los resultados obtenidos en una comparación en la que aparecen estos signos ortográficos no serán los obtenidos:

```
'ä' < 'b'  == false
'é' < 'f'  == false
```

4. Pasemos ahora a tratar el segundo método para comparar cadenas de caracteres en JavaScript. Como hemos indicado en la introducción, hablamos del método `localeCompare()`. Este método suele proporcionar mejores resultados que los operadores de comparación, aunque tiene un pequeño problema: la incompatibilidad con algunos navegadores.

5. Este método determina si dos cadenas de caracteres son equivalentes en la configuración regional actual. ¿Esto qué significa? Que los resultados devueltos serán en función del criterio de ordenación de la configuración regional predeterminada del sistema: -1 si se ordena el primer operando antes del segundo; +1 si se ordena el primer operando después del segundo, y 0 si las dos cadenas son equivalentes. Aquí puede ver algunos ejemplos de uso de este método comparativo:

```
var caso1 = "def";
var caso2 = "abc"
document.write(caso1.localeCompare(caso2) + "<br/>");
var caso3 = "ghi";
document.write(caso1.localeCompare(caso3)+ "<br/>");
var caso4 = "def";
document.write(caso1.localeCompare(caso4));
```

3

```
'ä' < 'b'  == false
// Si el primer operando fuera 'a',
el resultado sería true
'é' < 'f'  == false
// Si el primer operando fuera
el resultado sería true
```

4

```
var caso1 = "def";
var caso2 = "abc"
document.write(caso1.localeCompare(caso2) + "<br/>");
// El resultado devuelto es 1
var caso3 = "ghi";
document.write(caso1.localeCompare(caso3)+ "<br/>");
// El resultado devuelto es -1
var caso4 = "def";
document.write(caso1.localeCompare(caso4));
// El resultado devuelto es 0
```

Combinar cadenas de caracteres

SI EN EL EJERCICIO ANTERIOR HEMOS podido ver diferentes maneras de comparar cadenas de caracteres, en estas páginas trataremos la combinación de estos elementos en JavaScript.

1. El operador + es una de las opciones más útiles para combinar cadenas de caracteres. La condición requerida por este operador es que al menos uno de los operandos que se desea combinar sea, precisamente, una cadena de caracteres (string):

```
if (numFav) {
document.write('<p>El número ' + numFav + ' es un buen número</p>');
}
```

2. En el ejemplo anterior, el texto que se escribirá en pantalla si se cumple la condición definida para numFav es "El número X es un buen número". Si necesita recordar toda la información referente al caso especial del operador + no dude en recuperar el ejercicio 13 de este libro.

3. El operador de asignación compuesto += es el más recomendable en aquellos casos en que se desea combinar diferentes fragmentos de texto en una misma variable. Imagine por un momento que dispone de una variable con el nombre `casol`; dicha variable tiene el valor "aparca":

```
var casol = 'aparca'
```

1

```
if (numFav) {
 document.write('<p>El número ' + numFav + ' es un buen número</p>');
}
for (var num=11; num<=20; num++) {
 document.write('<h3>Número ' + num + '<br></h3>');
}
```

+

2

El caso del operador +

4. Si utilizamos el operador compuesto += para combinar el valor de la variable con otro valor distinto al de la misma, por ejemplo, 'coches', obtendremos un nuevo valor para la variable caso1: " aparcacoches". El modo de utilizar el operador es la siguiente: 3

```
caso1 += 'coches' // devuelve la palabra 'aparcacoches'
```

5. Otro ejemplo del uso del operador += podría ser el siguiente: 4

```
var caso2 = '';
   caso2 += 'Había una vez ';
   caso2 += 3;
   caso2 += ' cerditos';
   caso2 'Había una vez 3 cerditos'
```

6. Aunque los intérpretes de JavaScript actuales utilizan internamente la técnica del operador + para la combinación de cadenas de caracteres, existe un tercer método para combinar cadenas de caracteres y convertirlas en una nueva cadena. Se trata de agrupar en primer lugar en una matriz (array) todos los fragmentos de texto que se desea combinar y después juntarlos. Veamos esta técnica en un nuevo ejemplo, en el cual utilizaremos los mismo valores que en el paso 5: 5

```
var matriz = [ ];
  matriz.push('Había una vez ');
  matriz.push(3);
  matriz.push(' cerditos');
  matriz.join('')
  // El resultado de la combinación será
     'Había una vez 3 cerditos'
```

3

```
var caso1 = 'aparca'
caso1 += 'coches'
// devuelve la palabra 'aparcacoches'
```

4

```
var caso2 = '';
caso2 += 'Había una vez ';
caso2 += 3;
caso2 += ' cerditos';
caso2 'Había una vez 3 cerditos'
```

6

```
var matriz = [ ];
matriz.push('Había una vez ');
matriz.push(3);
matriz.push(' cerditos');
matriz.join('')
// El resultado de la combinación será
   'Había una vez 3 cerditos'
```

Métodos del constructor String

LA FUNCIÓN STRING PUEDE SER LLAMADA de dos modos distintos: como función normal (`String()`), método que convierte el valor indicado en una cadena de caracteres primitiva, y como constructor (`new String()`), que crea una nueva instancia del valor contenido en String. Cuando se utiliza la función `String` como constructor, existen múltiples métodos específicos que proporcionan distintos resultados.

1. En este ejercicio trataremos algunos de los métodos del constructor String más utilizados y útiles en JavaScript. Empezaremos por el método más básico, el que genera una nueva instancia del valor contenido en String: `new String()`. Esta nueva instancia es un objeto que envuelve el mencionado valor. Veamos un ejemplo:

   ```
   typeof new String('mercado') == 'object'
   ```

2. Como puede comprobar, el valor generado no es una cadena de texto sino un objeto, el cual será tratado como un objeto de secuencia de caracteres, a diferencia de las primitivas String, las cuales son tratadas como código fuente. 1

3. El método estático `String.fromCharCode()` devuelve como resultado una cadena creada mediante el uso de valores Uni-

1

```
caso1 = "5 + 7";
// crea una primitiva de cadena
que dará como resultado el valor numérico 12
caso2 = new String("5 + 7");
// crea un objeto String
que dará como resultado "5 + 7"
```

2

```
String.fromCharCode(32,33,34)
// Este ejemplo devuelve la cadena "abc"
```

code especificados. Este método requiere como parámetro la secuencia de números con los valores Unicode: 2

```
String.fromCharCode(num1, ...,num N)
```

4. Si desea convertir una matriz de valores numéricos en una cadena de caracteres, puede utilizar el método descrito complementado con `.apply`: 3

```
String.fromCharCode.apply(null, [32, 33, 34])
```

5. Y si lo que pretende es realizar la operación inversa a la del método `String.fromCharCode()` sepa que dispone del método `String.prototype.charCodeAt()`. Concretamente, este método devuelve un número indicando el valor Unicode del caracter en el índice proporcionado: 4

```
cadena.charCodeAt(índice);
```

6. El valor contenido en el parámetro índice debe ser un número entero entre 0 y 1 menos que la longitud de la cadena especificada. Si usted opta por no especificar ningún valor como índice, se seleccionará como valor predeterminado el 0. En el caso en que el índice proporcionado no se encuentre entre 0 y 1 menos de la longitud de la cadena, el método `charCodeAt()` devolverá el valor NaN.

7. Como información a tener en cuenta acerca de este método string diremos que en la versión 1.2 de JavaScript dicho método devuelve un número que indica el valor de la hoja de códigos ISO-Latin-1 del caracter correspondiente al índice proporcionado. El rango de esta hoja de códigos va del 0 al 255, de los cuales los 127 primeros coinciden perfectamente con los 127 primeros de la hoja de códigos ASCII.

042

IMPORTANTE

El rango del código Unicode va del 0 al 1,114,111. Tenga en cuenta que los primeros 128 códigos Unicode coinciden exactamente con los códigos de caracteres de la codificación ASCII.

3

```
String.fromCharCode.apply(null, [32, 33, 34])
// Este ejemplo devuelve la cadena "abc"
```

El método `string.charCodeAt()` es el contrario al método `String.fromCharCode()`.

4

```
"ABC".charCodeAt(0)
// Este método devuelve el valor 65.
que es el valor Unicode para A.
```

El método `string.charCodeAt()` siempre devolverá un valor menor de 65.536.

Propiedades y otros métodos de String

STRING DISPONE BÁSICAMENTE DE DOS PROPIEDADES: string.length y string.prototype. En un ejercicio anterior, en el cual fueron tratados los objetos envolvente, tuvimos ocasión de conocer brevemente esta segunda propiedad, la cual cuenta a su vez con distintos métodos para las cadenas de texto. En este ejercicio veremos en qué consisten las propiedades `.length` y `.prototype` y mencionaremos y describiremos algunos de sus métodos.

1. La propiedad string.length representa la longitud de la cadena de caracteres indicada. En el caso en que la cadena de texto esté vacía, la propiedad `.length` devuelve el valor 0. Se trata de una propiedad inmutable. Veamos un ejemplo básico de uso de esta propiedad: 1

   ```
   'mercado'.length == 7;
   ```

2. Como puede ver, la propiedad devuelve como resultado el número de caracteres de la cadena "mercado", en este caso, 7.
3. La propiedad `string.prototype`, por su parte, representa el prototipo de la clase String y puede utilizarse para agregar propiedades o métodos a todas las instancias de dicha clase. Todos los métodos de las cadenas de texto primitivas se almacenan en la propiedad `string.prototype`. Veamos algunos de estos métodos para primitivas, no para instancias de String.

Si el índice proporcionado para el método `charAt()` está fuera del rango de caracteres, JavaScript devuelve como resultado una cadena vacía.

1

```
var caso1 = "Mercado";


console.log("La palabra Mercado tiene " + caso1.length + " caracteres");
// "La palabra Mercado tiene 7 caracteres"
```

El resultado que devuelve cada una de estas líneas es: "`El caracter en el índice x es X`", siendo X cada uno de los caracteres de la palabra Bravo.

2

```
var caso1="Bravo";

console.log("El caracter en el índice 0 es '" + caso1.charAt(0) + "'")
console.log("El caracter en el índice 1 es '" + caso1.charAt(1) + "'")
console.log("El caracter en el índice 2 es '" + caso1.charAt(2) + "'")
console.log("El caracter en el índice 3 es '" + caso1.charAt(3) + "'")
console.log("El caracter en el índice 4 es '" + caso1.charAt(4) + "'")
```

4. El método `String.prototype.CharAt()` devuelve el caracter en el índice especificado. Este índice debe ser un número entero entre 0 y 1 menos que la longitud de la cadena. [2] Los caracteres de cualquier cadena se indexan de izquierda a derecha, siendo el primer caracter 0 y el último `.length - 1`.
5. El método `String.prototype.charCodeAt()` devuelve un número que indica el valor Unicode del caracter en el índice proporcionado. Igual que en el método descrito en el punto anterior, dicho índice debe ser un número entero entre 0 y 1 menos que la longitud de la cadena, y si no se especifica ningún valor, se asigna el predeterminado 0. [3]
6. El siguiente método, `String.prototype.slice()`, extrae un fragmento de la cadena indicada y devuelve como resultado una cadena nueva, teniendo en cuenta que cualquier cambio que se realice en una no afectará a la otra. Este método trabaja con dos parámetros: el del fragmento inicial y el del fragmento final. El fragmento inicial es el índice sobre cero en el cual empieza la extracción, y el fragmento final es, lógicamente, el índice sobre cero en el que finaliza. [4]
7. El método `String.prototype.split()` divide un objeto String en un vector de cadenas separando la cadena inicial en subcadenas. Este método trabaja con dos parámetros: el separador, que es el caracter que se utiliza para la separación de la cadena, y el límite, que es un número entero que indica el máximo de divisiones que se deben realizar. El separador es considerado como una cadena o como una expresión regular. [5]

043

IMPORTANTE

El método `String.prototype.substring()` funciona de manera muy parecida al método descrito en este ejercicio `String.prototype.slice()`. Sin embargo, debido a que en ocasiones gestiona negativamente las posiciones de los caracteres, se recomienda utilizar siempre que sea posible el método `.slice()`.

3

```
"ABC".charCodeAt(0)
// Devuelve el valor 65
```

4

```
var caso1 = "El mercado está abiertos.";
var caso2 = cadena1.slice(3, -2);
función print(caso2);
// La función print mostraría en pantalla:
"mercado está abierto"
```

5

```
var dias = "Lunes ;Martes; Miércoles ; Jueves ;Viernes ";
document.write(dias + "<br>" + "<br>");
var separador = /\s*;\s*/;
var listadias = dias.split(separador);
document.write(listadias);
// La cadena resultante después de la aplicación del método .split()
es: Lunes,Martes,Miércoles,Jueves,Viernes
```

Métodos para transformar strings

LA PROPIEDAD STRING.PROTOTYPE() DISPONE DE UNA serie de métodos que permiten transformar una cadena en otra. En este ejercicio trataremos algunos de estos métodos y daremos así por terminada la serie de ejercicios dedicados a las cadenas de caracteres.

1. Todos los métodos que permiten transformar cadenas de caracteres funcionan básicamente de la misma forma: la cadena original queda apartada (no desaparece) después de que hay sido transformada. Veamos algunos de estos métodos, en qué consisten y cómo funcionan.
2. El método `String.prototype.trim()` elimina cualquier espacio en blanco existente al inicio o al final de la cadena de caracteres. Entendemos por espacios en blanco en este caso cada uno de los caracteres en blanco (espacios, tabulaciones, etc.), así como todos los caracteres terminadores de línea (LF, CR, etc.). 1
3. Otro método de transformación de strings es `String.prototype.concat()`. Este método combina el texto de dos o más cadenas o strings y devuelve una cadena distinta. Los parámetros necesarios para que este método funcione son las dos cadenas que se desean combinar. Es preciso saber que los cambios efectuados en una cadena no afectan para nada a la nueva cadena. 2

1

```
var caso1 = '   foo  ';
console.log(caso1.trim());
// El resultado es una nueva cadena
con este contenido: 'foo'
```

El método `trim()` no afecta el valor mismo de la cadena.

2

```
caso1="Mañana "
caso2="será "
caso3="otro día'."
caso4=caso1.concat(caso2,caso3)
// La combinación de las tres cadenas
en caso4 devuelve la nueva cadena "Mañana será otro día'."
```

4. Si lo que necesita es obtener una misma cadena de caracteres en minúsculas, el método que debe utilizar para ello es `String.prototype.ToLowerCase()`. Este método no afecta al valor de la cadena en sí misma y su uso es realmente sencillo: 3

```
var texto="EJERCICIO"
document.write(texto.toLowerCase())
```

5. El método `String.prototype.ToLocaleLowerCase()` funciona del mismo modo que el método `ToLowerCase()`, descrito en el punto anterior, con la única diferencia que respeta las normas de la configuración regional actual del usuario.

6. El método `String.prototype.ToUpperCase()` realiza la transformación inversa a la que efectúa el método `ToLowerCase()`, esto es devolver el valor convertido en mayúsculas de la cadena que realiza la llamada. Este método no afecta al valor de la cadena en sí mismo. Para comprobar su funcionamiento, podemos utilizar el mismo ejemplo que hemos visto en el paso 4: 4

```
var texto="ejercicio"
document.write(texto.toUpperCase())
```

7. Por último, y al igual que para el método `ToLowerCase()`, existe el método `String.prototype.toLocaleUpperCase()`, el cual funciona del mismo modo que el método `ToUpperCase()`, con la única diferencia que respeta las normas de la configuración regional actual del usuario.

3

```
var texto="EJERCICIO"
document.write(texto.toLowerCase())
//La nueva cadena devuelta será "ejercicio"
```

4

```
var texto="ejercicio"
document.write(texto.toUpperCase())
//La nueva cadena devuelta será "EJERCICIO"
```

Buscar, comparar y comprobar strings

LA PROPIEDAD STRING.PROTOTYPE() TAMBIÉN CUENTA CON una serie de métodos destinados a buscar y comparar strings (`String.prototype.indexOf()`, `String.prototype.lastIndexOf()` y `String.prototype.localeCompare()`), así como otros que se utilizan con expresiones regulares (`String.prototype.search()`, `String.prototype.match()` y `String.prototype.replace()`).

1. En primer lugar, trataremos los métodos de la propiedad `String.prototype` que se utilizan para buscar y comparar cadenas de caracteres. Empezamos por el método `String.prototype.indexOf()`, el cual devuelve como resultado el valor de índice de la primera coincidencia del valor especificado. La búsqueda empieza desde el valor especificado como segundo parámetro. 1
2. El valor predeterminado de este índice de búsqueda es 0, pero puede ser un número entero entre 0 y la longitud de la cadena de caracteres.
3. El método `String.prototype.lastIndexOf()` funciona del mismo modo que el descrito en los pasos anteriores, con la diferencia que la búsqueda empieza siempre por el final de la cadena de caracteres. 2
4. El método `String.prototype.localeCompare()` se utiliza para comparar dos cadenas de caracteres en una misma configuración regional. Este método devuelve un valor numérico

1

```
var caso1="Los tres cerditos"
document.write("<P>El valor de índice de la primera letra s es " +
   caso1.indexOf("s"))
// El valor devuelto será 2
```

Tanto `indexOf()` como `lastIndexOf()` distinguen entre mayúsculas y minúsculas, aspecto que deberá tener en cuenta cuando escriba los caracteres o las cadenas de caracteres que desea buscar.

2

```
var caso1="Los tres cerditos"
document.write("<P>El valor de índice de la primera palabra tres
empezando por el final es " +
   caso1.indexOf("tres"))
// El valor devuelto será 9
```

que indica si la cadena de referencia se sitúa antes, después o al mismo nivel de ordenación que aquella con la cual se compara.

5. Los valores devueltos pueden ser tres: -1 si la cadena de referencia se ordena antes que la cadena comparada; 0 si ambas cadenas son iguales, y 1 si la cadena de referencia se ordena después de la cadena comparada. 3
6. En cuanto a los métodos de la propiedad `String.prototype()` que trabajan con expresiones regulares, empezamos refiriéndonos al método `String.prototype.search()`. Este método se utiliza para ejecutar una búsqueda entre una expresión regular indicada y el objeto String, sobre todo en aquellos casos en que se desea detectar si existe un patrón específico en una cadena. (Las expresiones regulares en JavaScript se tratarán más adelante en este libro.)
7. El método `String.prototype.match()` se utiliza para devolver las coincidencias existentes entre una cadena de caracteres indicada y una expresión regular concreta. Es posible utilizar flags o indicadores para advertir ciertos requisitos en el momento de la búsqueda. 4
8. El último método que deseamos mencionar en este ejercicio es `String.prototype.replace()`, con el cual es posible detectar coincidencias entre una expresión regular y una cadena especificada, reemplazando dicha coincidencia por una nueva cadena. 5

045

IMPORTANTE

En el método `String.prototype.replace()`, la cadena de reemplazo puede contener patrones de reemplazo especiales. Algunos de ellos son: `$$`, para insertar el símbolo `$`; `$&`, para insertar la subcadena coincidente, o `$'`, que inserta la parte de la cadena que precede a la subcadena coincidente.

3

```
var caso1 = "fg";
var caso2 = "hi";
var caso3 = caso1.localeCompare(caso2);
// El valor devuelto será -1, porque
la primera cadena se ordena antes de la segunda
```

4

```
var texto = "Si lo desea, consulte el Ejercicio 53";
var expresion = /(ejercicio)/i;
var resultado = texto.match(expresion);
console.log(resultado);
// El valor devuelto será Ejercicio
```

5

```
var texto1 = "Para que todos los días sean navidad...";
var texto2 = texto1.replace(/navidad/i, "Navidad");
print(texto2);
// El resultado del reemplazo será:
"Para que todos los días sean Navidad..."
```

Sentencias de bucle

EN JAVASCRIPT, ENTENDEMOS POR BUCLE AQUELLA serie de comandos que se ejecutan hasta conseguir una condición especificada. Las sentencias de bucle son las partículas que se utilizan en el código para definir este tipo de sentencias: for, while y do while, entre otras.

1. En este ejercicio trabajaremos con algunas de las sentencias de bucle en JavaScript. Empezaremos por una de las más conocidas: la sentencia de bucle `for`. Un bucle iniciado con esta sentencia se repite hasta que una condición específica se evalúa como falso. La sintaxis de la sentencia de bucle `for` es la siguiente:

```
for ([Expresióninicial]; [condición]; [incrementodelaExpresión])
sentencia
```

2. La sentencia de bucle `do...while`, por su parte, repite sentencias hasta que una condición se evalúa como falsa. La sintaxis de este tipo de sentencias es la siguiente:

```
do
  sentencia
while (condición);
```

3. En este caso la sentencia especificada se ejecuta como mínimo una vez antes de que la condición sea verificada. Si la condición se evalúa como true, la sentencia se ejecuta una vez más.

1

```
function cuest() {
 for (var i=0; i<preguntas.length; i++) {
  askQuestion(aleat);
  aleatorio = Math.floor(Math.random() *preguntas.length);
  aleat = preguntas[aleatorio];
 }
 var mensaje = 'Has contestado correctamente ' + puntos;
 mensaje += ' de ' +preguntas.length;
 mensaje += ' preguntas';
 var respuestas = document.getElementById('resultados');
 respuestas.innerHTML = mensaje;
}
```

2

```
for (var i=0; i<preguntas.length; i++) {
```

3

```
do {
   i += 1;
   document.write(i);
} while (i < 8);
```

El bucle `for` en JavaScript es similar al mismo bucle en otros lenguajes de programación, como Java y C.

En el caso contrario, es decir, cuando la condición es falsa, la ejecución pasa a la siguiente sentencia situada después de `do...while`.[3]

4. Otra de las sentencias de bucle más utilizadas y conocidas es while, la cual ejecuta sus sentencias mientras una condición indicada se evalúe como verdadera. En el caso en que la condición sea falsa, la sentencia incluida en el bucle detiene la ejecución y el control pasa a la sentencia siguiente situada después del bucle.[4]

5. La sentencia de bucle while tiene la siguiente sintaxis:

```
while (condición)
  sentencia
```

6. Además de las sentencias de bucle, los bucles aceptan una serie de mecanismos especiales, como son las etiquetas o las sentencias especiales `break` y `continue`. Las etiquetas proporcionan a la sentencia un identificador para poder referirse a ella en otra parte del script.[5] La sintaxis de las etiquetas es sencillamente la siguiente:

```
etiqueta  :
 sentencia
```

7. Por último, la sentencia `break` se utiliza para terminar una sentencia del tipo `loop`, `switch` o `label`[6] y la sentencia `continue` se utiliza para reiniciar sentencias del tipo `while`, `do...while`, `for` o `label`.[7]

046

IMPORTANTE

El valor de la etiqueta puede ser cualquier identificador de JavaScript, siempre y cuando no sea una palabra reservada.

4

```
caso1 = 0;
caso2 = 0;
while (caso1 < 3) {
   caso1++;
   caso2 += caso1;
}
```

En este ejemplo, el bucle se repite una y otra vez hasta que el valor `caso1` sea menor que 3.

5

```
etiqueta:
while (caso1 == true)
   función();
}
```

6

```
for (i = 0; i < a.length; i++) {
   if (a[i] == valor)
      break;
}
```

Ejemplo de sentencia `break`.

7

```
i = 0;
n = 0;
while (i < 5) {
   i++;
   if (i == 3)
      continue;
   n += i;
}
```

Ejemplo de sentencia `continue`.

Sentencias condicionales

IMPORTANTE

Resulta importante no confundir los valores lógicos primitivos true y false con los valores verdadero y falso de los objetos lógicos booleanos. Todo valor distinto a `undefined`, `null`, `0`, `NaN` o `""` (cadena vacía), así como cualquier objeto, devuelve `true` cuando se ejecuta en una sentencia condicional.

LAS SENTENCIAS CONDICIONALES SON CONJUNTOS DE comandos que se ejecutan sólo si la condición especificada resulta verdadera (`true`). JavaScript cuenta con dos tipos de sentencias condicionales: `if...else` y `switch`. En este ejercicio veremos todos sus detalles y su funcionamiento.

1. Empezamos este ejercicio quizás con la sentencia condicional más conocida: la formada por las cláusulas `if...else`. De hecho, la sentencia condicional está formada inicialmente sólo por la cláusula `if`. `if` se utiliza para ejecutar sentencias siempre y cuando la condición lógica sea verdadera. [1] En aquellos casos en que la condición lógica se prevea falsa, la sentencia `if` se complementará con la cláusula `else`. [2]

2. La sintaxis de las sentencias condicionales `if...else` tiene el siguiente aspecto:

```
if (condición)
  sentencia1
else
  sentencia2
```

3. Según esta sintaxis, si la condición devuelve `true`, la `sentencia1` se ejecuta; si no, se ejecutará la `sentencia2`. Dichas sentencias pueden ser de cualquier tipo, incluyendo otras sentencias `if`, las cuales se consideran en estos casos anidadas.

4. Cuando necesite varias condiciones en una secuencia e ir probando su ejecución una tras otra, puede optar por utilizar también la sentencia `else if...` [3] En estos casos, es recomenda-

1
```
var diaSemana=fecha.getDay()-1;
  if (diaSemana <0){
       diaSemana=6
   }
```

2
```
var hora = new Date();
var horas = hora.getHours();
 if (horas<12) {
  formatoHora = 'am';
 } else {
  formatoHora = 'pm';
 }
```

3
```
if (condición)
   sentencia1
[else if (condición2)
   sentencia2]
...
[else if (condición3)
   sentencia3]
[else
   sentencia4]
```

ble para una mejor organización utilizar bloques de sentencias (`{...}`).[4]

5. Dado que las asignaciones simples pueden ser confundidas con las expresiones de igualdad, se recomienda no utilizar asignaciones de este tipo en las expresiones condicionales. En el caso en que sea necesario escribirlas, hágalo entre paréntesis. [5]

6. Hablemos ahora de la sentencia condicional `switch`. Este tipo de sentencia permite a un script evaluar una expresión especificada y compararla con el valor indicado en la etiqueta de una expresión por casos. Si dicha comparación es positiva, es decir, si se encuentran coincidencias, el script pasará a ejecutar la sentencia asociada.

7. La sintaxis de la sentencia condicional `switch` tiene la siguiente estructura:

```
switch (expresión) {
  caso etiqueta1:
      sentencias1
      [break;]
   caso etiqueta2:
      sentencias2
      [break;]
   ...
   default:
      sentencias_predeterminadas
      [break;]
}
```

8. La sentencia `break`, mencionada en el ejercicio anterior, es opcional y se encuentra asociada con cada cláusula de caso y asegura que el script omita la sentencia `switch` al ejecutarse la sentencia coincidente y continúe con la siguiente sentencia después de dicha sentencia condicional.

4

```
if (bestCountry == 'Canadá') {
  document.write("<p>¡Coincido contigo! Canadá también es mi país favorito</p>");
} else if (bestCountry == 'Argentina' || bestCountry == 'Portugal') {
  document.write("<p>Aún no conozco " + bestCountry + ". Si vuelves algún día, avísame ;-)</p>");
} else {
  document.write("<p>¡Sí! " + bestCountry + " también es muy bonito</p>");
}
```

5

```
if ((x = y)) {

}
```

Es recomendable escribir entre paréntesis las asignaciones simples en una sentencia condicional.

El caso de la sentencia with

IMPORTANTE

La sentencia `debugger` de JavaScript se utiliza para llamar o invocar cualquier funcionalidad de depuración en un script. En el caso de no existir ninguna funcionalidad de este tipo, la sentencia no tiene efecto alguno.

LA SENTENCIA WITH ESTÁ CONSIDERADA UN caso especial en JavaScript. Si bien muchos desarrolladores recomiendan evitar su uso (en estas páginas explicaremos el porqué), los usuarios procedentes de otros lenguajes de programación la suelen utilizar como una estructura normal.

1. En este ejercicio trataremos el caso de la sentencia `with`. Empezaremos describiendo su función en JavaScript. La sentencia `with` es un elemento para la manipulación de objetos que establece el objeto predeterminado para un conjunto de sentencias. JavaScript busca por cualquier nombre no calificado dentro de un grupo de sentencias y determina si dichos nombres son propiedades del objeto por defecto.
2. La sintaxis de una sentencia with es aproximadamente como sigue:

```
with (objeto) {
 conjunto de sentencias
}
```

3. El resultado de la ejecución de la sentencia `with` es que convierte en variables locales para la `sentencia` las propiedades del elemento `objeto`:

```
var objeto = { primero: 'Marcos' };
with (obj) {
console.log('Hola '+primero);
}
```

1

```
var a, x, y;
var r = 10;
with (Math) {
   a = PI * r * r;
   x = r * cos(PI);
   y = r * sin(PI/2);
}
```

2

```
with (objeto) {
 conjunto de sentencias
}
```

Sintaxis de la sentencia `with`.

3

```
var objeto = { primero: 'Marcos' };
with (obj) {
console.log('Hola '+primero);
}
// El resultado será "Hola Marcos"
```

4. Además de esta aplicación, la sentencia with también se suele utilizar para la asignación de distintos atributos a un conjunto de elementos DOM específicos. El siguiente script es un ejemplo de esta aplicación: 4

```
with(document.getElementById('el').style) {
    backgroundColor = '#125';
    color = '#ffc';
    width = '300px';
    padding = '60px';
}
```

5. Como hemos indicado en la introducción, muchos desarrolladores evitan el uso de la sentencia `with` en JavaScript, llegando incluso a desaconsejarlo. ¿Por qué esta desaprobación? La razón debemos buscarla en tres aspectos: comportamientos inesperados, rendimiento y seguridad.
6. Como ya sabe, la sentencia `with` no permite agregar nuevas propiedades a los objetos, lo que significa que el resultado de esta práctica será la generación de nuevas variables globales independientes. El problema de esta nueva generación de variables es que son difíciles de rastrear y, por tanto, pueden producir comportamientos inestables en el script.
7. El hecho de que el intérprete de Javascript de cada navegador tenga que realizar múltiples llamadas antes de descubrir si un elemento es una propiedad o una variable real provoca una pérdida de rendimiento importante del sistema, la segunda de las razones por la cuales se desaprueba el uso de la sentencia `with`.
8. Y aproximadamente esta explicación es la misma para el tercer motivo, la seguridad.

IMPORTANTE

Según Brendan Eich, la seguridad es el principal motivo por el cual no se debe utilizar la sentencia `with` en JavaScript. De hecho, dicha sentencia ya no aparece en versiones estándar del lenguaje ECMAScript (recuerde, la base de JavaScript).

4

```
with(document.getElementById('el').style) {
    backgroundColor = '#125';
    color = '#ffc';
    width = '300px';
    padding = '60px';
}
//Cabe señalar que este uso específico de la sentencia with
se parece a una estructura de jQuery utilizada
para dar estilo a los elementos.
```

Gestionar excepciones en JavaScript

IMPORTANTE

Existen dos aspectos a tener en cuenta en la gestión de excepciones en JavaScript. El primero es que si se detecta un problema que no puede ser gestionado al instante en que se produce, es recomendable lanzar una excepción. El segundo, localizar el punto del script en el cual el error puede ser gestionado, mediante la captura de excepciones.

EN JAVASCRIPT, LA GESTIÓN O EL manejo de excepciones trabaja generalmente con grupos de sentencias que se encuentran relacionadas o acopladas. Si en el momento de ejecutar esta sentencias, una de ellas produce algún error, ya no se continuará con el resto de sentencias del grupo. Las excepciones son las que entrarán en juego en este punto para intentar recuperar aquellas sentencias del grupo que nos interesa ejecutar.

1. La gestión de excepciones en JavaScript funciona como en la mayoría de los lenguajes de programación: se agrupan las sentencias dentro de la cláusula `try` permitiendo interceptar excepciones en dichas sentencias. 1
2. Para lanzar una excepción, JavaScript cuenta con la sentencia `throw`, mientras que para manipularla se utiliza la sentencia `try...catch`. Veamos cómo funciona cada una de ellas.
3. La sentencia `throw` lanza una excepción definida por el usuario. Esta excepción puede contener valores de todo tipo que condicionarán el resultado de la gestión. Así, por ejemplo, podemos encontrar excepciones lanzadas de la siguiente manera: 2

```
throw "error4";
throw 35;
throw false;
```

1

```
function processFiles() {
 var fileNames = collectFileNames();
  var entries = extractAllEntries(fileNames);
   processEntries(entries);
}
function extractAllEntries(fileNames) {
var allEntries = new Entries(); fileNames.forEach(function (fileName) {
var entry = extractOneEntry(fileName);
allEntries.add(entry); }); //Esta sentencia no podrá ejecutarse
debido al error producido en el grupo siguiente
}
function extractOneEntry(fileName) {
var file = openFile(fileName); // Éste es el grupo de sentencias
                               en el cual se produce el error.
... }
...
```

En este script, no se ha aplicado ninguna gestión de excepciones, lo que significa que el grupo de sentencias que produce el error no permitirá que se ejecute el resto de sentencias.

4. Si al lanzar la excepción se decide especificar un objeto, es posible referenciar las propiedades de dicho objeto en un bloque encabezado por la sentencia `catch`. Si dichas propiedades son capturadas por la mencionada sentencia, se ejecutará la función indicada. Mire como ejemplo el siguiente script, en el cual se utilizan las dos (o tres) sentencias para la gestión de excepciones mencionadas anteriormente:

```
var caso1=prompt("Escriba un valor del 0 al 10:","");
try
  {
  if(caso1>10)
    {
    throw "error1";
    }
  else if(caso1<0)
    {
    throw "error2";
    }
  else if(isNaN(caso1))
    {
    throw "error3";
    }
  }
catch(er)
  {
  if(er=="error1")
    {
    alert("El valor debe ser mayor que 10");
    }
  if(er=="error2")
    {
    alert("El valor debe estar entre 0 y 10");
    }
  if(er=="error3")
    {
    alert("Esto no es un número...");
    }
  }
```

049

IMPORTANTE

La sentencia para la gestión de excepciones `try... catch` delimita un bloque de las sentencias que se deben comprobar y especifica una o más respuestas cuando se produce la excepción. Es decir, es la sentencia `try...catch` la que se ocupa de la excepción.

2

```
throw "error4";
// genera una excepción con un valor del tipo string
throw 35;
// genera una excepción con un valor del tipo numérico
throw false;
// genera una excepción con un valor del tipo booleano
```

Tenga en cuenta siempre escribir la sentencia `throw` en minúsculas; de lo contrario, el script generará un error de JavaScript.

Crear un objeto de error

IMPORTANTE

Existe un séptimo constructor de errores que no hemos incluido en la lista de este ejercicio debido a que ya no se utiliza actualmente; se trata del constructor `EvalError`, el cual indica un error relacionado con la función global `eval()`.

EXISTEN EN JAVASCRIPT CONSTRUCTORES DE ERRORES específicos. Cabe señalar que estos constructores fueron estandarizados en su momento por la entidad de especificaciones ECMAScript. En este ejercicio se describirá cuáles son estos constructores de error y cómo se utilizan en un script.

1. En total, son seis los constructores de errores estandarizados por ECMAScript: `Error`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError` y `URIError`. Veamos en qué consiste cada uno de ellos.
2. El constructor `Error` es un constructor genérico de errores. De hecho, el resto de constructores son estrictamente subconstructores. El constructor Error crea un objeto de error; las instancias de este objeto son lanzadas cuando se produce el tiempo de ejecución del error.
3. El constructor `RangeError` se utiliza para especificar un valor numérico que ha sobrepasado un rango de valores permitido. Su sintaxis es idéntica a la del constructor genérico `Error`:

```
new RangeError([mensaje[, nombreArchivo[, numeroLinea]]])
//Sintaxis del constructor RangeError
```

4. El constructor `ReferenceError` se utiliza para indicar que ha sido detectado un valor de referencia no válido. Este valor es

1

```
new Error([mensaje[, nombreArchivo[, numeroLinea]]])
//Sintaxis del constructor genérico Error
```

2

```
var caso1 = function(num) {
  if (num < MIN || num > MAX) {
    throw new RangeError('El valor debe estar comprendido entre ' + MIN + ' y ' + MAX);
  }
};

try {
  caso1(500);
}
catch (e) {
  if (e instanceof RangeError) {
    // En este instante se ejecuta rangeError
  }
}
```

Compruebe como se complementan en el mismo script las sentencias `throw`, `try...catch` y el constructor `RangeError`.

en la mayoría de los casos una variable desconocida, no declarada:

```
try {
  var caso1 = variableNoDefinida;
} catch (e) {
  console.log(e instanceof ReferenceError); // true
  console.log(e.message); // "variableNoDefinida no
  está definida"
  console.log(e.name);// "ReferenceError"
  console.log(e.fileName);// "Scratchpad/1"
  console.log(e.lineNumber);// 2
  console.log(e.columnNumber);// 6
  console.log(e.stack);// "@Scratchpad/2:2:7\n"
}
```

5. El constructor `SyntaxError`, como puede seguramente deducir por su nombre, representa un error cuando intenta interpretar sintácticamente un fragmento de código no válido. Uno de los análisis que pueden ejecutar el constructor `SyntaxError`es mediante la función `eval()`.[3]
6. El siguiente constructor es `TypeError`. Este constructor lanza un error cuando un valor está escrito de un modo inesperado para el operador o la función a la que se refiere.[4]
7. Por último, el constructor de errores `URIError` indica que una de las funciones de manejo del código URI ha sido utilizado de un modo incompatible con su definición.[5] A título informativo diremos que el código URI se utiliza en Internet para escribir los identificadores de nombres (URN) y ubicaciones (URL). La sintaxis del constructor `URIError` es idéntica a la del resto de constructores definidos en este ejercicio.

[3]

```
eval('65 +')
 SyntaxError: Unexpected end of file
```

La sintaxis del constructor `SyntaxError` es la misma que para los constructor `RangeError` y `ReferenceError`.

[5]

```
decodeURI('%8')
  URIError: URI mal escrito
```

[4]

```
caso1.foo
   TypeError: Cannot read property 'foo' of caso1
```

Funciones en JavaScript

EN JAVASCRIPT, UNA FUNCIÓN ES UN procedimiento, es decir, un conjunto de declaraciones definidas para llevar a cabo una tarea específica o un cálculo de valores. Las funciones son uno de los pilares fundamentales en este lenguaje de programación. Las funciones deben ser definidas para, posteriormente, poder ser llamadas.

1. En este ejercicio vamos a ver distintos aspectos relacionados con las funciones en JavaScript. Antes de empezar debemos advertir que la definición de funciones será tratada en exclusiva en el ejercicio siguiente (ejercicio 52), por lo que en estas páginas sólo realizaremos una breve descripción de este proceso.
2. La declaración de una función es la manera en que ésta es definida. En esta declaración habrá una sentencia de retorno que devuelve un valor para la definición, [1] y si tal retorno no existe, se devolverá implícitamente el valor ya conocido undefined. [2]
3. En JavaScript, las funciones [3] pueden comportarse, una vez llamadas, de tres modos distintos: como una función normal, como constructor o como método. El modo en que es llamada la función es lo que determinada su comportamiento. Veamos brevemente en qué consiste cada uno de estos comportamientos.

1

```
function caso1(x) {
 devuelve x;
}
```

Aspecto básico de una declaración de función

2

```
function caso1() { }
caso1()
indefinido
```

3

```
<script type="text/javascript">
$(document).ready(function() {
 $('span.enf').each(function() {
  var cita=$(this).clone();
   cita.removeClass('enf');
   cita.addClass('enfasis');
   $(this).before(cita);
 });
});
</script>
```

051

4. Las funciones normales son aquellas que se llaman directamente. Por norma, las funciones normales empiezan siempre en minúsculas. En el ejemplo siguiente puede ver la definición de una función simple, a la que hemos denominado `cuadrado`:

```
cuadrado(valor) {
  valor * valor;
}
```

5. En este ejemplo, la función `cuadrado` toma un argumento, que hemos denominado `valor`. La sentencia de esta función podrí leerse como sigue: "devuelve el argumento de la función, `valor`, multiplicado por sí mismo". La sentencia `valor * valor`será la que mostrará el número devuelto por la función.
6. Las funciones también pueden comportarse como constructores si son llamadas mediante el operador `new`. Como ya hemos visto en alguna ocasión, este operador genera nuevos objetos. Cuando la función se comporta como constructor debe escribirse con la inicial en mayúsculas.
7. Por último, hemos indicado que las funciones pueden comportarse como métodos, y esto es así cuando la función puede almacenarse entre las propiedades de un objeto. En este caso, la función o método debe llamarse a través de dicho objeto. Tenga en cuenta que tanto las funciones comportadas como constructores y como métodos serán tratadas más adelante por separado en este libro.

IMPORTANTE

Existen dos términos relacionados directamente con las funciones que en JavaScript, en ocasiones, se utilizan de forma indistinta erróneamente: "parámetro" y "argumento". Sin embargo, cada una de ellas es definida de forma diferente: los **parámetros** se utilizan para definir una función, mientras que los **argumentos** se ocupan de llamarla.

4

```
function fechaCompleta(fecha,formato) {
 var diaSemana=fecha.getDay()-1;
   if (diaSemana <0){
       diaSemana=6
   }
```

Éste es un ejemplo de una función normal.

fecha.getDate()

6

```
var nombreDia=dias[diaSemana];
var dia=fecha.getDate();
var diaDosDigitos=dia;
   if (dia<10) {
       diaDosDigitos = '0' + dia;
   }
```

Las funciones que se comportan como método se escriben siempre con la inicial en minúsculas.

new Date();

5

```
var fechaActual = new Date();
var fechaFutura = new Date(2015,2,01);
var diferencia = fechaFutura.getTime() - fechaActual.getTime();
var totalDias = Math.round(diferencia/(1000 * 60 * 60 * 24));
document.write("¡Faltan " + totalDias + " días para la inauguración!");
```

Definición de funciones

EN JAVASCRIPT, EXISTEN TRES MÉTODOS PARA definir funciones: mediante una definición, mediante una expresión o mediante el constructor Function(). En este ejercicio veremos las características de cada uno de estos métodos, dejando para el siguiente la explicación de cuál es el mejor de ellos.

1. La declaración o definición de una función se realiza mediante la palabra clave Function seguida de el nombre de la función en sí (parámetro, en este caso, opcional), los argumentos para la función y las declaraciones JavaScript que definen la función.
2. El modo de distinguir los argumentos de las declaraciones de JavaScript son los signos entre los cuales se escriben en la definición: los argumentos se escriben entre paréntesis y separados por una coma, mientras que las declaraciones de JavaScript se encierran entre llaves ({}).
3. Los parámetros primitivos se pasan a las funciones por valor, lo que significa que si la función cambia el valor del parámetro, dicha modificación no queda reflejada de forma global o en llamadas distintas de la misma función.
4. En el caso de que se pase un objeto como parámetro y la función cambia las propiedades de dicho objeto, la modificación realizada se visualiza desde fuera de la función.

1

```
25  function askQuestion(pregunta) {
26   var respuesta = prompt(pregunta[0],'');
27    if (respuesta == pregunta[1]) {
28     alert('¡Correcto!');
29     puntos++;
30    } else {
31     alert('Lo sentimos. La respue
32    }
33  }
```

En la línea 25 puede ver la definición de la función con nombre (`function askQuestion`) y el argumento entre paréntesis (`pregunta`); desde la línea 26 hasta la 33 se encierran las declaraciones, encerradas entre llaves.

2

```
function funcion(objeto) {
  objeto.prop = "Seat";
}

var coche = {prop: "Peugeot", modelo: "308", año: 2002},
    ejemplo1,
    ejemplo2;

ejemplo1 = coche.prop;

funcion(coche);
ejemplo2 = coche.prop;
```

052

5. Como hemos indicado en la introducción de este ejercicio, las funciones también pueden ser creadas mediante una expresión de función. Esta expresión puede ser anónima, es decir, no tener nombre, o, por lo contrario, es posible proporcionar uno, en cuyo caso podrá ser utilizado dentro de la misma función o en otro proceso distinto. 3
6. La declaración de funciones mediante una expresión es recomendable cuando se pasa una función como argumento a otra función. 4
7. En JavaScript, una función puede ser definida basándose a una condición. Puede ver un ejemplo de ello en el siguiente script:

```
var funcion;
if (num == 0){
  funcion = function(objeto) {
    objeto.prop = "Peugeot"
  }
}
```

8. En este ejemplo, la definición de la función denominada `funcion`se realiza sólo si el valor de `num` es igual a `0`.
9. En cuanto a la definición de funciones mediante el constructor `Function()`, éste se utiliza para generar funciones de una cadena en tiempo de ejecución, del mismo modo que hace la función predefinida `eval()`. El constructor `Function` crea un nuevo objeto del tipo `Function`. De hecho, en JavaScript todas las funciones son realmente un objeto de este tipo.
10. Los objetos del tipo `Function` son analizadas cuando se crea la función. Cabe decir que este método no es tan eficiente como la declaración de funciones mediante expresiones o mediante definiciones.

3

```
var cuadrado = function(numero) {return numero * numero};
var caso1 = cuadrado(4)
```

En este ejemplo, la variable `caso1` devuelve como resultado el valor 16.

4

```
function mapa(f,a) {
  var resultado = [], // Se crea una matriz
      i;
  for (i = 0; i != a.length; i++)
    resultado[i] = f(a[i]);
  return resultado;
}
```

En este ejemplo, la función `mapa` se define a continuación y, posteriormente, es llamada con una función anónima como primer parámetro.

¿Declaración o expresión de funciones?

EN EL EJERCICIO ANTERIOR HEMOS PODIDO comprobar que JavaScript admite diferentes métodos para declarar o definir funciones. Teniendo en cuenta que el método constructor Function() es el menos utilizado debido a sus características, los desarrolladores tienden a inclinarse por los métodos de expresión o declaración. Sin embargo, ¿cuál de ellos es mejor? ¿Cuál es el más aconsejable?

1. En este ejercicio siguiente trataremos de averiguar cuál es la mejor elección para declarar funciones en JavaScript y por qué. Empezaremos por decir que una misma sentencia puede ser declarada mediante una definición de función y mediante una expresión de función. Un ejemplo de ello es la siguiente declaración de función:

```
function ejemplo(caso1) {
 return caso1;
}
```

2. Que puede ser también declarada mediante una variable y una expresión de función como sigue:

```
var ejemplo = function (caso1) {
 return caso1;
};
```

3. Dicen los expertos que la declaración de funciones tiene una serie de ventajas con respecto a la expresión de funciones; en concreto, dos, que son las siguientes: primero, que pueden ser llamadas antes de que hayan sido declaradas y segundo, que tienen nombre.

1

```
function ejemplo(caso1) {
 return caso1;
}

// Esta sentencia también
puede declararse como sigue:

var ejemplo = function (caso1) {
 return caso1;
};
```

4. Las declaraciones de función forman parte de un script global, lo que supone que son evaluadas antes que cualquier otra expresión de función, teniendo preferencia sobre cualquier expresión que la preceda, incluso si la declaración se encuentra al final del código fuente. Un ejemplo de esta característica es el siguiente script: 2

```
alert( suma( 7, 8 ) );
// El cuadro de alerta mostrará el valor 15
function suma( caso1, caso2 ){
  return caso1 + caso2;
}
```

5. La razón por la cual el código anterior funciona correctamente es que los intérpretes de JavaScript mueven la declaración de la función al inicio. Las declaraciones mediante variables también son movidas, pero sólo parcialmente, es decir, solamente la declaración, no la sentencia que debe llevarse a cabo con ella.

6. La segunda característica que convierten a las declaraciones de función en más recomendables que las expresiones es el hecho de poder poner un nombre a la función. La mayoría de los intérpretes de JavaScript admiten la propiedad no estándar name para objetos de función y las declaraciones de función tienen esta propiedad. En cambio, las expresiones de función son anónimas y su nombre es una cadena vacía. Por esta razón, siempre se les deberá asignar un nombre, sobre todo porque el nombre de la función resulta útil en el proceso de depuración del script. Un ejemplo de declaración de función con nombre es el siguiente: 3

```
function ejemplo() { }
ejemplo.name
'ejemplo'
```

IMPORTANTE

En inglés, la característica de las declaraciones de función mediante la cual estas sentencias forman parte de un script global y, por ello, pueden ser situados en cualquier punto del script, se denomina *hoisting*.

2

```
alert( suma( 7, 8 ) );
// El cuadro de alerta mostrará el valor 15
function suma( caso1, caso2 ){
  return caso1 + caso2;
}
```

3

```
function ejemplo() { }
ejemplo.name
'ejemplo'
// Declaración de función con nombre

var caso1 = function ejemplo() {};
caso1.name
'ejemplo'
// Expresión de función con nombre
```

Controlar parámetros nulos o extra

EN JAVASCRIPT, LAS FUNCIONES PUEDEN SER llamadas con cualquier número de parámetros reales, independientemente del número de parámetros formales que han sido definidos. Dicho esto, puede ocurrir que una función disponga de más parámetros reales que formales o bien que disponga de menos parámetros reales que formales. En este ejercicio veremos qué ocurre en cada uno de los casos.

1. En el caso en que existan más parámetros actuales que formales, los parámetros considerados extra no se tienen en cuenta, aunque pueden ser recuperados mediante un argumento específico para matrices (`argArray`). 1
2. Por el contrario, si son menos los parámetros reales que los formales, los parámetros formales desaparecidos tienen siempre el valor `undefined`.
3. La variable especial `arguments` sólo puede encontrarse dentro de las funciones, incluyendo también los métodos. Un objeto del tipo matriz es el que sostiene todos los parámetros reales de la llamada de función actual. 2
4. Las características de la variable especial del tipo matriz son las siguientes:

1

```
func(arg1, arg2, arg3)
func.apply(null, [arg1, arg2, arg3])
```

2

```
function logArgs() {
for (var i=0; i<arguments.length; i++) {
            console.log(i+'. '+arguments[i]);
        }
}
```

Este script refleja el uso de la variable especial denominada `arguments`.

- Sobre todo y ante todo, se trata de un elemento del tipo matriz, no de una matriz propiamente dicha. Esto significa que cuenta con la propiedad `length` y que los parámetros específicos pueden leerse y escribirse mediante valores de índice.

- La variable especial `arguments` no tiene métodos, aunque pueden tomarse prestados de una matriz o bien convertir las variables en matrices.

- La variable especial del tipo matriz es un objeto, lo que significa que puede utilizar cualquier método de objeto y cualquier operador disponibles.[3]

5. En cuanto a los parámetros desaparecidos o nulos, se pueden determinar las causas de esta omisión de tres modos distintos:

   - En primer lugar, comprobando si el parámetro es del tipo `undefined`;

   - En segundo lugar, interpretando el parámetro como booleano, en cuyo caso el parámetro `undefined` se considera `false`. Sin embargo, hay que tener en cuenta que muchos otros valores también pueden ser considerados `false`, por lo que no será posible distinguir entre el parámetro 0 y el desaparecido.

   - En tercer lugar, comprobando la longitud del parámetro `arguments` que asegure una aridad mínima.

6. En el caso en que un parámetro sea opcional, será preciso proporcionar un valor predeterminado si ha desaparecido.

3

```
function ejemplo() { return 1 in arguments } > ejemplo('caso1')
false
> ejemplo('caso1', 'caso2')
true
```

En este ejemplo, se utiliza el operador `in` para comprobar si el valor `arguments` cuenta con algún índice asignado.

4

```
var texto="ejercicio"
document.write(texto.toUpperCase())
//La nueva cadena devuelta será "EJERCICIO"
```

Parámetros con nombre

EN TODO LENGUAJE DE PROGRAMACIÓN, CUANDO se llama una función o un método, es preciso relacionar los parámetros reales con los parámetros formales. En este ejercicio veremos cómo llevar a cabo esta relación y estudiaremos la importancia y el funcionamiento de los parámetros con nombre.

1. La relación de los parámetros reales de una función con los parámetros formales puede llevarse a cabo de dos formas distintas: mediante la relación de los parámetros posicionales por posición y, en el caso de los parámetros con nombre, mediante nombres o etiquetas. Veamos en qué consisten ambos métodos.
2. Los parámetros posicionales son relacionados por posición. Esto significa que el primer parámetro real se relaciona con el primer parámetro formal, y así sucesivamente.
3. En el caso de los parámetros con nombre, dichos nombres están asociados con los parámetros formales en el caso de las definiciones de función y con las etiquetas de los parámetros reales en el caso de las llamadas de función.
4. Los parámetros con nombre proporcionan dos ventajas en el código: por un lado proporcionan descripciones para los argumentos en las llamadas de función y por otro, trabajan bien para los parámetros opcionales.

1

```
var valor = 1;
var string = "¡Hola!";
var booleano = true;

function valores(val, str, boo)
{
   document.write(val);
   document.write(str);
   document.write(boo);
}

valores(valor, string, booleano);
```

055

5. Como una función puede disponer de más de un parámetro, es probable que surja alguna duda acerca del uso de cada uno de estos parámetros. Gracias al lenguaje de programación interpretado Python, es posible utilizar parámetros con nombre para evitar estas dudas. Vea este ejemplo:

```
entradas(5, 14, 23)
```

6. Imagine que los valores (parámetros) que se encuentran entre paréntesis proceden de una selección de una base de datos. Es evidente que, a la hora de interpretar el código, no quede demasiado clara la procedencia de estos valores. Es en estos casos que los parámetros con nombre resultan de gran ayuda:

```
entradas(inicio=5, final=14, sección=23)
```

7. Es preciso tener en cuenta que JavaScript no admite parámetros con nombre, como en el caso de Python u otros lenguajes de programación. Sin embargo, cuenta con una buena solución para poder gestionar estos parámetros: utilizar un objeto literal para nombrar los parámetros y pasarlos como si fueran parámetros reales. Veamos esta solución aplicada en el ejemplo anterior:

```
entradas({ inicio: 5, final: 14, sección: 23 });
```

8. En este ejemplo, la función recibe un objeto con las propiedades `inicio`, `final` y `seccion`, pudiendo omitirlas cuando le convenga:

```
entradas({ sección: 5 });
entradas({ final: 14, start: 23 });
entradas();
```

9. Además de implementar entradas como se muestra en la imagen 2, también es posible combinar los parámetros de posición con los parámetros con nombre. Vea un ejemplo de este caso en la imagen 3.

IMPORTANTE

El lenguaje de programación **Python** fue creado a finales de los años ochenta en el Centro para las Matemáticas y la Informática, en los Países Bajos, para ser el sucesor del lenguaje de programación ABC e interactuar con el sistema operativo Amoeba.

2

Logotipo de la compañía desarrolladora Python Software Foundation.

3

```
function entradas(opciones) {
opciones = opciones || {};
var inicio = opciones.inicio || 0;
var final = opciones.final || getDbLength();
var seccion = opciones.seccion || 1;
...
}
```

Los ejemplos mostrados en los pasos 6, 7 y 8 pueden implementarse en un script de esta manera.

4

```
entradas(posArg1, posArg2, { nombreArg1: 7, nombreArg2: true });
```

Declaración de variables

EN JAVASCRIPT Y EN LA MAYORÍA de los lenguajes de programación, las variables están consideradas como uno de los elementos fundamentales para la realización de scripts. Aunque ya hemos podido ver en acción estos elementos en ejercicios anteriores, en éste describiremos en qué consisten las variables y cuál es el procedimiento que hay que seguir para declararlas.

1. Empecemos definiendo qué es una variable. Una variable es un espacio en memoria en el cual se almacena cualquier tipo de dato necesario para realizar las acciones de un script. El modo de acceder a cada una de las variables creadas es mediante su nombre. 1
2. Los nombres de las variables deben ser asignados por el desarrollador, siguiendo, eso sí, una serie de normas específicas. Por ejemplo, el nombre de una variable estará formado por caracteres alfanuméricos y opcionalmente el signo guión bajo (_), teniendo en cuenta que siempre debe empezar por una letra o el mencionado signo. 2
3. Otra norma a tener en cuenta es que los nombres de las variables no pueden contener signos aritméticos, así como otros tipos de signos o caracteres (**$**, @...).
4. Los nombres de las variables pueden estar formados por más de una palabra, en cuyo caso se puede proceder de dos formas:

1
```
var countries = ['Italia',
                 'Francia',
                 'Dinamarca',
                 'Reino Unido',
                 'Alemania',
                 'Rusia',
                 'Grecia',
                 'China'
                ];
var bestCountry = prompt('¿Cuál es tu país favorito?','');
```

3
```
var fechaActual = new Date();
var fechaFutura = new Date(2015,2,01);
var diferencia = fechaFutura.getTime()
```

2
```
var valor1 = '5';
var valor2 = '8';
```

Los nombres de las variables pueden contener letras y números, aunque siempre empezarán por un caracter alfabético.

escribiendo todas las palabras juntas, sin espacios y destacándolas mediante la inicial de cada una de ellas en mayúsculas, [3] o bien separándolas con el signo guión bajo. [4]

5. En cualquier caso, todas las variables se declaran mediante la sentencia `var` situada antes del nombre de variable. Una vez declarada la variable por completo, es decir, con la sentencia `var` y su nombre, es posible referirse a ella sin la sentencia, únicamente con el nombre. [5]

6. JavaScript, debido a su condición de lenguaje de programación un tanto "especial" permite pasar por alto la declaración de las variables; es decir, no es obligatorio (aunque sí recomendable) declarar la variable al inicio del script (de hecho, esta condición ya ha sido estudiada en un ejercicio anterior).

7. A continuación le mostramos tres formas de declarar y utilizar un par de variables: [6]

```
var caso1
var caso2
```

8. Así definimos la variable. Si deseamos asignarle los correspondientes valores, lo haremos de la siguiente forma:

```
var caso1 = 125;
var caso2 = 2548;
```

9. Y también es posible declarar las dos variables en una misma línea, en cuyo caso deberá escribirlas separadas por comas, como se muestra a continuación:

```
var caso1, caso2
```

[4]

```
function mapa(f,a) {
  var resultado_matriz = [],
      i;
  for (i = 0; i != a.length; i++)
    resultado_matriz[i] = f(a[i]);
  return resultado_matriz;
}
```

Es importante también evitar el uso de determinadas palabras reservadas en el lenguaje JavaScript como nombre para una variable, como `return` o `for`.

[5]

```
var a, x, y;
var r = 10;
with (Math) {
    a = PI * r * r;
    x = r * cos(PI);
    y = r * sin(PI/2);
}
```

[6]

```
var caso1
var caso2


var caso1 = 125;
var caso2 = 2548;


var caso1, caso2
```

Tres formas distintas de incluir variables en un script.

El ámbito de las variables

¿HASTA DÓNDE LLEGAN LAS VARIABLES DENTRO de un script? Con el ámbito de las variables queremos referirnos a aquellas ubicaciones del script desde las cuales se accede a cada una de las variables. En este ejercicio trataremos este concepto, así como otros aspectos relacionados con las variables en JavaScript.

1. Como hemos indicado en la introducción de este ejercicio, el ámbito de una variable son los puntos dentro del script desde los cuales se tiene acceso a ella. Por ejemplo, en el siguiente script: 1

```
function ejemplo() {
 var caso1;
}
```

   la ubicación directa de la variable `caso1` es la función `ejemplo`.

2. En JavaScript, la estructura estática de un script determina el ámbito de una variable; esto significa que no existe influencia alguna por parte del origen de la llamada de una función.

3. Puede ocurrir que el ámbito de una variable se encuentre anidado dentro de otro ámbito; en dichos casos, la variable será accesible en todas estas ubicaciones. Veamos un ejemplo de ello:

```
function ejemplo(arg) { function ejemplo2() {
  console.log('arg: '+arg);
}
ejemplo2(); }
console.log(ejemplo('Hola'));
```

IMPORTANTE

Además de cuanto hemos estudiado en este ejercicio, también podemos señalar, en relación al ámbito o la ubicación de las variables, que JavaScript desplaza las declaraciones de variables al inicio de su alcance directo. Este comportamiento permite entender qué sucede cuando se accede a una variable antes de que haya sido declarada:

```
function ejemplo() {
 console.log(caso2); // undefined
 var caso2 = 'hola';
 console.log(caso2); // hola
}
```

1

```
$(document).ready(function() {
 $('span.enf').each(function() {
  var cita=$(this).clone();
   cita.removeClass('enf');
   cita.addClass('enfasis');
   $(this).before(cita);
 });
});
```

2

```
var caso1, caso2


function ejemplo(arg) { function ejemplo2() {
  console.log('arg: '+arg);
}
ejemplo2(); }

console.log(ejemplo('Hola'));
```

4. En este caso, el alcance directo del parámetro `arges` la función `ejemplo`, teniendo en cuenta que también se puede acceder a ella desde la función `ejemplo2`.
5. Existe otro caso en el cual el nombre y el parámetro de las variables puede resultar alterado debido a la anidación de las funciones desde las cuales se asignan dichas variables. Si en un ámbito del script se declara una variable con un nombre que ya ha sido utilizado en otro punto del mismo, el acceso a la variable situada fuera de la función anidada queda bloqueado dentro de esta función. Aunque pueda parecer un poco confuso, vea este ejemplo para acabar de comprender este comportamiento: 3

```
var caso1 = "hola";
 function ejemplo() {
  var caso1 = "adiós";
  console.log(caso1);
}
ejemplo();
console.log(ejemplo);
```

6. Como puede ver, dentro de la función `ejemplo()` la variable `caso1` con el valor `hola` queda eclipsada por la variable `caso1` con el valor `adiós`.
7. Por último diremos que JavaScript es el único lenguaje de programación en el cual son las funciones las únicas que pueden presentar nuevas ubicaciones para las variables: 4

```
function ejemplo() {
 { // el bloque para la función ejemplo() empieza aquí
  var caso1 = 4;
 } // el bloque para la función ejemplo() termina aquí
 console.log(caso1); // El resultado del script es 4
}
```

3

```
var caso1 = "hola";
 function ejemplo() {
  var caso1 = "adiós";
  console.log(caso1);
}
ejemplo();
console.log(caso1);
```

4

```
function ejemplo() {
 { // el bloque para la función ejemplo() empieza aquí
  var caso1 = 4;
 } // el bloque para la función ejemplo() termina aquí
 console.log(caso1); // El resultado del script es 4
}
```

En este ejemplo, la variable `caso1` es accesible desde cualquier punto de la función `ejemplo`, no sólo desde dentro del bloque de dicha función.

Variables globales y locales

IMPORTANTE

Las variables locales se generan y se eliminan cada vez que se ejecuta una función y no es posible tener acceso a ella desde ningún código externo a dicha función.

UNA VARIABLE GLOBAL ES UNA VARIABLE que se declara fuera de la definición de una función, y su valor es accesible y modificable desde su script. En el caso en que la variable sea declarada dentro de una función, entonces dicha variable es local. JavaScript tiene dos ámbitos, el global y el local, y es acerca del uso y el significado de ambos que tratará este ejercicio.

1. Como hemos podido ver en el ejercicio anterior, hablamos de ámbito de las variables para indicar el punto del script en el cual están disponibles. En el caso de JavaScript, esto significa que las variables declaradas en una página web estarán disponibles dentro de la misma página, es decir, que no se puede tener acceso a ellas desde una página distinta.
2. Dentro del ámbito global, es posible crear otros ámbitos mediante la definición de funciones y, dentro de cada una de estas funciones, es posible generar otros ámbitos. Cada uno de los ámbitos tiene acceso a las variables declaradas en su interior, así como a aquellas que se encuentran en los ámbitos que lo rodean. Esto significa que, dado que el ámbito global rodea al resto de ámbitos existentes, es posible acceder a las variables incluidas en ellos desde cualquier lugar: 1

```
// Ámbito global
var variableGlobal = 'hola';
function ejemplo() {
```

1

```
// Ámbito global
var variableGlobal = 'hola';
function ejemplo() {
 var variableLocal = true;
 function ejemplo2() {
   //Es posible acceder a todas las variables de los ámbitos de alrededor
   var variableLocal2 = 'adiós';
    variableLocal = false;
           variableGlobal = 'hola';
       }
}
//Ámbito global
```

```
var variableLocal = true;
function ejemplo2() {
  var variableLocal2 = 'adiós';
   variableLocal = false;
          variableGlobal = 'hola';
      }
}
```

058

3. Las variables globales son aquellas que han sido declaradas en el ámbito más amplio posible, es decir, en JavaScript dentro de una misma página web. La declaración de variables globales se realiza simplemente entre etiquetas de script y con la sentencia `var`, como hemos visto hasta el momento: 2

```
<script type="text/javascript">
 var casol = 125;
</script>
```

4. Como ha podido comprobar en el script de ejemplo del paso 2, las variables locales son aquellas que se declaran en lugares más acotados, como es el caso de las funciones. El acceso a este tipo de variables sólo puede realizarse en el punto en el cual se han creado. Podrá reconocer las variables locales porque se escriben encerradas entre llaves ({ }). 3

5. Además de ser declaradas dentro de una función, como hemos visto en el ejemplo anterior, las variables locales también pueden ser declaradas en otros ámbitos, como por ejemplo en un bucle.

6. En el ejercicio anterior, ya apuntamos que los nombres de las variables globales y locales pueden coincidir, aunque habrá que tener en cuenta que la variable en vigor será la que se encuentre en cada uno de su ámbito (global o local).

IMPORTANTE

Las variables pueden declararse con o sin la sentencia `var`. Sin embargo, es importante saber que el resultado de uno u otro caso es distinto. Si utiliza `var`, la variable declarada será local, mientras que si no lo utiliza, la variable será global.

2

```
<script type="text/javascript">
    var numFav = prompt('¿Cuál es tu número favorito?');
</script>
```

Para evitar posibles errores, es recomendable no utilizar el mismo nombre para las variables globales y locales.

3

```
<script type="text/javascript">
  if(numFav == '7','5','3') {
  var colorFav = prompt('¡Igual! ¿Y cuál es tu color favorito?');
  if(colorFav == 'verde') {
     document.write('<p>¡Somos compatibles!</p>');
  } else {
     document.write('<p>Lástima...</p>');
    }
  }
</script>
```

Declaración de variables con var

COMO YA HEMOS INDICADO EN EL ejercicio anterior, JavaScript permite al desarrollador declarar variables utilizando o no la sentencia var. Sin embargo, es importante tener en cuenta que el significado resultante de este uso será distinto en cada caso.

1. En este ejercicio trataremos la declaración de variables globales y locales con el parámetro `var`. El uso de este parámetro convierte a la variable que estamos declarando en una variable local al ámbito en el cual es declarada. En cambio, si la declaración se realiza sin el parámetro `var`, la variable se convierte automáticamente en global a toda la página, sea cual sea el ámbito en el cual haya sido declarada.
2. En el caso de una variable global, es indiferente si se declara con el parámetro `var` o si se prescinde de él, precisamente debido a su condición de global. Esta diferencia podemos encontrarla sobre todo en una función; si utilizamos el parámetro `var` para declarar la variable, ésta será local a la función, mientras que si no lo hacemos, la variable será global a la página.
3. Es importante comprender esta diferencia para poder controlar sin problemas el uso de las variables en una página web: un tratamiento inadecuado puede suponer la sustitución de los datos contenidos en una variable.

```
1
var hora = new Date();
var horas = hora.getHours();
 if (horas<12) {
  formatoHora = 'am';
 } else {
  formatoHora = 'pm';
 }
 horas = horas % 12;
  if (horas==0) {
   horas = 12;
  }
```

4. Veamos todas estas consideraciones en un script:

```
<script>
var caso1 = 10
function ejemplo (){
    caso1 = 20
    document.write(caso1) //El resultado será 20
}
document.write(caso1) // El resultado será 10
// llamamos a la función
ejemplo()
document.write(caso1) // El resultado será 20
</script>
```

5. Analicemos este script. La variable `caso1` es del tipo global y contiene el valor 10. Seguidamente, la función `ejemplo()` utiliza la misma variable global `caso1` aunque sin el parámetro `var`. Esto mantiene la variable como global dentro de la función, y es por esta razón que el resultado que se mostrará en pantalla será 10, el valor contenido en la variable `caso1`.

6. En esta situación, cuando se ejecute la función `ejemplo()`, la variable `caso1` quedará sustituida y el dato contenido en ella se perderá.

7. Por todo cuanto hemos visto en éste y en el ejercicio anterior, los expertos desaconsejan la creación de nuevas variables globales y animan a trabajar siempre que sea posible con variables locales.

2

```
<script>
var caso1 = 10
function ejemplo (){
    caso1 = 20
    document.write(caso1) //El resultado será 20
}
document.write(caso1) // El resultado será 10
// llamamos a la función
ejemplo()
document.write(caso1) // El resultado será 20
</script>
```

Objetos sencillos

EN JAVASCRIPT, LOS OBJETOS SON RELACIONES entre cadenas y valores. Las entradas (también conocidas claves o valores) de los objetos se conocen denominan propiedades y el valor de una propiedad representa siempre una cadena de texto. El valor de una propiedad puede ser cualquier valor de JavaScript, incluyendo las funciones. Por último, podemos destacar que los métodos son propiedades cuyos valores son funciones.

1. En este ejercicio trataremos los objetos sencillos en JavaScript, sus características y su funcionamiento en los programas. Empezaremos detallando los tres tipos de propiedades que pueden contener los objetos. 1
2. El primer tipo son las propiedades propiamente dichas, es decir, las propiedades normales que todos conocemos, que son asignaciones desde claves de cadenas hasta valores. Entre estas propiedades de datos con nombre se encuentran los métodos.
3. El segundo tipo de propiedades de los objetos son métodos especiales cuyas llamadas que parecen propiedades de lectura y de escritura. Este tipo de propiedades permiten contabilizar los valores de las propiedades. 2
4. Por último, las propiedades internas, las cuales existen sólo en el lenguaje de especificaciones ECMAScript. JavaScript no tiene acceso a estas propiedades directamente, aunque sí puede

1

```
function Empleado () {
  this.nombre = "";
  this.dept = "general";
}
```

2

```
function Director () {
  this.reports = [];
}
Director.prototype = new Empleado;

function Adjunto () {
  this.projects = [];
}
Adjunto.prototype = new Empleado;
```

hacerlo de forma indirecta. La especificación de ECMAScript representa este tipo de propiedades entre claudátors (`[ ]`).

5. Una vez descritos los tipos de propiedades existentes para los objetos, vamos a dedicarnos a los objetos literales. En JavaScript, los objetos literales permiten la creación directa de objetos completos o, lo que es lo mismo, instancias directas del elemento `Object`. A continuación, presentamos un fragmento de código en el cual podrá identificar objetos, propiedades y métodos: 3

```
var caso1 = {
 name: 'Marta',

 describe: function () {
  return 'Usted es '+this.name;
 },
};
```

6. Este código utiliza un objeto literal para asignar un objeto a la variable que hemos denominado `caso1`. Dicho objeto tiene dos propiedades, `name` y `describe`, siendo ésta última también un método. La razón por la cual se ha utilizado la partícula `this` es la referencia al objeto actual. `this` se utiliza siempre en métodos.

7. Además de actuar como asignadores desde cadenas hasta valores, los objetos también admiten herencias entre ellos y pueden ser protegidos ante posibles cambios. La capacidad de crear directamente objetos es una de las características más relevantes de JavaScript. En este contexto, es posible empezar con objetos concretos e introducir posteriormente abstracciones.

3

```
var caso1 = {
 name: 'Marta',

 describe: function () {
  return 'Usted es '+this.name;
 },
};
```

Convertir valores en objetos

EN OCASIONES, AUNQUE NO SEA EL uso más frecuente, será necesaria la conversión de una serie de valores en objetos. Dicha conversión se realiza gracias al elemento `Object()` utilizado como función, y no como constructor. En este ejercicio conoceremos cómo se lleva a cabo esta conversión y cuáles son los resultados obtenidos según los datos de origen.

1. El uso del elemento `Object()` como función permite convertir cualquier tipo de valor en un objeto. Empecemos comprobando qué resultados obtenemos de esta conversión, según los tipos de datos evaluados. Vea para ello la tabla siguiente: 1

undefined	{ }
null	{ }
dato booleano	new Boolean(bool)
dato numérico	new Number(num)
cadena de caracteres	new String(str)

2. Si se intenta evaluar un objeto del tipo `Object()`, sepa que no se obtiene ningún resultado, puesto que la conversión dentro de un mismo tipo de datos no es posible. Recuerde que, tal y como vimos en un ejercicio anterior, la conversión entre tipos de datos se puede realizar mediante el operador `instanceof`. 2

1

undefined	{ }
null	{ }
true	new Boolean(bool)
123	new Number(num)
'hola'	new String(str)

2

```
Object(null) instanceof Object
// Devuelve true

Object(false) instanceof Boolean
// Devuelve true

var obj = {};
Object(obj) === obj
// Devuelve true
```

Estos son algunos ejemplos de los resultados obtenidos en un proceso de conversión

061

3. Si lo que necesita saber antes de proceder a la conversión es si un valor determinado es un objeto, siempre puede recurrir a la función `isObject()` o bien al operador `typeof()`, el cual también fue tratado en un ejercicio anterior.[3] Vea el uso de la mencionada función en el siguiente ejemplo:[4]

```
function isObject(valor) {
 return valor === Object(valor);
}
```

4. Tenga en cuenta que el uso de la función `isObject()` creará un nuevo objeto si el valor evaluado no lo es. Si a usted no le interesa que se genere un nuevo elemento del tipo Objeto, entonces realice la comprobación con el operador `typeof`.
5. En cualquier caso, el uso de `isObject()` devolverá como resultado el valor true, siempre que los datos evaluados sean objetos, matrices o funciones. Si no desea que las funciones se evalúen, deberá pasar como segundo parámetro el valor `true`, con lo que el intérprete de JavaScript no considerará las funciones como objetos.
6. El uso de objetos como constructores proporciona el mismo resultado que si se utilizan como funciones, como hemos visto hasta el momento. Aquí puede ver un ejemplo de ello:[5]

```
var obj = {};
new Object(obj) === obj
// devuelve true
new Object(123) instanceof Number
// Devuelve true
```

IMPORTANTE

Los expertos desarrolladores aconsejan no utilizar el elemento `Object()` como constructor, sino que si es posible siempre es mejor utilizar un objeto literal vacío: `var obj = {};`

3

```
document.write("<p>El tipo de boleano es: " + typeof boleano)
document.write("<p>El tipo de numerico es: " + typeof numerico)
document.write("<p>El tipo de texto es: " + typeof texto)
document.write("<p>El tipo de fecha es: " + typeof fecha)
```

Uso del operador `typeof` para comprobar el tipo de dato que es un valor determinado.

4

```
function isObject(valor) {
 return valor === Object(valor);
}
```

5

```
var obj = {};
new Object(obj) === obj
// devuelve true
new Object(123) instanceof Number
// Devuelve true
```

Uso de `Object()` como constructor para convertir datos en objetos.

El parámetro this en funciones y métodos

EN ESTE EJERCICIO, TRATAREMOS, ENTRE OTROS parámetros, el uso de `this` como parámetro implícito en funciones y métodos. Cuando llamamos una función, el parámetro `this` siempre se presenta como un parámetro, aunque sea implícito.

1. Antes de empezar, queremos realizar un apunto acerca del valor `this` en JavaScript. Este valor está considerado como complejo dentro del lenguaje de programación que estamos tratando, puesto que es capaz tanto de provocar comportamientos inesperados en la interpretación del código como producir grandes estructuras de código reutilizable. El valor de `this` queda determinado por la forma en que sea invocada la función en que se encuentra. Por todo ello, un uso correcto y apropiado de este valor resulta esencial para la programación orientada a objetos.
2. Dicho esto, pasaremos a definir el parámetro que nos ocupa. `this` es normalmente un parámetro cuyo valor se refiere al propietario de la función que lo está invocando; si no existe tal función, se refiere al objeto en el cual la función es un método.
3. En el caso de no formar parte de una estructura definida, el parámetro `this` se comporta como un objeto con métodos, puesto que el propietario de una función siempre es el ámbito

1

```
$(document).ready(function() {
 $('#bloque').each(function() {
  $(this).hover(
   function() {
    $(this).addClass('hoverBloque');
    status=$(this).find('a').attr('href');
   },
   function() {
    $(this).removeClass('hoverBloque');
    status='';
   });
  $(this).click(function() {
   location = $(this).find('a').attr('href');
  });
 });
});
```

global. Si pensamos en los navegadores web, este objeto es el parámetro `window`. Veamos un ejemplo de ello en el siguiente código: 2

```
console.log( this === window );  // Devuelve true
function prueba(){
  console.log( this === window);
}
prueba(); // Devuelve true
```

4. En funciones normales de modo estricto, el parámetro `this` siempre tiene el valor `undefined`. Vea si no este ejemplo: 3

```
function ejemplo() { 'uso estricto'; return this }
ejemplo() === undefined
// Devuelve true
```

5. Actualmente, el patrón de invocación más utilizado por los desarrolladores es el de invocación por método. ¿En qué consiste este patrón? En una sentencia, la función queda almacenada como propiedad de un objeto y se convierte en método. Al invocar a dicho método, el parámetro this se refiere al propio objeto, sin buscar en el contexto global. Veamos de forma práctica este uso del parámetro `this`: 4

```
var caso1 = {
  nombre : 'Ingrid',
  nacionalidad : 'danesa',
  usuario : function(){
    return this.nombre + ' ' + this.nacionalidad;
  }
}
console.log( caso1.usuario() );
// El resultado devuelto por el programa será Ingrid danesa
```

2

```
console.log( this === window );
// Devuelve true
function prueba(){
  console.log( this === win
}
prueba();
// Devuelve true
```

4

```
var caso1 = {
  nombre : 'Ingrid',
  nacionalidad : 'danesa',
  usuario : function(){
    return this.nombre + ' ' + this.nacionalidad;
  }
}
console.log( caso1.usuario() );
// El resultado devuelto por el programa será Ingrid danesa
```

3

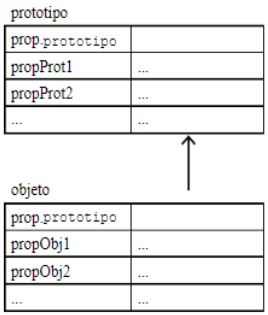
```
function ejemplo() { 'uso estricto'; return this }
ejemplo() === undefined
// Devuelve true
```

Sepa que en el caso del parámetro `this` en la invocación de métodos, el valor de dicho parámetro se denomina **receptor**.

Relación de prototipo entre objetos

EN JAVASCRIPT, PROTOTYPE ES UNA PROPIEDAD de funciones y de objetos generados por una función constructora. El prototipo de una función es un objeto y se utiliza, sobre todo, cuando dicha función es aplicada como constructor. Como veremos en este ejercicio, la relación de prototipo entre dos objetos está basada en la herencia.

1. En este ejercicio, trataremos el tema de los prototipos en JavaScript, concretamente del uso de estas propiedades entre dos o más objetos. Como hemos indicado en la introducción, la relación de prototipo entre dos objetos está basada en la herencia. ¿Qué significa esta afirmación? Sencillamente que un objeto puede utilizar a otro objeto como su prototipo y heredar así todas sus propiedades.
2. La cadena de objetos conectados mediante la propiedad `prototype` se denomina **cadena de prototipo**. Al acceder a las propiedades de un objeto, JavaScript rastrea la cadena de prototipo siempre hacia arriba, hasta dar con la propiedad solicitada. En este esquema puede ver representada la estructura de una cadena de prototipo:



063

3. Si se recorre toda la cadena de prototipo y no se encuentra la propiedad especificada, el resultado será el valor `undefined`. Para poder comprender gráficamente cómo funciona la herencia entre objetos basada en prototipos, a continuación incluimos un fragmento de código:

```
var ejemplo1 = {
 nacionalidad: function () {
  return 'nombre: '+this.nombre; }
};
var ejemplo2 = {
  [[Prototype]]: ejemplo1,
    nombre: 'ejemplo2'
};
//Tenga en cuenta que, en este ejemplo, [[Prototype]]
es un elemento inventado para representar la propiedad
prototype.
```

4. En este ejemplo, la variable `ejemplo2` hereda la propiedad `nacionalidad` de la variable `ejemplo1`, aunque también posee una propiedad propia, no heredada, denominada `nombre`. Debe saber que es posible acceder a la propiedad heredada como si fuera del objeto mismo, y esto es posible gracias a la cadena de prototipo, la cual se comporta como si se trata de un objeto sencillo.

5. Y este comportamiento también podemos detectarlo cuando llamamos a un método: el valor del parámetro `this` es siempre el objeto en el cual empieza la búsqueda del método y no el punto en que éste se encuentra, lo que permite al método acceder a todas las propiedades incluidas en la cadena de prototipo. Vea como ejemplo las siguientes líneas basadas en el ejemplo anterior:

```
ejemplo2.nacionalidad()
'nombre: ejemplo2'
```

IMPORTANTE

En una cadena de prototipo, una propiedad de un objeto sustituye a una propiedad con la misma clave en el objeto que se crea.

1

```
var ejemplo1 = {
 nacionalidad: function () {
  return 'nombre: '+this.nombre; }
};
var ejemplo2 = {
  [[Prototype]]: ejemplo1,
    nombre: 'ejemplo2'
};
```

Tenga en cuenta que, en este ejemplo, `[[Prototype]]` es un elemento inventado para representar la propiedad `prototype`.

2

```
ejemplo2.nacionalidad()
'nombre: ejemplo2'
```

En este caso vea como dentro de la propiedad `nacionalidad` el parámetro `this` es `ejemplo2`, lo que permite al método acceder a la propiedad `ejemplo2.nombre`.

Compartir datos entre objetos

COMO HEMOS PODIDO COMPROBAR EN EL ejercicio anterior, la función de prototipo en JavaScript se basa en la herencia de propiedades entre objetos. Del mismo modo, este lenguaje de programación también permite compartir datos entre objetos mediante el prototipado.

1. Es evidente que los prototipos son una óptima opción para compartir datos entre objetos, en el sentido que múltiples objetos pueden tener el mismo prototipo, el cual, a su vez, controla todas las propiedades compartidas de estos objetos. Vea el siguiente ejemplo en el cual las dos variables existentes comparten un mismo método:

```
var mujer = { nombre: 'Candela',
 metodo: function () {
  return 'Se llama '+this.nombre;
 }
};
var hombre = { nombre: 'Mariano',
 metodo: function () {
  return 'Se llama '+this.nombre;
 }
};
```

2. Efectivamente, las variables `mujer` y `hombre` comparten el método denominada en este ejemplo `metodo`. Pues bien, JavaS-

1

```
var mujer = { nombre: 'Candela',
 metodo: function () {
  return 'Se llama '+this.nombre;
 }
};

var hombre = { nombre: 'Mariano',
 metodo: function () {
return 'Se llama '+this.nombre;
 }
};

// El método compartido por estos dos objetos
es el denóminado "metodo"
```

cript permite generar un nuevo prototipo a partir de tipos de objeto similares (en este caso, `mujer` y `hombre`) e incluir en él el o los métodos también compartidos (en este caso, `metodo`). Veamos cómo quedaría el fragmento anterior una vez generado el prototipo compartido: 2

```
var protoComp = {
 metodo: function () {
  return 'Se llama '+this.nombre;
 }
};
var mujer = {
 [[Prototype]]: protoComp,
  nombre: 'Candela'
};
var hombre = {
 [[Prototype]]: protoComp,
 nombre: 'Mariano'
};
```

3. El funcionamiento del proceso de compartición de datos entre objetos se basa en que los datos se encuentran en el primer objeto de una cadena de prototipo y los métodos, en los objetos posteriores.
4. Así, afirmamos que en JavaScript la preparación de una propiedad afecta sólo al primer objeto en una cadena de prototipo, mientras que la asignación de dicha propiedad considera la cadena completa.
5. En el próximo ejercicio veremos cómo crear nuevos objetos preparando y asignando prototipos.

2

```
var protoComp = {
 metodo: function () {
  return 'Se llama '+this.nombre;
 }
};
var mujer = {
 [[Prototype]]: protoComp,
  nombre: 'Candela'
};
var hombre = {
 [[Prototype]]: protoComp,
 nombre: 'Mariano'
};
```

En este ejemplo, hemos denominado al prototipo común `protComp`. La interacción entre objetos, propiedades y métodos sería la siguiente:

```
mujer.metodo()
// resultado: "Se llama Candela"
hombre.metodo()
// resultado: "Se llama Mariano"
```

Crear nuevos objetos a partir de prototipos

IMPORTANTE

En otros lenguajes de programación, como Java o C++, la creación de nuevos objetos se realiza mediante el uso de clases o de interfaces. Sin embargo, como JavaScript no dispone de clases del modo que se entienden en estos otros lenguajes de programación, debe utilizar otros métodos para realizar la misma acción, y en este caso, este método es la herencia basada en prototipos.

SI NECESITAMOS CREAR NUEVOS OBJETOS EN JavaScript que reflejen las mismas propiedades que otros ya existentes, podemos recurrir a la copia de propiedades o, lo que es lo mismo, a la herencia basada en prototipos. En el ejercicio anterior hemos podido ver cómo compartir datos entre objetos utilizando este método. En este ejercicio, veremos cómo generar nuevos objetos mediante la herencia basada en prototipos.

1. Imagine que necesita crear lo que en otros lenguajes se denomina "clase" desde la cual poder instanciar objetos, con las mismas propiedades y atributos. Como en JavaScript las clases no existen, será necesario generar directamente un objeto del cual poder copiar sus propiedades. Para ello, JavaScript utiliza la palabra `new` para que cualquier función pueda ser instanciada como un nuevo objeto. Veamos el siguiente ejemplo:

```
function Ejemplo1() {
  alert("¡Hola Mundo!")
}
var caso1 = new Ejemplo1()
```

2. Las primeras tres líneas representan un script con una función convencional, que incluye un cuadro de alerta con el texto indicado. Sin embargo, la cuarta línea del código incluye una variable que generará un nuevo objeto basado en la función `ejemplo1()`, es decir, a partir de su "clase" prototipo.

3. Una vez se ha creado el nuevo objeto, la función utilizada como base pasa a ser el constructor de dicho objeto, lo que

1

```
function Ejemplo1() {
  alert("¡Hola Mundo!")
}
var caso1 = new Ejemplo1()
```

El nuevo ejemplo se crea con la partícula `new` a partir de su prototipo `ejemplo1()`.

2

```
function Ejemplo1(nombre) {
  alert("¡Hola Mundo!")
  this.nombre = nombre
  this.saluda = function() {
     alert("hola "+this.nombre)
  }
}
var caso1 = new Ejemplo1("niños")
var caso2 = new Ejemplo1("niñas")
caso1.saluda() // El resultado para este objeto será "hola niños"
caso2.saluda() // El resultado para este objeto será "hola niñas"
```

significa que será posible pasarle nuevos parámetros para asignarlos en cualquier ocasión al objeto.

4. Otro aspecto a tener en cuenta es que cuando deseemos acceder a cualquiera de los atributos asignados al objeto, será necesario utilizar el operador this.atributo. Veamos sobre un ejemplo la ampliación del código anterior que refleja el uso del nuevo objeto creado a partir de su prototipo:

```
function Ejemplo1(nombre) {
  alert("¡Hola Mundo!")
  this.nombre = nombre
  this.saluda = function() {
     alert("hola "+this.nombre)
  }
}
var caso1 = new Ejemplo1("niños")
var caso2 = new Ejemplo1("niñas")
caso1.saluda()
caso2.saluda()
```

5. Para evitar que las funciones que se comportan como prototipos se confundan con otras funciones normales o con variables, los desarrolladores recomiendan escribirlas con la inicial en mayúsculas (`Ejemplo1()`).

6. JavaScript también permite generar nuevos objetos basados en prototipos utilizando funciones existentes como expresiones. En el ejemplo siguiente puede ver el mismo fragmento que en pasos anteriores reconvertido utilizando este otro proceso:

```
var Ejemplo1 = function(nombre) {
    alert("¡Hola Mundo!")
    this.nombre = nombre
    this.saluda = function() {
        alert("hola "+this.nombre)
    }
}
```

3

```
var Ejemplo1 = function(nombre) {
    alert("¡Hola Mundo!")
    this.nombre = nombre
    this.saluda = function() {
        alert("hola "+this.nombre)
    }
}
```

Podemos crear nuevos objetos basados en prototipos utilizando funciones existentes o expresiones.

Repetir y detectar propiedades I

EN OCASIONES PUEDE RESULTAR CONVENIENTE CONOCER dónde se encuentra una propiedad en concreto dentro de un script en el cual se ha practicado la herencia de propiedades. La repetición y la detección de propiedades se encuentra influida por dos aspectos: la herencia propiamente dicha y la enumerabilidad de las propiedades.

1. En este ejercicio veremos cómo repetir y detectar propiedades. Como hemos indicado en la introducción de este ejercicio, la repetición y la detección de propiedades está influida por dos aspectos: por un lado, la herencia, es decir, se confrontan las propiedades propias del objeto con las heredadas, y por otro, la enumerabilidad, es decir, se confrontan las propiedades enumerables con las no enumerables. Veamos paso a paso qué significan estos dos aspectos.
2. En cuanto a la herencia de propiedades, las propiedades propias del objeto se almacenan directamente dentro de dicho objeto, mientras que las propiedades heredadas lo hacen dentro de su prototipo.
3. ¿Cómo podemos detectar las propiedades propias de un objeto? Mediante la función de JavaScript `Object.getOwnPropertyNames(objeto)`, en la cual el elemento `(objeto)` es obligatorio. 1

```
function Legumbre(medida, peso, forma) {
    this.medida = medida;
    this.peso = peso;
    this.forma = forma;

    // Definimos un método para la función anterior:
    this.toString = function () {
        return (this.medida + ", " + this.peso + ", " + this.forma);
    }
}

// Creamos un nuevo objeto con propiedades here    s:
var lenteja = new Legumbre("mini", 0.2, "plana");

// Obtenemos las propiedades enumerables y    s métodos
del objeto en una matriz:
var legumbres = Object.getOwnPropertyNames(lenteja);
document.write (legumbres);
```

Object.getOwnPropertyNames(lenteja)

El tipo de resultado que devuelve esta función es una matriz que contiene los nombres de las propiedades propias del objeto.

4. Hablemos ahora de la enumerabilidad de las propiedades. La enumerabilidad de una propiedad es,en realidad, un atributo, una marca que puede ser `true`o `false`. El caso es que la enumerabilidad en raras ocasiones es importante y puede normalmente ser ignorada.
5. Por otro lado, también podemos definir las propiedades enumerables en JavaScript como aquellas que se pueden ejecutar repetidamente en un ciclo, como el que marcan los parámetros `for...in`
6. En cualquier caso, el propietario de estas propiedades se determina según si la propiedad pertenece al objeto directamente y no a la cadena de prototipos. (Si necesita recordar algún aspecto relacionado con la cadena de prototipos en JavaScript, no dude en recuperar el ejercicio 63 de este libro.)
7. Si deseamos obtener una lista de las propiedades enumerables de un script sepa que tiene a su disposición la función `Object.keys(objeto)`, cuyo parámetro `(objeto)` también es obligatorio. Este objeto, que contiene las propiedades y los métodos a analizar, puede ser un objeto creado o bien uno existente en el Modelo de Objeto de Documento (el DOM).
8. El tipo de resultado que devuelve la función `Object.keys(objeto)` es una matriz con los nombres de las propiedades y los métodos enumerables del objeto. Tenga en cuenta que si el valor proporcionado para el parámetro (objeto) no corresponde precisamente a un objeto en Javascript, se producirá un error del tipo `TypeError`.

2

```
function Legumbre(medida, peso, forma) {
    this.medida = medida;
    this.peso = peso;
    this.forma = forma;

    // Definimos un método para la función anterior:
    this.toString = function () {
        return (this.medida + ", " + this.peso + ", " + this.forma);
    }
}                    Object.keys(lenteja);

// Creamos un nuevo objeto con pr  edades heredadas:
var lenteja = new Legumbre("min    0.2, "plana");

// Obtenemos las propiedades  numerables y los métodos
del objeto en una matriz:
var legumbres = Object.keys(lenteja);
document.write (legumbres);
```

Repetir y detectar propiedades II

SI EN EL EJERCICIO ANTERIOR HEMOS podido realizar un primer acercamiento a las funciones de JavaScript que nos permiten detectar y listar tanto las propiedades propias de un objeto como las enumerables, en estas páginas trataremos de profundizar en este aspecto insistiendo en los métodos que este lenguaje de programación pone a nuestra disposición para gestionar las propiedades heredadas o no de los objetos.

1. Si lo que usted desea es hacer una lista con todas las propiedades de un objeto, tanto las propias como las heredadas, debe saber que tiene dos opciones: utilizar el bucle `for...in` o implementar una función personal que repita todas las funciones, tanto propias como enumerables). Veamos cada una de estas opciones en qué consiste.
2. El bucle `for...in` sólo repite aquellas propiedades que son enumerables de los objetos mismos y aquellas que el objeto hereda de su prototipo. La sintaxis de este bucle es la siguiente: 1

```
for (variable in objeto) {...
}
```

3. En cada repetición, se asigna al valor `variable` un nombre de propiedad distinto. Por su parte, el valor `objeto` de la sentencia contiene el objeto cuyas propiedades enumerables han sido repetidas. El resultado obtenido por cada propiedad distinta será la ejecución de una sentencia.

1

```
var obj = {a:1, b:2, c:3};

for (var prop in obj) {
  console.log("o." + prop + " = " + obj[prop]);
}
```

El resultado obtenido tras la ejecución de la sentencia for...in es: "o.a = 1", "o.b = 2" y "o.c = 3".

2

```
function getAllPropertyNames(objeto) {
 var resultado = [];
 while (objeto) {
   Array.prototype.push.apply(resultado, Object.getOwnPropertyNames(objeto));
   objeto = Object.getPrototypeOf(objeto);
 }
return resultado;
}
```

Después de la sentencia `while()`, se añaden los nombres de las propiedades propias del elemento `'objeto'` al elemento `'resultado'`.

4. Para entender la segunda de las opciones indicadas en el paso 1 de este ejercicio, será mejor contemplar el siguiente ejemplo: 2

```
function getAllPropertyNames(objeto) {
 var resultado = [];
 while (objeto) {
   Array.prototype.push.apply(resultado,
   Object.getOwnPropertyNames(objeto));
   objeto = Object.getPrototypeOf(objeto);
 }
return resultado;
}
```

5. Además de realizar las acciones que acabamos de ver con las propiedades de un objeto, JavaScript propone métodos específicos para comprobar tanto si un objeto tiene una determinada propiedad como si una propiedad determinada existe dentro de un objeto.

6. El operador `in` devuelve como resultado `true`, tanto ante propiedades propias como enumerables. La sintaxis de este operador es `propiedad in objeto`, siendo los términos `propiedad` y `objeto` los elementos a comprobar. El resultado será verdadero si el elemento `objeto` tiene la propiedad indicada. 9

7. El método `hasOwnProperty()` se utiliza para averiguar si un objeto tiene la propiedad especificada. A diferencia del operador `in`, este método podrá devolver como resultado `true` o `false`, según si la propiedad es enumerable o no enumerable (`true`) o si es enumerable heredada o no enumerable heredada (`false`). 5 Tenga en cuenta que todo objeto descendiente de otro objeto hereda el método `hasOwnProperty()`.

3

```
var animal = new String("gato");
"length" in animal
// devuelve true en cuanto la propiedad "length"
pertenece a String
```

4

```
objeto1 = new Object();
objeto1.propiedad = 'existe';

function cambio() {
  objeto1.newprop = objeto1.propiedad;
  delete objeto1.propiedad;
}

objeto.hasOwnProperty('propiedad');   // Devuelve true
cambio();
objeto.hasOwnProperty('propiedad');   // Devuelve false
```

Resulta importante saber que los expertos en JavaScript recomiendan evitar la llamada del método `hasOwnProperty()` directamente sobre un objeto debido al riesgo de que pueda ser sobrescrito.

Proteger objetos

IMPORTANTE

El método `Object.preventExtensions(objeto)` evita que se añadan nuevas propiedades al objeto especificado, pero no al objeto prototipo.

ESTÁ MÁS QUE COMPROBADO QUE ACTUALMENTE no existe ninguna garantía que proteja los métodos al ser aplicados en matrices u objetos. Por esta razón, JavaScript cuenta con ciertos métodos que permiten proteger el valor de las propiedades de estos elementos, métodos que describimos en este ejercicio.

1. Son tres los métodos considerados como más efectivos para proteger el valor de las propiedades de los objetos ante posibles alteraciones involuntarias. Dichos métodos son los siguientes:

```
Object.preventExtensions(objeto)
Object.seal(objeto)
Object.freeze(objeto)
```

2. El primero (prevención) sirve para evitar que se añadan nuevas propiedades a un objeto `(objeto)`; el segundo (sellado) se utiliza para lo mismo que el anterior y además, para evitar la manipulación de las propiedades existentes, y el tercero, y más eficiente, (congelación) protege ante cualquier escritura todas las propiedades y, además, sella directamente o evita la edición del elemento `(objeto)`. Veamos estos métodos uno a uno.

3. El método `Object.preventExtensions(objeto)` hace que un objeto deje de admitir extensiones, lo que significa que no podrá adquirir nuevas propiedades, a parte de las que ya dispone. Es importante tener en cuenta que este método sólo evita que se añadan propiedades en el objeto, pero no que las existentes sean eliminadas. Por esta razón, éste es el método de protección de objetos menos consistente. 1

1

```
var objeto1 = {};
var objeto2 = Object.preventExtensions(objeto1);
assert(objeto1 === objeto2);
```

En este ejemplo, el método `Object.preventExtensions()` devuelve como resultado el mismo objeto `objeto1` convertido en no ampliable.

4. Resulta interesante saber que Javascript permite convertir en no ampliable un objeto, pero no permite llevar a cabo la acción contraria, es decir, convertirlo en ampliable.
5. El método `Object.seal(objeto)`, como su nombre indica, sella un objeto evitando que se añadan nuevas propiedades y marcando las existentes como no configurables, lo que significa que el conjunto de propiedades del objeto se convierten en fijas y no manipulables. 2 Este método también protege a las propiedades de ser convertidas de propiedades de datos a propiedades de descritor de acceso, y viceversa.
6. Si se intenta añadir propiedades a un objeto sellado o bien eliminar las existentes, o bien si se intenta realizar una conversión entre tipos de propiedades (como hemos indicado en el paso anterior), sepa que se producirá un error en la ejecución del script, pudiendo ser el no funcionamiento del mismo como la aparición de un error del tipo `TypeError`.
7. El tercer y más eficiente método de protección de objetos es `Object.freeze()`. Este método evita que se añadan propiedades al objeto, que las propiedades existentes sean eliminadas y también que las condiciones de enumerabilidad, configuración y escritura de estas propiedades no sean alteradas. En resumen, el objeto, gracias a este método, se convierte en inmutable. 3
8. En el caso de las propiedades de datos, los valores de un objeto congelado no podrán ser modificados, mientras que las propiedades de descripción de acceso continúan funcionando con normalidad.

IMPORTANTE

Las variables pueden declararse con o sin la sentencia `var`. Sin embargo, es importante saber que el resultado de uno u otro caso es distinto. Si utiliza `var`, la variable declarada será local, mientras que si no lo utiliza, la variable será global.

2

```
var objeto1 = {
  prop: function() {},
  foo: 'bar'
};
var objeto1 = Object.seal(objeto1);

assert(objeto2 === objeto1);
assert(Object.isSealed(objeto1) === true);
```

3

`Object.freeze(objeto1)`

```
var objeto1 = {
  prop: function() {},
  foo: 'bar'
};
var objeto1 = Object.freeze(objeto1);

assert(objeto2 === objeto1);
assert(Object.isFrozen(objeto1) === true);
```

Cualquier intento de modificar o alterar las propiedades de un objeto congelado producirá un error en la ejecución del script.

Constructores de objetos

EN JAVASCRIPT, LOS CONSTRUCTORES SON LO que en otros lenguajes de programación son las clases. Se trata de funciones que ayudan a crear objetos similares entre ellos. Los constructores no dejan de ser funciones convencionales, aunque nombradas, configuradas e invocadas de forma algo distinta.

1. En este ejercicio veremos cómo funcionan los constructores en JavaScript. Empecemos. Un constructor es una función que se invoca mediante el operador `new`. Por norma general, el nombre de las funciones constructoras se escribe con la inicial en mayúsculas para distinguirlas de las funciones convencionales, las cuales, como ya sabe, las escribimos con la inicial en minúscula.
2. ¿Cómo funciona el operador `new`? `new` sigue básicamente dos pasos: primero crea el objeto y después configura los datos de dicho objeto. Los objetos creados por el constructor se denominan **instancias**.
3. Podemos convertir una función normal en un constructor de objetos utilizando para ello el operador `new`. Veamos un ejemplo:

```
function coche(fabricante, modelo, año) {
   this.fabricante = fabricante;
   this.modelo = modelo;
   this.año = año;
}
```

new Legumbre

1

```
// Creamos un nuevo objeto con propiedades heredadas:
var lenteja = new Legumbre("mini", 0.2, "plana");

// Obtenemos las propiedades enumerables y los métodos
del objeto en una matriz:
var legumbres = Object.getOwnPropertyNames(lenteja);
document.write (legumbres);
```

function cuest()

2

```
function cuest() {
 for (var i=0; i<preguntas.length; i++) {
  askQuestion(aleat);
  aleatorio = Math.floor(Math.random() *preguntas.length);
  aleat = preguntas[aleatorio];
 }
```

4. Este fragmento corresponde a una función normal con un nombre, unas propiedades y un método. Recuerde que utilizamos `this` para asignar valores a las propiedades del objeto basadas en los valores pasados a la función. Ahora, gracias al operador `new`, crearemos dos nuevos objetos basados en la función `coche`: 3

```
objeto1 = new coche("Peugeot", "3008", 2015);
objeto2 = new coche("Seat", "Ibiza", 2012);
```

5. `objeto1` y `objeto2` son instancias de `coche`. Y podríamos generar tantas como quisiéramos o necesitáramos. Otro aspecto interesante respecto al uso de constructores para crear nuevos objetos es que un objeto puede poseer una propiedad que, en sí misma, es un objeto. Por ejemplo:

```
function propietario(nombre, edad) {
   this.nombre = nombre;
   this.edad = edad;
}
persona1 = new propietario("Candela Sol", 29 );
persona2 = new propietario("Ana Polo", 30);
```

6. Ahora, retomaríamos la definición del objeto `coche` para incluir una nueva propiedad basada en el objeto `propietario`, denominada por ejemplo, cliente, la cual se reflejaría en las instancias de los objetos. 4

7. Una vez hecho esto, podríamos utilizar una nueva propiedad para que en lugar del nombre del objeto que aparece en la sentencia se muestre el correspondiente valor o, en este caso, nombre del cliente: 5

```
objeto1.cliente.nombre
```

5

```
objeto1 = new coche("Peugeot", "3008", 2015, "Candela Sol");
objeto2 = new coche("Seat", "Ibiza", 2012, persona2);
```

3

```
function coche(fabricante, modelo, año) {
   this.fabricante = fabricante;
   this.modelo = modelo;
   this.año = año;
}

objeto1 = new coche("Peugeot", "3008", 2015);
objeto2 = new coche("Seat", "Ibiza", 2012);
```

4

```
coche(fabricante, modelo, año, cliente) {
abricante = fabricante;
odelo = modelo;
this.año = año;
this.cliente
}

objeto1 = new coche("Peugeot", "3008", 2015, persona1);
objeto2 = new coche("Seat", "Ibiza", 2012, persona2);
```

Compruebe que en lugar de pasar una cadena literal o un valor entero en el momento de la creación de los nuevos objetos, se pasan los objetos `persona1` y `persona2` como argumentos para la propiedad `cliente`.

Herencias entre constructores

EN JAVASCRIPT ES POSIBLE GENERAR CONSTRUCTORES derivados de un constructor principal, que hereden todas sus propiedades y características y que, además, puedan disponer de otras propias. A estos constructores heredados podríamos denominarlos subconstructores, y aunque no existe en este lenguaje de programación ningún mecanismo automático que permita generarlos rápidamente, sí que existen ciertos procesos más manuales, que describiremos a continuación.

1. En este ejercicio veremos cómo funciona en JavaScript la herencia entre constructores. La idea de este tipo de herencia consiste en que el **subconstructor** (el que hereda) tendrá todas las propiedades del **constructor principal** (el que proporciona la herencia), tanto las de prototipo como las de las instancias; además, el subconstructor dispondrá también de sus propias propiedades. Pero ¿cómo llegamos a conseguir este subconstructor? Vayamos por partes.
2. Es necesario invocar o llamar al constructor principal para poder heredar sus propiedades de instancias, puesto que dichas propiedades se encuentran dentro de dicho constructor.[1] Luego se invocará el subconstructor mediante el operador `new` para que el parámetro implícito `this` se refiera a una instancia nueva.

[1]

```
function Subconstructor(prop1, prop2, prop3, prop4) {
 Constructor.call(this, prop1, prop2); // (1)
 this.prop3 = prop3; // (2)
 this.prop4 = prop4; // (3)
}
```

3. Para que el resultado sea el esperado, es imprescindible que el constructor principal no sea llamado mediante el operador `new`, puesto que lo que conseguiríamos es crear una instancia precisamente del constructor principal. El proceso correcto es llamar al constructor principal como una función y representar la subinstancia actual como el valor del parámetro `this`.
4. Llegados a este punto, las propiedades compartidas entre ambos constructores, como los métodos, se almacenan en la instancia prototipo. ¿Cómo podemos hacer que las propiedades del prototipo del constructor principal pasen al subconstructor? Mediante la función `Object.create()`, la cual genera un nuevo objeto cuyo prototipo es el del constructor principal. 2
5. Tras realizar todos estos pasos, ya habremos obtenido nuestro subconstructor, el cual contendrá, además de sus propias propiedades, aquellas contenidas en el constructor principal. 3 Sin embargo, debemos asegurarnos de que todas y cada una de las instancias del subconstructor deben ser también instancias del constructor principal. Y para ello, podemos utilizar una vez más el operador `instanceof`:

```
subInstancia instanceof Subconstructor
Subconstructor.prototype.isPrototypeOf(subInstancia)
```

6. Como `Subconstructor.prototype` es uno de los prototipos de `subInstancia`, las dos afirmaciones anteriores son verdaderas. Del mismo modo, también son verdaderas las siguientes afirmaciones, dado que `subInstancia` es también un instancia del constructor principal (`Constructor`):

```
subInstancia instanceof Constructor
Constructor.prototype.isPrototypeOf(subInstancia)
```

2

```
Subconstructor.prototype = Object.create(Constructor.prototype);
    Subconstructor.prototype.constructor = Subconstructor;
    Subconstructor.prototype.metodoB = ...;
    Subconstructor.prototype.metodoC = ...;
```

3

Constructor.prototype

metodoA	
metodoB	...

Subconstructor.prototype

[[Prototype]]	
metodoB	...
metodoC	...

subInstancia

[[Prototype]]	
prop1	...
prop2	...
prop3	...
prop4	...

Métodos comunes a todos los objetos

IMPORTANTE

`Object.prototype.toLocaleString()` es otro método que devuelve una cadena de caracteres que representa al objeto. La implementación por defecto llama al método `toString()`. `toLocaleString` quedará invalidado por aquellos objetos descendientes con fines específicos locales.

TODOS LOS OBJETOS TIENEN LA PROPIEDAD Object.prototype en su cadena de prototipos. Del mismo modo, dicha propiedad dispone de una serie de métodos que proporciona a sus prototipos. En este ejercicio veremos cuáles son estos métodos y cómo se utilizan.

1. Empezaremos por un par de métodos que se utilizan para convertir un objeto en un valor primitivo: los métodos `Object.prototype.toString()` y `Object.prototype.valueOf()`. El primero de ellos devuelve una cadena de caracteres que representa a un objeto, mientras que el segundo convierte un objeto en un número.
2. Por defecto, todos los objetos que descienden de un elemento Object heredan el método `toString`. Si dicho método no se sobrescribe en el nuevo objeto, devuelve la cadena `[objeto type]`, siendo el valor `type` el tipo de objeto creado.
3. El método de conversión `Object.prototype.valueOf()` devuelve el valor primitivo del objeto indicado. Es interesante saber que en la mayoría de los casos no será usted quien invoque este método, sino que JavaScript lo hará automáticamente al detectar un objeto en el lugar en que se esperaba un valor primitivo. Aun así, si usted lo necesita, puede hacerlo de la manera siguiente:

```
numero.valueOf()
```

1

```
var objeto = new Object();
objeto.toString();
// Devuelve [object Object]
```

2

```
[object Object]
```

```
numero.prototype.valueOf = function() {
 return valorPrimitivoPersonalizado;
};
```

071

4. Igual que el método `toString`, `valueOf()` es heredado por todos los objetos descendientes del elemento `Object`, lo que significa que todos los objetos incluidos en el núcleo del lenguaje devolverán un valor apropiado después de sobrescribir el método. En el caso en que un objeto no cuente con ningún valor primitivo, el método `valueOf()` devuelve el objeto en sí. [2]

5. Existen una serie de métodos que se ocupan de las propiedades y la herencia de prototipos. Se trata de `Object.prototype.isPrototypeOf()`, `Object.prototype.hasOwnProperty()` y `Object.prototype.propertyIsEnumerable()`. Veamos brevemente en qué consiste cada uno de estos métodos.

6. `Object.prototype.isPrototypeOf()` comprueba si existe un objeto determinado dentro de una cadena de prototipo de otro objeto. Si el objeto examinado forma parte de dicha cadena, el método devuelve `true`. [3] Si no, devuelve `false`. [4]

7. El método `Object.prototype.hasOwnProperty()` devuelve un valor booleano e indica si el objeto cuenta con la propiedad especificada. Todo objeto que desciende del elemento `Object` hereda este método, el cual se utiliza sobre todo para determinar si la propiedad de un objeto es una propiedad directa de dicho objeto. Si la propiedad existe en el objeto, y no en su prototipo, el método devuelve `true`. [5]

8. Por último, el método `Object.prototype.propertyIsEnumerable()` devuelve un valor booleano que indica si una propiedad especificada es enumerable o no. El valor devuelto será `true` si la propiedad es enumerable; en caso contrario, el resultado será `false`. Todos los objetos disponen de este método.

IMPORTANTE

La diferencia entre el método `isPrototypeOf` y el operador `instanceof` es que dicho operador realiza las comprobaciones de los objetos sobre el prototipo de una función, no sobre la función misma.

[3]
```
var prototipo = { };
var objeto = Object.create(prototipo);
prototipo.isPrototypeOf(objeto)
// El método devuelve true
```

[4]
```
var prototipo = { };
var objeto = Object.create(prototipo);
objeto.isPrototypeOf(objeto)
// El método devuelve false
```

[5]
```
var prototipo = { propiedad: 'ejemplo1' };
var objeto = Object.create(prototipo);
objeto.prop = 'ejemplo2';

Object.prototype.hasOwnProperty.call(objeto, 'prop')
// El método devuelve true
```

[6]
```
var objeto = { propiedad: 'ejemplo1' };
objeto.propertyIsEnumerable('propiedad')
// En este caso, el método devuelve true

var objeto = { propiedad: 'ejemplo1' };
objeto.propertyIsEnumerable('toString')
// En este caso, el método devuelve false
```

Trabajar con matrices

EN JAVASCRIPT, UNA MATRIZ ES UN mapa desde índices, es decir, valores numéricos que tienen su inicio en el valor 0, hasta valores arbitrarios. Estos valores se denominan los elementos de una matriz. El método más recomendable para crear una matriz es mediante la enumeración directa de sus elementos, cuya posición especifica implícitamente su índice.

1. Aunque ya hemos tenido ocasión de trabajar con ellas, en este ejercicio y en los siguientes trataremos con mayor detalle el uso y el funcionamiento de las matrices en JavaScript. Este primer ejercicio servirá como introducción a los siguientes, en los cuales detallaremos cada uno de los aspectos destacados en estas páginas.
2. El primer aspecto que hay que conocer en torno a las matrices es su creación y el acceso a los elementos que la componen. Dicha creación se realiza habitualmente escribiendo literalmente sus elementos entre claudátors y se accede a ellos utilizando los valores de índice correspondientes. 1
3. Los elementos de una matriz pueden incrementarse o eliminarse fácilmente gracias a la propiedad de las matrices `length`. 2 Cuando se trata de incrementar el número de elementos, recuerde que también puede utilizar el método `push()`. 3

1

```
var arr = [ 'fruta', 'verdura', 'carne' ];
arr[1]
// Como el índice 0 corresponde
al primer elemento de una matriz,
el resultado será 'verdura'
```

2

```
var arr = [ 'fruta', 'verdura', 'carne' ];
arr[arr.length] = 'pescado';
// En este caso, la propiedad length
añade un nuevo elemento a la matriz
```

3

```
var arr = [ 'fruta', 'verdura', 'carne' ];
arr.push('pescado')
// En este caso, es el método push()
el que añade un nuevo elemento a la matriz
```

```
var arr = [ 'fruta', 'verdura', 'carne' ];
arr.length = 2;
// Como l matriz tiene tres elementos,
el valor 2 asignado a la propiedad length
elimina el último de estos elementos,
por lo que el resultado de la matriz será
[ 'fruta', 'verdura' ]
```

4. En JavaScript, no es necesario que los elementos de las matrices sigan un orden lógico, así como pueden existir en ellas espacios vacíos. Esto significa que algún índice puede no contener ningún elemento: 4

```
var matriz = [ ];
matriz[0] = 'fruta';
matriz[2] = 'carne';
```

5. Los elementos contenidos en la matriz `matriz` serán los siguientes:

```
[ 'fruta', , 'carne' ]
```

6. El valor 1, que corresponde al segundo elemento de la matriz, se encuentra vacío. Sin embargo, debe saber que la mayoría de los intérpretes JavaScript optimizan las matrices sin huecos y almacenan los elementos de forma continua y lógica.

7. Como objetos que son, las matrices también pueden tener propiedades. En este aspecto, cabe destacar que dichas propiedades no se consideran elementos de la matriz; de hecho, no se consideran ni parte de ellas, por lo que en el código quedarán reflejadas independientemente: 5

```
var matriz = [ 'fruta', 'verdura' ];
matriz.propiedad = "huerto";
// La matriz continúa siendo [ 'fruta', 'verdura' ]
// La propiedad matriz.propiedad
tiene el valor "huerto"
```

8. En los próximos ejercicios iremos detallando cada uno de estos puntos para que el trabajo con matrices en JavaScript no represente ningún problema para usted.

4

```
var matriz = [ ];
matriz[0] = 'fruta';
matriz[2] = 'carne';
// El resultado de la matriz será
[ 'fruta', , 'carne' ], con el segundo
elemento vacío
```

5

```
var matriz = [ 'fruta', 'verdura' ];
matriz.propiedad = "huerto";
// La matriz continúa siendo [ 'fruta', 'verdura' ]
// La propiedad matriz.propiedad
tiene el valor "huerto"
```

Crear matrices

AUNQUE LA FORMA MÁS SENCILLA Y, a su vez, más utilizada de crear matrices en JavaScript es escribiendo sus elementos literalmente, debe saber que también puede realizar esta tarea utilizando el constructor Array, el cual permite dos usos distintos: la creación de una matriz vacía con una longitud determinada y la inicialización de una matriz con elementos. En este ejercicio veremos todas estas formas de crear matrices en JavaScript.

1. Cuando crea una matriz escribiendo los elementos que la componen literalmente en el script, debe saber que las comillas en las matrices son ignoradas. [1] Vea este ejemplo:

```
var frutas = [ 'manzana', 'pera', 'uva' ];
[ 'manzana', 'pera' ].length
// El resultado es 2 (elementos)
 ['manzana', 'pera', ].length
// El resultado es 2 (elementos)
[ 'manzana', 'pera', ,].length
// El resultado son 3 elementos, puesto que existe un
espacio vacío entre las dos últimas comas
```

2. Además de generar matrices a partir de la escritura literal de sus elementos, JavaScript también permite crearlas con el constructor `Array`. A diferencia de otros constructores, `Array` no necesita el operador `new`, es opcional. Esto significa que el comportamiento será el mismo si se invoca como función (sin `new`) que como constructor.

3. El constructor `Array` puede generar matrices de dos formas distintas: creando una matriz vacía con una longitud determinada e inicializando una matriz con elementos. En el primer

[1]
```
var frutas = [ 'manzana', 'pera', 'uva' ];
[ 'manzana', 'pera' ].length
// El resultado es 2 (elementos)
 ['manzana', 'pera', ].length
// El resultado es 2 (elementos)
[ 'manzana', 'pera', ,].length
// El resultado son 3 elementos, puesto que
existe un espacio vacío entre las dos últimas comas
```

[2]
```
frutas = new Array(10);
frutas[0] = "manzana";
frutas[1] = "pera";
frutas[2] = "uva";
```

caso, el constructor Array especifica un único parámetro numérico, así como la longitud inicial de la matriz: 2

```
frutas = new Array(10);
frutas[0] = "manzana";
frutas[1] = "pera";
frutas[2] = "uva";
```

4. En este ejemplo, la variable `frutas` contiene una nueva matriz generada mediante el constructor `Array`, la cual tiene una longitud de 10 elementos, de los cuales los tres primeros se especifican en las siguientes líneas del script.

5. El segundo procedimiento del constructor `Array`es la inicialización de una matriz con elementos, método que resulta muy similar a la creación de matrices literales. Sin embargo, los expertos en JavaScript no recomiendan este procedimiento puesto que no permite la creación de matrices con un único número como elemento. La razón es que la asignación de dicho valor numérico puede confundirse con la propiedad `length` de la matriz: 3

```
new Array(6.2)
    RangeError: longitud de matriz no válida
// El intérprete de JavaScript reconocerá el valor 6.2
como uno de los índices de la matriz
```

6. A diferencia de otros lenguajes de programación, JavaScript no admite directamente matrices multidimensionales. Por esta razón, este lenguaje utiliza la anidación entre matrices para conseguir el mismo resultado. Las matrices multidimensionales almacenan los datos en más de una dimensión y, por tanto, necesitan dos índices para acceder a cada elemento de la matriz. 4

3

```
new Array(6.2)
    RangeError: longitud de matriz no válida
// El intérprete de JavaScript reconocerá
el valor 6.2 como uno de los índices de la matriz
```

4

```
var temperatura_barcelona = new Array(2)
temperatura_barcelona[0] = 8
temperatura_barcelona[1] = 6

var temperatura_Bilbao = new Array (2)
temperatura_Bilbao[0] = 3
temperatura_Bilbao[1] = 2

var temperatura_Madrid = new Array (2)
temperatura_Madrid[0] = 6
temperatura_Madrid[1] = 5

// La matriz multidimensional a partir de estas
tres matrices sería la siguiente:

var temperaturas = new Array (3)
temperaturas[0] = temperatura_barcelona
temperaturas[1] = temperatura_Bilbao
temperatura[2] = temperatura_Madrid
```

La propiedad length en una matriz

BÁSICAMENTE, LA FUNCIÓN DE LA PROPIEDAD `length` en matrices se ocupa de seguir el valor de índice más elevado. Esto no significa que `length` se dedique a contar los elementos de una matriz, puesto que para ello JavaScript dispone de funciones específicas.

1. En este ejercicio trataremos todos los detalles de la propiedad `length` en las matrices. Antes de empezar debe tener muy claro que las matrices son elementos dispersos, lo que significa que no necesitan que sus elementos sigan un elemento lógico y ordenado sino que pueden contener elementos "vacíos". Con ello queremos decir que la propiedad `length` no siempre será el número de elementos de una matriz, sino que se ocupa de seguir el valor del índice más elevado.
2. Sepa que, si lo que desea es conocer el número real de elementos contenidos en una raíz, usted puede construir una función personalizada para ello e incluir el método `forEach`: 1

```
function contElem(matriz) {
 var recuento = 0;
matriz.forEach(function () {
   recuento++;
 });
return recuento; }
```

3. En este ejemplo, gracias al método `forEach` podemos contar todos los elementos pasando por alto los elementos vacíos o huecos (trataremos estos elementos vacíos en matrices en el ejercicio siguiente). 2

1

```
function contElem(matriz) {
 var recuento = 0;
matriz.forEach(function () {
   recuento++;
 });
return recuento; }
```

El método `forEach` realiza la acción especificada para cada elemento de una matriz.

2

```
contElem([ 'manzana', 'pera' ])
// El resultado será 2
contElem([ 'manzana', , 'pera' ])
// El resultado será 2
```

074

4. La longitud de una matriz, definida en la propiedad length de la misma, puede incrementarse manualmente cuando sea necesario. Sin embargo, tenga en cuenta que esta acción no añadirá literalmente elementos a la matriz sino sencillamente nuevos espacios: 3

```
var frutas = [ 'manzana', 'pera', 'uva' ];
frutas.length = 4;
// El resultado será [ 'manzana', 'pera', 'uva', ,]
```

5. La propiedad `length` nos indica 4 elementos y lo que ha hecho es añadir un espacio vacío al final de la cadena de elementos. Aunque por defecto los nuevo elementos vacíos se agregan al final de la matriz, la propiedad `length` puede utilizarse como indicador para ubicar nuevos elementos en un punto determinado: 4

```
var frutas = [ 'manzana', 'pera', 'uva' ];
frutas.push('naranja')
frutas.length = 5;
// El resultado será [ 'manzana', 'pera', 'uva', , 'naranja']
```

6. Si lo que deseamos es generar una matriz completamente vacía, podemos utilizar el constructor Array indicando la longitud inicial de la matriz:

```
var frutas = new Array(3);
frutas.length = 3
contElem(frutas) // Dará como resultado 0
```

7. Para terminar, imagine que necesita eliminar todos los elementos de una matriz puesto que tiene previsto reutilizarla con otro contenido. En este caso, sencillamente deberá fijar la propiedad `length` a 0: 5

```
function limpiarMatriz(frutas) {
 frutas.length = 0;
}
```

IMPORTANTE

Debe tener en cuenta que el proceso para eliminar todos los elementos de una matriz puede resultar bastante lento, según el número de elementos contenidos en dicha matriz. Paradójicamente, el proceso contrario, es decir, la creación de matrices vacías, es bastante rápido.

3

```
var frutas = [ 'manzana', 'pera', 'uva' ];
frutas.length = 4;
// El resultado será [ 'manzana', 'pera', 'uva', ,]
// No se añaden nuevo elementos literales, sólo espacios
```

Si no se indica lo contrario, la propiedad `length` añade los nuevos elementos vacíos al final de la matriz.

4

```
var frutas = [ 'manzana', 'pera', 'uva' ];
frutas.push('naranja')
frutas.length = 5;
// El resultado será [ 'manzana', 'pera', 'uva', , 'naranja']
// El nuevo elemento agregado mediante el método push()
se coloca en el punto indicado por la propiedad length
```

Gracias a la interacción entre `push()` y `length`, podemos agregar nuevos elementos literales a una matriz en el punto que nos interesa.

5

```
var frutas = [ 'manzana', 'pera', 'uva' ];
limpiarMatriz(frutas)
// El resultado será frutas = [ ]
```

Huecos en matrices

HABLAMOS DE HUECOS EN MATRICES PARA referirnos a aquellos índices que son más pequeños que la longitud que han desaparecido de la matriz. El hecho de que se puedan generar estos huecos se debe a que las matrices son mapas que van desde los mencionados índices hasta los valores. JavaScript otorga a estos tipos de índices el valor `undefined`.

1. En este ejercicio aprenderá a crear huecos en matrices, así como otros aspectos relacionados con estos elementos vacíos en matrices. Sin embargo, antes de empezar, es preciso advertir que los expertos aconsejan fervorosamente evitar el uso de huecos en las matrices. La razón es que, si bien JavaScript dispone de métodos para gestionarlos, no siempre lo hace de forma consistente; los elementos vacíos en las matrices pueden afectar el rendimiento de los scripts negativamente.
2. Dicho esto, ahora sí, vamos a empezar mostrándole cómo generar huecos en una matriz. JavaScript dispone de diferentes métodos para ello: asignando nuevos índices sin elementos a la matriz y omitiendo valores en la lista de elementos.
3. La generación de un elemento vacío mediante la asignación de un índice sin elementos se vio en el ejercicio anterior, aplicado a la propiedad `length`. Un ejemplo sencillo de este tipo de creación de huecos sería el fragmento de código que puede ver en la imagen 2.

1

```
<script type="text/javascript">
    var countries = ['Italia',
                     'Francia',
                     'Dinamarca',
                     'Reino Unido',
                     'Alemania',
                     'Rusia',
                     'Grecia',
                     'China'
                    ];
```

Los expertos recomiendan realizar matrices con todos sus elementos y sus índices para evitar así posibles errores en la interpretación de los scripts.

2

```
var frutas = [ ];
frutas[0] = 'manzana';
frutas[2] = 'uva';
//El índice 1 en esta matriz es un hueco
```

En este caso, la matriz `frutas` dispone de dos elementos en los índices 0 y 2; el índice 1 está vacío y, por consecuente, representa un hueco en la matriz.

075

4. La segunda forma de generar un hueco en una matriz es omitiendo algún valor en el momento de la escritura literal de la matriz. En este caso, deberá aparecer un espacio vacío delimitado por comas, aunque no necesariamente por comillas, puesto que como ya indicamos en un ejercicio anterior, estos signos son ignorados por el intérprete de Javascript.

5. Las matrices que contienen huecos se denominan en programación matrices dispersas, en comparación con las que no los contienen, que se denominan densas. Veamos algunos aspectos que nos pueden ayudar a identificar una matriz dispersa y a compararla con una densa. El siguiente ejemplo muestra una matriz dispersa con dos huecos y un valor:

```
var frutas = [ , , 'uva'];
```

6. Y en la siguiente matriz densa, podemos ver la misma estructura, aunque en lugar de mostrar espacios vacío se muestra el valor `undefined`:

```
var frutas = [ undefined, undefined , 'uva'];
```

7. Aunque estructuralmente puede no mostrar ninguna diferencia, sí que podemos encontrarla en el resultado obtenido para cada una de ellas. La matriz dispersa no tiene un índice 0 y, por tanto, si se evalúa, dará como resultado `false`, mientras que la matriz densa tiene un índice 0 y, por tanto, dará como resultado `true`.

8. El uso del operador `for` para conseguir la iteración de las matrices da el mismo resultado para los dos tipos de matrices. Por último diremos que, tal como veremos con mayor detalle en el ejercicio siguiente, el operador `forEach` ignora los huecos pero no los elementos `undefined`, por lo que en este caso ambos tipos de matrices son distintos.

IMPORTANTE

Las matrices que no contienen huecos se denominan en programación matrices densas. Estas matrices son continuas y tienen un elemento para cada índice; el primer índice es el 0 y el último en la propiedad `.length` el **-1**.

3

```
var frutas = ['manzana', , 'uva' ];
//El índice 1 en esta matriz es un hueco
```

Si el elemento vacío de la matriz se ubica en la última posición de la misma, tenga en cuenta que deberá escribir igualmente las dos comas que lo delimitan.

4

```
var frutas = [ , , 'uva'];
// Matriz dispersa
var frutas2 = [ undefined, undefined , 'uva'];
// Matriz densa
```

5

```
for (var i=0; i<frutas.length; i++) console.log(frutas[i]);
undefined
undefined
uva
//Uso de for en una matriz dispersa

for (var i=0; i<frutas2.length; i++) console.log(frutas2[i])
undefined
undefined
uva
//Uso de for en una matriz densa
```

Ambas matrices tienen la misma longitud (propiedad `length`), en este caso 3.

Operaciones para gestionar huecos

AUNQUE, COMO YA HEMOS INDICADO, LOS expertos aconsejan evitar en la medida de los posible los huecos en las matrices en JavaScript, en ocasiones será necesario trabajar con ellos y, en dichos casos, saber cómo gestionarlos. El lenguaje de programación JavaScript cuenta con distintos operadores que permiten o bien ignorar estos huecos o bien tenerlos en consideración para poder integrarlos perfectamente en nuestros scripts.

1. En este ejercicio veremos qué operadores podemos utilizar tanto para ignorar los huecos existentes en una matriz como para tenerlos en cuenta y manejarlos. Empecemos.
2. Los operadores `forEach()` y `every()` se saltan los huecos en una matriz al encontrarlos. La diferencia entre ambos es el tipo de resultado que generan al interpretar el script: el primero devolverá los índices para cada elemento con contenido y el segundo, devolverá un valor booleano, `true` o `false`
3. Un operador que ignora los huecos en matrices pero que los mantiene es `map()`, mientras que el operador `filter()` los elimina directamente de la matriz. En este caso, es decir, si lo que le conviene es que en lugar de huecos se devuelva un valor de undefined para estos elementos, también puede optar por generar una función personalizada que convierta precisamente los huecos en elementos undefined. Por ejemplo:

1

```
['manzana',, 'uva'].forEach(function (x,i) {
  console.log(i+'.'+x) })
0.manzana
2.uva
// Gestión de huecos con el operador forEach()

['manzana',, 'uva'].every(function (x) {
  return typeof x === 'string' })
true
// Gestión de huecos con el operador every()
```

El operador `some()` funciona de forma similar al operador `every()`.

2

```
['manzana',, 'uva'].map(function (x,i) {
  return i+'.'+x })
[ '0.manzana', , '2.uva' ]
// Gestión de huecos con el operador map()

['manzana',, 'uva'].filter(function (x) {
  return true })
[ 'manzana', 'uva' ]
// Gestión de huecos con el operador filter()
```

El operador `filter()` elimina los huecos de las matrices.

```
function convertirHuecos(frutas) {
  var resultado = [];
  for (var i=0; i < frutas.length; i++) {
   resultado[i] = frutas[i];
  }
 return resultado;
}
```

076

4. Otros métodos para gestionar los huecos en matrices son `join()`, que convierte no sólo los huecos sino también los elementos `undefined` y `null`, en cadenas de caracteres vacías, y `sort()`, que mantiene los huecos originales al mismo tiempo que realiza la correspondencia con el pertinente índice.

5. La gestión o manipulación de huecos en matrices también puede llevarse a cabo mediante el bucle for-in. Este bucle lista correctamente las propiedades de una matriz; dichas propiedades son un indicador de los índices de las matrices:

```
for (var key in ['manzana',, 'uva']) { console.log(key) }
// El resultado es 0 2
```

6. La última operación que deseamos mostrarle en este ejercicio es la función de prototipo apply(). ¿Cómo trabaja esta función? `apply()` transforma todos los huecos detectados en una matriz en argumentos cuyo valor es undefined. Esto significa que esta función también puede ser utilizada para generar matrices con elementos undefined. Vea a continuación un ejemplo de uso de esta función:

```
frutas.apply(null, frutas(3))
//Dará como resultado [ undefined, undefined, undefined ]
```

3

```
function convertirHuecos(frutas) {
  var resultado = [];
  for (var i=0; i < frutas.length; i++) {
   resultado[i] = frutas[i];
  }
 return resultado;
}
// El resultado de aplicar esta función e
[ 'manzana', undefined, 'uva' ]
```

6

```
function ejemplo1() { return [].slice.call(arguments) }
ejemplo1.apply(null, [ , , ,])
[ undefined, undefined, undefined ]
// La función ejemplo1 devuelve sus argumentos como matriz.
```

5

```
['manzana',, 'uva'].sort() // La propiedad length devuelve 3
[ 'manzana', 'uva', , ]
// Gestión de huecos con el operador sort()
```

4

```
['manzana',, 'uva'].join('-')
'manzana--uva'
[ 'manzana', undefined, 'uva' ].join('-')
'manzana--uva'
// Gestión de huecos con el operador join()
```

Añadir y eliminar elementos de una matriz

LOS MÉTODOS ARRAY.PROTOTYPE DE JAVASCRIPT PERMITEN llevar a cabo diferentes acciones sobre las matrices. Entre estas acciones se encuentran la posibilidad de añadir y eliminar elementos de una matriz, teniendo en cuenta que, en este caso, es decir, utilizando los métodos que se describirán a continuación, dicha matriz se verá modificada.

1. En este ejercicio veremos algunos de los métodos de prototipo para agregar y eliminar elementos de una matriz. Como hemos indicado en la introducción, los métodos aquí detallados están considerados destructivos, lo que significa que la matriz contenida en los métodos invocados se verá alterada en algún aspecto.
2. Empecemos por el método `Array.prototype.shift()`, con el cual se elimina el primer elemento de la matriz y se devuelve dicho elemento. El método `Array.prototype.shift()` altera la propiedad `length` de la matriz. 1
3. El método `Array.prototype.shift()` tiene su método opuesto, `Array.prototype.unshift()`, el cual añade un elemento al inicio de la matriz y da como resultado la nueva propiedad `length`. 2
4. El método `Array.prototype.pop()` elimina el último elemento de la matriz y lo devuelve como resultado, mientras que el

1

```
var frutas = ['manzana', 'pera', 'uva', 'naranja'];
var frutas2 = frutas.shift();
// El resultado será ['pera', 'uva', 'naranja']
```

2

```
var frutas = ['manzana', 'pera', 'uva', 'naranja'];
var frutas2 = frutas.unshift('mango');
// El resultado será 5, la nueva longitud de la matriz
```

método `Array.prototype.push()` realiza la acción contraria, es decir, añade el elemento especificado al final de la matriz. 3

5. El método `Array.prototype.splice()` cambia el contenido de una matriz eliminando los elementos existentes y, opcionalmente, sustituyéndolos por otros indicados. La sintaxis de este método incluye el índice en el cual debe iniciarse el cambio en la matriz y, también de manera opcional, un número entero que indica el número de los elementos que se deben eliminar. Si se realiza la sustitución de estos elementos eliminados por otros nuevos, estos se especificarán a continuación de los otros parámetros. 4
6. El resultado devuelto por este método es una matriz con los elementos eliminados. Si sólo se ha suprimido un elemento, la matriz mostrará dicho único elemento, mientras que si no se ha eliminado ninguno, el resultado será una matriz vacía. 5
7. El parámetro que contiene el índice de los elementos a eliminar es opcional, tal y como hemos indicado. Sin embargo, tenga en cuenta que si no se especifica ningún valor para este parámetro, todos los elementos situados detrás del índice que indica el inicio de la alteración de la matriz serán eliminados.
8. Si se especifica un número distinto de elementos a insertar que de elementos a eliminar, la matriz tendrá una longitud (`length`) distinta al finalizar la llamada del método.

3

```
var frutas = ['manzana', 'pera', 'uva', 'naranja'];
var frutas2 = frutas.pop();
// El resultado será ['manzana', 'pera', 'uva']


var frutas = ['manzana', 'pera', 'uva', 'naranja'];
var frutas2 = frutas.push('mango', 'mandarina');
// El resultado será 5, la nueva longitud de la matriz
```

El método `Array.prototype.push()` añade los nuevos elementos indicados al final de la matriz.

4

```
var frutas = ['manzana', 'pera', 'uva', 'naranja'];
var frutas2 = frutas. splice(1, 3, 'mango', 'platano');
// 1 indica el índice en el cual se inicia el cambio en la matriz;
// 3 indica el elemento que debemos eliminar;
// 'mango' y 'platano' son los elementos que se añadiran detrás del índice 1
// El resultado será ['manzana', 'pera', 'mango', 'platano']
```

5

```
var frutas = ['manzana', 'pera', 'uva', 'naranja'];
var frutas2 = frutas. splice(1, 0, 'mango', 'platano');
// El resultado que devuelve el método es [ ],
es decir, una matriz vacía.
```

Si el índice de inicio es mayor que la longitud de la matriz, dicho índice de inicio se equipara a la mencionada longitud.

Ordenar y alterar elementos en una matriz

SI EN EL EJERCICIO ANTERIOR HEMOS estudiado los métodos que utiliza JavaScript para añadir y eliminar elementos de una matriz, en éste continuaremos comprobando otros métodos que permiten modificar el contenido de las matrices. En concreto, se mostrarán los métodos `Array.prototype` destinados a ordenar y alterar los elementos de una matriz.

1. Todos los métodos descritos en este ejercicio son destructivos con las matrices que gestionan, lo que significa que alterarán su contenido y seguramente alguna de sus propiedades. Empezaremos por el método `Array.prototype.reverse()`. Dicho método, como su nombre indica, invierte el orden de los elementos de una matriz y devuelve como resultado una referencia a la matriz original (modificada). Esta inversión significa que el primer elemento será el último y el último, el primero.
2. Este método no contiene ningún parámetro y se aplica del modo siguiente: 1

```
var frutas = ['manzana', 'pera', 'uva'];
var frutas2 = frutas.reverse();
// El resultado de la variable será
['uva', 'pera', 'manzana']
```

3. Como puede comprobar, el uso y el resultado de este método resulta muy sencillo e intuitivo. Hablemos ahora de un segundo método que, si bien su uso es también muy sencillo, su aplicación puede resultar un tanto ambigua por distintas

1

```
var frutas = ['manzana', 'pera', 'uva'];
var frutas2 = frutas.reverse();
// El resultado de la variable será
['uva', 'pera', 'manzana']
```

2

```
var frutas = ['ananas', 'bananas', 'cereza'];
var frutas2 = frutas.sort();
// El resultado de la variable será
['cereza', 'ananas', 'banana']
```

razones. Se trata del método `Array.prototype.sort()`, el cual ordena los elementos de una matriz siguiendo por defecto el orden por puntos de código establecido en el estándar de codificación Unicode y devolviendo la matriz modificada. Este estándar reserva un nombre e identificador único para cada caracter o símbolo, lo que se denomina punto de código o *code point*. He aquí un ejemplo: 2

```
var frutas = ['ananas', 'bananas', 'cereza'];
var frutas2 = frutas.sort();
// El resultado de la variable será
['cereza', 'ananas', 'banana']
```

4. Alfabéticamente, la ordenación de los elementos correspondería al de la matriz frutas, y vea como tras la ordenación con el método `sort()` los elementos han quedado totalmente alterados, respondiendo al estándar de codificación Unicode.

5. Si los elementos contenidos en la matriz no son caracteres de texto sino números, el método `sort()` convierte previamente los valores en cadenas de caracteres y los compara con el estándar de codificación Unicode para alterar el orden de los elementos. Igual que en el caso del orden alfabético, el orden numérico resultará totalmente distinto tras la aplicación del método. Vea si no este ejemplo: 3

```
var edades = [1, 2, 10, 21];
edades.sort();
// El resultado tras la ordenación será [1, 10, 2, 21]
```

6. Como en el caso del método descrito anteriormente, el método `array.sort()` no contiene ningún parámetro obligatorio, aunque sí puede contener uno opcional, el que especifica una función que definirá cómo se realizará la ordenación de los elementos. 5

IMPORTANTE

En el método `array.prototype.sort()`, si se especifica como parámetro una función de comparación, los elementos de la matriz se ordenarán según el valor resultante de la función comparativa.

3

```
var edades = [1, 2, 10, 21];
edades.sort();
// El resultado tras la ordenación será [1, 10, 2, 21]
```

Siguiendo el orden numérico, los elementos de la variable edades se encontrarían en la posición adecuada.

4

```
var puntos = [4, 2, 5, 1, 3];
puntos.sort(function(a, b) {
  return a - b;
});
print(puntos);
```

El método `array.sort()` puede ser utilizado sin problemas con expresiones de función.

Dividir y juntar elementos en matrices

EXISTEN UNA SERIE DE MÉTODOS DEL tipo `array.prototype` destinados a manipular los elementos de una matriz de una forma no destructiva. Esto significa que dichos métodos no alterarán los elementos sino que, a menudo, devolverán nuevas matrices.

1. En este ejercicio veremos algunos de los métodos no destructivos para manipular matrices. Empezaremos por el método `Array.prototype.concat()`, con el cual se obtiene una nueva matriz formada por los elementos contenidos en el objeto en el cual ha sido llamada, seguido por cada argumento, los elementos de este argumento (en el caso en que el argumento sea una matriz) o el argumento mismo (en el caso en que el argumento no sea una matriz).
2. En el siguiente ejemplo puede ver la concatenación de dos matrices: 1

```
var frutas = ['ananas', 'bananas', 'cereza'];
var frutas2 = ['manzana', 'pera', 'uva', 'naranja'];
var batido = frutas.concat(frutas2);
```

3. Si lo que desea es copiar ciertos elementos de una matriz dentro de otra, entonces puede utilizar el método `Array.prototype.slice()`. Este método cuenta con dos parámetros: el primero indica el inicio de la extracción de elementos de la primera matriz y el segundo, el final de la mencionada extracción.

1

```
var frutas = ['ananas', 'bananas', 'cereza'];
var frutas2 = ['manzana', 'pera', 'uva', 'naranja'];
var batido = frutas.concat(frutas2);
// El resultado de la concatenación es la siguiente:
['ananas', 'bananas', 'cereza', 'manzana', 'pera', 'uva', 'naranja']
```

Sepa que la concatenación de matrices no se limita a dos, sino que puede utilizar del mismo modo el método `concat()` con más matrices.

2

```
var frutas1 = ['manzana', 'pera', 'uva', 'naranja'];
var batido = frutas.slice(1, 3);
// El resultado de la extracción es la siguiente:
['pera', 'uva']
```

4. En el siguiente ejemplo puede ver la extracción de una matriz y la creación de otra nueva: 2

```
var frutas1 = ['manzana', 'pera', 'uva', 'naranja'];
var batido = frutas.slice(1, 3);
```

5. Visto este ejemplo, podemos comprobar como, efectivamente, el método `Array.prototype.slice()` no altera la matriz original, sino que realiza una copia de la misma modificada. En el caso en que el parámetro final no se especifique en el método, se devolverá la misma propiedad length de la matriz original: 3

```
var frutas1 = ['manzana', 'pera', 'uva', 'naranja'];
var batido = frutas.slice(1);
// El resultado será ['pera', 'uva', 'naranja']
```

6. El método `Array.prototype.join()` genera una cadena de caracteres aplicando la función `toString()` a todos los elementos de la matriz. Cada uno de los elementos se mostrarán en la matriz separados por el signo indicado como parámetro en el método (por ejemplo, -). Si no se especifica este separador de elementos, se utilizará una coma como tal:

```
var frutas = ['manzana', 'pera', 'uva', 'naranja'];
var misFrutas = frutas.join();
// El resultado será 'manzana,pera,uva,naranja'
```

7. (Mire la imagen 4 para comprobar otras aplicaciones de este método.) 4 Con respecto al método `Array.prototype.join()`, debemos indicar que dicho método convierte los valores `undefined` y `null` en una cadena de texto vacía. Y lo mismo ocurre con los huecos o espacios vacíos, que también son convertidos en cadenas de texto vacías.

3

```
var frutas1 = ['manzana', 'pera', 'uva', 'naranja'];
var batido = frutas.slice();
// El resultado será ['manzana', 'pera', 'uva', 'naranja']
```

Si no existe ni el parámetro inicial ni el final, se respeta exactamente la matriz original.

4

```
var frutas = ['manzana', 'pera', 'uva', 'naranja'];
var misFrutas = frutas.join();
// El resultado será 'manzana,pera,uva,naranja'
var misFrutas = frutas.join(', ');
// El resultado será 'manzana, pera, uva, naranja'
var misFrutas = frutas.join(' + ');
// El resultado será 'manzana + pera + uva + naranja'
```

Buscar valores en una matriz

LOS MÉTODOS ARRAY.PROTOTYPE.INDEXOF() y `array.prototype.lastIndexOf()` SE utilizan en JavaScript para buscar valor específicos en matrices. Se trata de valores que no alteran en absoluto el contenido de la matriz rastreada, es decir, son métodos no destructivos.

1. En este sencillo ejercicio conocerá los dos métodos de matrices más utilizados para localizar o buscar valores en una matriz. Empezaremos por el método `Array.prototype.indexOf()`. Este método se basa en un valor o varios valores de búsqueda especificado entre los paréntesis del método y otro valor que indica el índice de la matriz a partir del cual debe empezar a rastrearse. La sintaxis de este método sería como sigue:

   ```
   array.indexOf(valorBusqueda[, desde])
   ```

2. El valor que devuelve el método `Array.prototype.indexOf()` es el primer índice en el que se localice el elemento especificado. 1 Si no se indica ningún índice de inicio, el rastreo empieza por el primer elemento; si no, por el índice especificado. 2 En el caso en que dicho elemento no se encuentre, el valor devuelto será -1. 3

3. Tenga en cuenta que si el índice especificado como inicio para la búsqueda es mayor que la propiedad length de la matriz, el resultado será -1, puesto que la búsqueda no podría realizarse.

1

```
var frutas = ['manzana', 'pera', 'uva'];
var mejor = frutas.indexOf('uva');
// El valor devuelto es 2
```

Recuerde que el valor devuelto por este método es el índice correspondiente al valor encontrado.

El valor no encontrado se refleja en el resultado -1.

3

```
var frutas = ['manzana', 'pera', 'uva'];
var mejor = frutas.indexOf('naranja');
// El valor devuelto es -1
```

2

```
var frutas = ['manzana', 'pera', 'uva'];
var mejor = frutas.indexOf('uva', 1);
// El valor devuelto es 0
```

En este caso, el valor devuelto es 0 porque se indica que la búsqueda se inicia en el índice 1.

En el caso en que este índice sea negativo, deberán restarse posiciones desde el final de la matriz y utilizar el elemento resultante como inicio para la búsqueda, teniendo en cuenta que dicha búsqueda continuará efectuándose de forma incremental.

4. El segundo método de matrices para localizar elementos o valores es, tal como hemos dicho en la introducción, `array.prototype.lastIndexOf()`. Se trata de un método más concreto que el anterior, puesto que devuelve el último índice en el cual se encuentra un determinado valor en una matriz y, además, la búsqueda se realiza hacia atrás. La sintaxis de este método es la siguiente:

```
arr.lastIndexOf(valorBusqueda[, desde = arr.length])
```

5. Como puede ver, en este caso también existe el parámetro opcional desde, que indica la posición a partir de la cual debe iniciarse la búsqueda. Sin embargo, este parámetro es ligeramente distinto al que hemos visto anteriormente, puesto que se basa en la propiedad `length` de la matriz. Por defecto, el índice de valores coincide con la longitud de la matriz, lo que significa que la búsqueda se realizará por todos los elementos.

6. Si el valor del índice es mayor o igual que la propiedad length de la matriz, la búsqueda también se realizará por todos los elementos. Sin embargo, si dicho índice es negativo, el recuento se lleva a cabo desde el último elemento de la matriz. E incluso en estos casos la búsqueda se realiza desde el final hacia el principio. Por último, cabe destacar que si el valor que se está buscando no se encuentra, el valor devuelto será -1.

4

```
var frutas = ['manzana', 'pera', 'uva', 'naranja'];
var mejor = frutas.lastIndexOf('pera');
// El valor devuelto es 2
```

5

```
var frutas = ['manzana', 'pera', 'uva', 'naranja'];
var mejor = frutas.lastIndexOf('cereza');
// El valor devuelto es -1
```

El método `array.prototype.lastIndexOf()` se basa en el método de igualdad estricta (===).

Examinar, transformar y reducir matrices

EN ESTE EJERCICIO LE MOSTRAREMOS UNA serie de métodos destinados a recorrer matrices en JavaScript. Todos estos métodos pueden dividirse según su aplicación: unos examinan las matrices, otros las transforman en diferentes aspectos y otros las reducen.

1. Con este ejercicio terminamos la sección dedicada a la manipulación de matrices en JavaScript. Empezaremos describiendo tres métodos para el examen de estos elementos.
2. El método `Array.prototype.forEach()`, que ya hemos mencionado en alguna ocasión, recorre los elementos de una matriz ejecutando una función especificada en cada uno de ellos. Esta función precisa tres argumentos: el correspondiente al elemento que se está procesando en la matriz, el del índice perteneciente a este elemento y el del nombre de la matriz sobre la cual se ha invocado el método `forEach()`. 1
3. El método `Array.prototype.every()` comprueba si todos los elementos de una matriz pasan el test implementado por la función especificada en dicho método. Igual que en el método anterior, esta función necesita tres argumentos: el valor, el índice y la matriz. Si los elementos comprobados dan como resultado `true`, el método continúa aplicando la función para las siguientes, hasta que se localiza un elemento que da como resultado `false`. 2
4. El método `Array.prototype.some()` es muy parecido al anterior, con la diferencia que éste devuelve como resultado `true`

1

```
function resultados(valor, índice, matriz) {
    document.write("valor: " + valor);
    document.write(" valor: " + índice);
    document.write("<br />");
}
var frutas = ['manzana', 'pera', 'uva', 'naranja'];
frutas.forEach(resultados);

// valor: manzana índice: 0
// valor: pera índice: 1
// valor: uva index: 2
// valor: naranja index: 3
```

2

```
function mayorQue(valor, índice, matriz) {
  return valor >= 10;
}
[30, 7, 18, 250, 94].every(mayorQue);   // false
[30, 70, 18, 250, 94].every(mayorQue); // true
```

Debe saber que la función especificada en el método `forEach()` no es invocada para índices de elementos eliminados, pero sí para elementos cuyo valor sea `undefined`.

si encuentra como mínimo un elemento verdadero para la función especificada. 3

5. Pasemos a los métodos de transformación incluidos en los tipos de métodos para recorrer matrices. El primero de ellos es `Array.prototype.map()`, que crea una nueva matriz con los resultados procedentes de una función invocada para cada uno de los elementos de la matriz. Los argumentos utilizados en la función invocada son los mismos que para los métodos tratados anteriormente. 4
6. Por su parte, el método `Array.prototype.filter()` da como resultado una matriz únicamente con los elementos originales que devuelven `true` ante la función especificada. A diferencia del resto de métodos estudiados en este ejercicio, `filter()` sólo necesita como parámetro la función que deben pasar los elementos de la matriz. Los elementos invocados por la función son sólo los que disponen de índice, no los elementos eliminados o los que no tienen valores. 5
7. Por último, hablemos de los métodos que permiten reducir matrices después de recorrerlas. Uno de estos métodos es `Array.prototype.reduce()`, el cual es capaz de aplicar una función a dos elementos de una matriz a la vez, de izquierda a derecha, con la intención de reducirlo a un único valor. La función de este método recibe cuatro argumentos: el valor inicial, el valor del elemento actual, el índice actual y la matriz sobre la cual se invoca la función.
8. El método `Array.prototype.reduceRigth()`se comporta igual que el anterior, pero aplicando la función especificada sobre los elementos de derecha a izquierda. 6

IMPORTANTE

En el método `Array.prototype.reduceRigth()` la primera vez que se llama a la función, el valor inicial y el valor actual puede ser uno de estos dos valores. Si se utiliza un valor inicial en la llamada a este método, el valor previo será igual a dicho valor y el valor actual será igual al último valor de la matriz.

3

```
function mayorQue10(valor, índice, matriz) {
  return valor > 10;
}
[2, 5, 8, 1, 4].some(mayorQue10);  // false
[12, 5, 8, 1, 4].some(mayorQue10); // true
```

4

```
var numeros = [ 1, 2, 3 ];
 numeros.map(function (x) {
  return 2 * x
 })
// El resultado devuelto será 2, 4, 6
```

5

```
function mayorQue(valor) {
  return valor >= 10;
}
var resultado = [12, 5, 8, 130, 44].filter(mayorQue);
// El resultado será [12, 130, 44]
```

6

```
var numeros = [0,1,2,3,4]
 numeros.reduceRight(function(valorInicial, valorActual, índice, array){
  return valorInicial + valorActual;
 });

// Resultado para cada parámetro:
valorInicial = 4, valorActual = 3, índice = 3
```

Expresiones regulares

LAS EXPRESIONES REGULARES SON PATRONES QUE se utilizan para localizar una determinada combinación de caracteres dentro de una cadena de texto. En JavaScript, las expresiones regulares también son objetos. Estos patrones son utilizados a través de los métodos `exec` y `test` de la función `RegExp`, así como los métodos `match`, `replace`, `search` y `split` de la función `String`.

1. Con este ejercicio empezamos una serie dedicada a las expresiones regulares en JavaScript. En estas páginas, sencillamente realizaremos una aproximación a estos elementos de combinación de caracteres. Empecemos.
2. La expresión regular más básica consiste en un caracter literal sencillo, como por ejemplo, una a. En la frase "Marta es profesora", esta expresión regular coincide con la primera aparición de este caracter, y concretamente, combina la letra "M" con la letra "a". 1
3. Seguramente usted ha utilizado, aun sin saberlo, expresiones regulares en alguna ocasión. En programación, por ejemplo, en DOS, cuando escribe la orden `dir*.*` con la intención de mostrar todos los archivos de un directorio, el patrón `*` coincide con cualquier cadena de caracteres. O en otro ejemplo, cuando insertamos la orden `*.txt` para buscar todos los archivos con la extensión txt en un directorio o carpeta. 2 Pues

1

```
var cadena = /d(b+)d/g;
var miCadena = myRe.exec("cdbbdbsbz");
```

2

```
C:\>dir *.com > log.txt

C:\>type log.txt

 El volumen de la unidad C es ST3_8420
 El número de serie del volumen es 3F50-15FB
 Directorio de C:\

COMMAND  COM         96,306  05-05-99 10:22p COMMAND.COM
         1 archivos         96,306 bytes
         0 directorios          377,71 MB libres

C:\>
```

bien, en JavaScript, la expresión regular equivalente a esta anotación sería `.*\.txt$`.

4. La potencia de las expresiones regulares se encuentra en la flexibilidad de los patrones, los cuales pueden ser confrontados con cualquier palabra o cadena de texto que tenga una estructura conocida. Por ejemplo, si deseamos encontrar en un listado el nombre exacto Julio, podemos utilizar como patrón este mismo nombre; sin embargo, si también queremos encontrar otros nombres como Julia o Julieta, necesitaremos otros tipos de caracteres para completar las expresiones.
5. Los patrones que se utilizan en las expresiones regulares están formados por un conjunto de caracteres, es decir, un grupo de letras, números o signos) o bien por un tipo de caracteres especiales, denominados metacaracteres.
6. Los metacaracteres representan otros caracteres y permiten una búsqueda contextual, es decir, aquella que no puede solventarse con cadenas literales. El porqué de la denominación "metacaracteres" se debe a que no se representan a ellos mismos, sino que son interpretados de un modo especial.
7. Los metacaracteres más utilizados en las expresiones regulares son doce:

\	^	$	.	|	?
*	+	(	)	[	]

En el ejercicio siguiente podremos comprobar toda la sintaxis de las expresiones regulares en JavaScript.

4

```
//Replace each \ with \\ so that C# doesn't treat \ as escape character
//Pattern: Start of string, any integers, 0 or 1 letter, end of word
string sPattern = "^[0-9]+([A-Za-z]\\b)?";
string sString = Row.Address1 ?? ""; //Coalesce to empty string if NULL
```

5

\	^	$	.	|	?
*	+	(	)	[	]

Estos son los metacaracteres básicos de las expresiones regulares en JavaScript.

Sintaxis de las expresiones regulares

EN EL EJERCICIO ANTERIOR HEMOS REALIZADO una primera aproximación a las expresiones regulares en JavaScript y hemos tenido una primera toma de contacto con los caracteres especiales básicos de dichas expresiones regulares. En este ejercicio veremos el significado de cada uno de estos caracteres y añadiremos muchos otros.

1. Como hemos indicado en el ejercicio anterior, son 12 los metacaracteres de las expresiones regulares: `\^$ .*+?( ){ }|`
2. El caracter `\` se utiliza junto con otro caracter literal para crear expresiones regulares especiales: `\d` es un método abreviado que se corresponde con un dígito de 0 a 9 (`[0-9]`); `\f` se corresponde con un salto de página, y `\r`, un retorno de carro.
3. El caracter `\` también se utiliza para "escapar" caracteres. Imagine que necesita utilizar uno de los metacaracteres como un caracter literal; si no se indica de alguna manera, el intérprete de JavaScript utilizará dicho metacaracter como caracter especial y no como su equivalente en caracter literal. Es en estos casos que debemos utilizar la barra inclinada `\` para escapar el caracter. Por ejemplo, si desea escribir la suma de los valores 2 y 8, deberá escapar el caracter especial + con el metacaracter `\`, puesto que sino se interpretará el signo + como el operador matemático que es. La expresión regular correcta será `2\+8=10`.

`\w-\`

1
```
/[\w-\.]{3,}@([\w-]{2,}\.)*([\w-]{2,}\.)[\w-]{2,4}/
```

Ésta sería una expresión regular que podría utilizarse en la programación de un formulario para validar la casilla destinada a la dirección de correo electrónico.

2
```
^\d{1,2}\/\d{1,2}\/\d{2,4}$
^(0[1-9]|1\d|2[0-3]):([0-5]\d):([0-5]\d)$
```

Éstas dos expresiones permitirían validar la fecha y la hora consecutivamente insertadas en un formulario.

4. El metacaracter `^` se corresponde con un principio de entrada o línea, mientras que el caracter especial `$` representa un fin de entrada o línea. Utilizamos el caracter `^` para indicar que el patrón debe aparecer al principio de la cadena de caracteres comparada y utilizamos el caracter `$` para indicar que dicho patrón debe aparecer al final del conjunto de caracteres. Dada su finalidad, estos dos caracteres se denominan metacaracteres de posicionamiento o anclas,
5. El punto (`.`) también puede utilizarse como metacaracter, en cuyo caso se utiliza para representar cualquier otro caracter, excepto una nueva línea. Recuerde que si lo que desea es buscar el signo de puntuación `.`, deberá escapar el caracter con la barra inclinada `\.`
6. Los metacaracteres `*`, `?` y `+` se denominan cuantificadores o multiplicadores, [3] y se aplican al caracter o grupo de caracteres que les preceden para indicar en qué número deben encontrarse en la cadena para que haya una coincidencia. Cada uno de estos cuantificadores también puede ser expresado del modo siguiente:

   `*` equivale a `{0,}` (0 o mas veces)
   `+` equivale a `{1,}` (1 o mas veces)
   `?` equivale a `{0,1}` (0 o 1 vez)
7. Los paréntesis `( )` y las llaves `{ }` también se consideran caracteres especiales. Estos se utilizan concretamente para indicar el número máximo y mínimo que debe darse para que exista una coincidencia. [4]
8. Por último, el metacaracter `|` se utiliza en las expresiones regulares para separar dos alternativas: `x|y`.

3

```
^(?:\+|-)?\d+$
^[0-9]{2,3}-? ?[0-9]{6,7}$
```

Expresiones regulares para validar en un formulario un campo en que deba insertarse un valor numérico entero y un campo en el cual deba introducirse un número de teléfono.

4

```
^(ht|f)tp(s?)\:\/\/[0-9a-zA-Z]
([-.\w]*[0-9a-zA-Z])*(:(0-9)*)*(\/?)(
[a-zA-Z0-9\-\.\?\,\'\/\\\+&%\$#_]*)?$
```

Y ésta sería una expresión regular que permitiría validar la introducción de una URL en un campo de un formulario.

Crear expresiones regulares

LAS EXPRESIONES REGULARES SON UNA DE las herramientas más útiles en cualquier lenguaje de programación. Aunque tienen múltiples aplicaciones, como por ejemplo la búsqueda de texto, una de las más utilizadas en JavaScript es el de la validación de los campos de un formulario. La correcta combinación de los metacaracteres y los caracteres literales son la fórmula para crear expresiones regulares que se comporten según lo esperado.

1. En este ejercicio y sabiendo que existen otras aplicaciones para las expresiones regulares en JavaScript, le mostraremos cómo generar una expresión que nos ayudaría a validar un campo de correo electrónico en un formulario. La idea es proporcionar paso a paso los distintos caracteres que conformarán esta expresión concreta y describir el porqué de su uso. Empecemos.

2. Teniendo en cuenta que la sintaxis de una dirección de correo electrónico es, habitualmente, la siguiente: **midireccion@dominio.sufijo**, una expresión regular que validaría un campo de este tipo en un formulario sería la siguiente:

```
^[a-z]+@[a-z]+\.[a-z]{2,4}$
```

3. Veamos por partes cómo hemos generado esta expresión regular. En primer lugar encontramos el metacaracter ^, el cual, como recordará, indica el inicio de la cadena de caracteres que deberá buscarse o identificar.

1

```
<script>
    function ValidarCadena() {
        email = "^[a-z]+@[a-z]+\.[a-z]{2,4}$";
        resultado = new RegExp(email);

        if (document.getElementById("textValidReg").value.match(resultado))
            alert("OK");
        else
            alert("No válido");
    }
</script>
```

Éste sería un ejemplo de un script en el cual se utiliza la expresión regular que hemos estudiado para validar el e-mail de un formulario.

4. A continuación, encontramos entre claudátors `[ ]` las letras `a-z`; los claudátors se utilizan para indicar un rango de caracteres, tanto números como letras. [2] En este caso, deseamos que esta primera parte de la dirección de correo electrónico muestre una palabra.
5. Con el metacaracter `+` incluido después de la cadena de caracteres estamos indicando que los caracteres precedentes se repetirán más de una vez, es decir, se podrán utilizar los caracteres alfabéticos necesarios para formar la palabra adecuada.
6. Seguidamente encontramos el símbolo `@`, éste utilizado con su sentido literal; es decir, así aparecerá en la cadena de caracteres coincidente.
7. Inmediatamente después, y para indicar el inicio de lo que será el dominio de la dirección de correo electrónico, repetimos la cadena `[a-z]+`, que tiene el mismo significado que en el uso descrito anteriormente, es decir, la utilizamos para designar la aparición de una cadena de caracteres alfabéticos un número de veces sin especificar.
8. Seguidamente, debemos indicar el punto que separa el dominio del sufijo en la dirección de e-mail. Sin embargo, no podemos incluir este signo sin más, puesto que el intérprete de JavaScript lo identificará como un metacaracter dentro de la expresión regular, con el significado que este conlleva. Por esta razón, necesitamos situar delante del punto una barra inclinada para escaparlo, `\.`. [3]
9. Después del punto, vuelve a aparecer la cadena `[a-z]` seguida de otra cadena, esta vez numérica: `{2,4}`. Con esta combinación de metacaracteres estamos indicando que la última parte de la dirección de e-mail mostrará entre 2 y 4 caracteres alfabéticos (com, net,org...).
10. El último metacaracter, `$`, nos indica el final de la expresión regular.

[2]

```
^([ ]*)$
```

Esta expresión puede utilizarse para buscar párrafos sin contenido, sólo con espacios.

[3]

```
\<[1-9][0-9]*\>
\<[0-9]+,[0-9]*\>
\<[0-9]+[,|\.][0-9]*\>
```

La expresión regular de la primera línea puede utilizarse para buscar números enteros; la segunda, para buscar números con decimales, y la tercera, para buscar números con decimales para los cuales no sabemos si se ha utilizado coma o punto como separador.

Trabajar con fechas en JavaScript

JAVASCRIPT PERMITE PROGRAMAR FUNCIONES para mostrar la fecha actual en una página en diferentes formatos. Para conseguir esta flexibilidad en el formato, es preciso combinar distintas variables con una serie de caracteres especiales que sirven de código, un código inventado por el usuario lo suficientemente claro para poder manejarlo con soltura y sencillez.

1. Éste será el primero de una serie de ejercicios dedicados al trabajo con fechas en JavaScript. Este lenguaje de programación permite llevar a cabo múltiples operaciones combinando funciones, métodos y constructores con el fin de mostrar y gestionar la fecha actual en nuestros documentos.
2. Debemos advertirle que el presente ejercicio es un ejemplo práctico del uso de variables y funciones en un script real. En los siguientes profundizaremos en el funcionamiento de constructores y métodos. Empecemos. En primer lugar podríamos escribir una función como la siguiente, la cual contendrá un argumento y lo almacenará en la variable `fecha` que contendrá un objeto *Date* de JavaScript:

```
function fechaCompleta(fecha) {
}
```

3. Seguidamente, definimos las distintas variables para cada parte de la fecha. La correspondiente al día de la semana sería como sigue: 1

```
var diaSemana=fecha.getDay()-1;
```

4. El método `getDay()` se encarga de seleccionar el elemento correcto de la matriz que crearemos a continuación con los días de la semana mediante el uso de valores de índice. A continuación añadiríamos la sentencia condicional que da sentido al

1

```
function fechaCompleta(fecha) {
 var diaSemana=fecha.getDay()-1
}
```

2

```
function fechaCompleta(fecha) {
 var diaSemana=fecha.getDay()-1;
  if (diaSemana <0){
    diaSemana=6
  }
}
```

El valor -1 en esta función se encarga de restar uno a los días de la matriz, teniendo en cuenta que para el método `getDay()`, el primer día de la semana es el domingo.

valor -1 de la sentencia anterior; con ella conseguiremos que el primer día de la semana sea lunes en lugar de domingo: 2

```
if (diaSemana <0){
 diaSemana=6
}
```

5. Seguiríamos con una variable que contendrá una matriz para almacenar los nombres de los días de la semana y otra variable que almacenará el resultado de las dos anteriores: 3

```
var dias = ['Lunes', 'Martes', 'Miércoles', 'Jueves',
 'Viernes', 'Sábado', 'Domingo'];
var nombreDia=dias[diaSemana];
```

6. Las variables que añadiríamos a continuación permitirían gestionar los elementos del mes en la fecha: 4

```
var dia=fecha.getDate();
var mes=fecha.getMonth();
var mesActual=mes+1;
```

7. La primera variable es la que genera el mes actual; sin embargo, esta generación la realiza mediante un valor de índice que, como ya sabe, empieza por 0, razón por la cual hemos creado la segunda variable, que indica la necesidad de sumar 1 al mes seleccionado. Las variables con los nombres de los meses serían como las siguientes:

```
var meses = ['enero', 'febrero', 'marzo', 'abril',
'mayo', 'junio', 'julio', 'agosto', 'septiembre', 'octubre', 'noviembre', 'diciembre'];
var nombreMes=meses[mes];
```

8. Y ya sólo nos faltaría configurar el año para que la fecha se mostrara completa: 5

```
var year=fecha.getFullYear()
```

3

```
function fechaCompleta(fecha) {
 var diaSemana=fecha.getDay()-1;
  if (diaSemana <0){
    diaSemana=6
  }
 var dias = ['Lunes', 'Martes', 'Miércoles', 'Jueves',
 'Viernes', 'Sábado', 'Domingo'];
 var nombreDia=dias[diaSemana];
}
```

4

```
function fechaCompleta(fecha) {
 var diaSemana=fecha.getDay()-1;
  if (diaSemana <0){
    diaSemana=6
  }
 var dias = ['Lunes', 'Martes', 'Miércoles',
 'Viernes', 'Sábado', 'Domingo'];
 var nombreDia=dias[diaSemana];
 var dia=fecha.getDate();
 var mes=fecha.getMonth();
 var mesActual=mes+1;
}
```

5

```
 var mesActual=mes+1;
 var meses = ['enero', 'febrero', 'marzo',
 'junio', 'julio', 'agosto', 'septiembre',
 var nombreMes=meses[mes];
 var year=fecha.getFullYear()
}
```

El método `getFullYear()` muestra el año con cuatro dígitos (por ejemplo, 2014).

El constructor Date

EXISTEN EN JAVASCRIPT DISTINTAS MANERAS DE llamar o invocar al constructor Date. Este constructor se encarga de obtener una fecha, el día y la hora actuales y poder manipularlas posteriormente. Para trabajar con fechas es preciso instanciar un objeto de la clase Date.

1. En este ejercicio realizaremos una primera aproximación al constructor Date. En JavaScript, disponemos de cuatro procedimientos para invocar al constructor Date. Veamos uno a uno en qué consisten.
2. La fórmula `new Date()` permite construir una fecha a partir de los datos indicados. Hay que tener en cuenta que la hora se interpreta según la zona horaria configurada en el equipo. `new Date()` puede contar con los siguientes parámetros: año; mes (de 0 a 11, teniendo en cuenta que el índice 0 corresponde al mes de enero); fecha (de 1 a 31); horas (de 0 a 23; minutos (de 0 a 59); segundos (de 0 a 59), y milisegundos (de 0 a 999). 1
3. Otro procedimiento para obtener la fecha actual con el constructor Date es utilizar una cadena de caracteres que contenga la fecha actual en formato digital, por ejemplo, 2015-02-27. Dicha cadena se convierte en número, el cual se utiliza para invocar al constructor `new Date(number)`. 2

1

```
new Date(2015, 1, 27, 14, 55)
  Date {Viernes Feb 27 2015 14:55:00 GMT+0100 (CET)}
```

Éste es un ejemplo de uso para obtener la fecha actual con el constructor `new Date()`.

Sepa que JavaScript ha heredado del lenguaje de programación Java la característica de utilizar los índices de 0 a 11 para los meses del año.

2

```
new Date('2015-02-27')
        Date {Viernes Febrero 29 2015 02:00:00 GMT+0200 (CEST)}
```

4. El siguiente procedimiento que deseamos tratar es el que utiliza un valor temporal para especificar el número de milisegundos desde el 1 de enero de 1970 a las 00:00:00: [3]

```
new Date(0)
   Date {Viernes Enero 01 1970 01:00:00 GMT+0100 (CET)}
```

5. Este procedimiento tienen un método opuesto; se trata del método `getTime()`, el cual devuelve como resultado de la invocación los milisegundos:

```
new Date(123).getTime()
// devuelve 123
```

6. En el caso de utilizar como argumento el valor `NaN` (Not a Number), el resultado será una instancia especial del objeto `Date`, en concreto, una fecha no válida. [4]

```
var fecha = new Date(NaN);
fecha.toString()
// Da como resultado una fecha no válida.
var fecha = new Date(NaN);
fecha.getTime() o bien fecha.getYear()
// Da como resultado NaN;
```

7. Por último, existe un cuarto procedimiento para obtener la fecha actual y consiste en el constructor `new Date()`, esta vez sin utilizar argumento alguno entre los paréntesis. Cabe destacar que este procedimiento funciona del mismo modo y proporciona los mismos resultados que el método `new Date(Date.now())`.

8. En el próximo ejercicio estudiaremos los distintos métodos para el constructor `Date` en JavaScript, y en el siguiente, trataremos los métodos para dicho constructor como prototipo.

3

```
new Date(0)
   Date {Viernes Enero 01 1970 01:00:00 GMT+0100 (CET)}
```

4

```
var fecha = new Date(NaN);
fecha.toString()
// Da como resultado una fecha no válida.
var fecha = new Date(NaN);
fecha.getTime() o bien fecha.getYear()
// Da como resultado NaN;
```

Según el método utilizado con el constructor `Date`, el argumento `NaN` proporciona resultados distintos.

Métodos del constructor Date

PARA CREAR UN OBJETO DATE CON un día y hora distintos de los actuales tenemos que indicar entre paréntesis el momento con que inicializar el objeto. Hay varias maneras de expresar un día y hora válidas, por eso podemos construir una fecha guiándonos por varios esquemas, como hemos visto en el ejercicio anterior.

1. En este ejercicio nos centraremos en los métodos que utiliza el constructor `Date`. En concreto son tres los métodos que describiremos a continuación: `Date.now()`, `Date.parse(cadenaFecha)` y `Date.UTC(año, mes, día...)`. Veamos con todo detalle cada uno de estos métodos.
2. El método `Date.now()` devuelve como resultado el número de milisegundos transcurridos desde las 00:00:00 UTC del 1 de enero de 1970. El valor utilizado como resultado para este método será siempre numérico. La sintaxis del método `Date.now()` sería la siguiente:

   ```
   var fechaMiliseg = Date.now()
   ```

3. Es preciso tener en cuenta que, cuando se utiliza este método para crear registros temporales o identificadores únicos, el sistema operativo Windows tiene una precisión de unos 15 milisegundos; esto significa que se podrían producir valores iguales si el método se invoca muchas veces seguidas, sin respetar un intervalo de tiempo prudencial.

1

```
var inicio = Date.now();
var respuesta = prompt("¿Cómo te llamas?", "");
var final = Date.now();
var tiempoEspera = (final - inicio) / 1000;
document.write("Has tardado " + elapsed + " segundos" + " en escribir " + response);
```

En este script se ha utilizado el método `Date.now()` en dos ocasiones para determinar el tiempo transcurrido hasta realizar una determinada acción.

2

```
var minutos = 1000 * 60;
var horas = minute * 60;
var dia = hour * 24;

date = new Date("1/1/2001");
var hora = date.getTime();

document.write(Math.round(hora / dia) + " días desde el 1/1/1970 hasta el 1/1/2001");
```

Ejemplo de uso en un script del método `getTime()`, equivalente al método `date.now()` para el constructor `Date`.

087

4. El método `date.now()` tiene el mismo efecto que el método del objeto `Date` `getTime()` 2
5. Pasemos ahora al segundo método que deseamos describir en este ejercicio. Se trata de `Date.parse()`, el cual transforma una cadena con la representación de una fecha y hora indicada como parámetro, y devuelve como resultado el número de milisegundos desde las 00:00:00 del 1 de enero de 1970, hora local.
6. Es preciso tener en cuenta que la cadena con la fecha que se utiliza como parámetro para el método `Date.parse()` debe encontrarse en el formato inglés, es decir, Febr 27, 2015. Dicho método es útil para establecer valores de fecha basados en cadenas de este tipo. Debido a que `parse` es un método estático de `Date`, es recomendable utilizarlo siempre como `Date.parse()`, en vez de como un método del objeto `Date`generado por usted. 2
7. Si el parámetro indicado entre los paréntesis del método `Date.parse()` no tiene el formato de fecha indicado, el resultado será `NaN`.
8. El último método del constructor Date que trataremos en este ejercicio es `Date.UTC(...)`. Este método puede contar con distintos parámetros, como el año, el mes, el día, la hora... De hecho, este método acepta los mismos parámetros que el formato más largo del constructor Date. El método Date.UTC() devuelve el número de milisegundos transcurridos desde las 00:00:00 del 1 de enero de 1970, hora universal. 4
9. El método `Date.UTC()` difiere del constructor en dos factores: `UTC` utiliza la fecha y la hora universales en lugar de locales y `UTC` devuelve el resultado como valor numérico en vez de como un objeto `Date`.

3

```
var fecha = "Noviembre 1, 1997 10:15 AM";
var result = Date.parse(fecha);
document.write(result);
// El resultado en milisegundos será 878404500000
```

4

```
var minutos = 1000 * 60;
var horas = MinMilli * 60;
var dias = HrMilli * 24;

var fecha = new Date("June 1, 1990");
var year = date.getFullYear();
var mes = date.getMonth();
var dia = date.getDay();
```

```
var nuevoDia = new Date("January 16, 2020");
var yearActual = newDay.getUTCFullYear();
var mesActual = newDay.getUTCMonth();
var diaMesActual = newDay.getUTCDate();

var t1 = Date.UTC(year, mes - 1, dia)
var t2 = Date.UTC(yearActual, mesActual, diaMesActual);

var diasPasados = (t2 - t1) / diasMilisegundos;

document.write(dias);
// El resultado será 10848
```

Métodos para el prototipo Date

SI EN EL EJERCICIO ANTERIOR HEMOS estudiado los métodos que utiliza JavaScript para el constructor `Date`, en éste nos ocuparemos de otros métodos concretos para el constructor `Date.prototype` de JavaScript.

1. Si deseamos obtener y definir una fecha basada en la hora local, podemos utilizar `Date.prototype.get«Unit»()` y `Date.prototype.set«Unit»(number)`, donde `Unit` puede ser un año (`FullYear`), un mes (`Month`), un día del mes (`Date`), una hora (`Hours`), un minuto (`Minutes`), un segundo (`Seconds`) o un milisegundo (`Milliseconds`). `Date.prototype.get«Unit»()` devuelve como resultado el valor indicado en `Unit` según la hora local; por su parte, `Date.prototype.set«Unit»(number)` define el valor de `Unit` también según la hora local.
2. Si lo que desea es obtener o definir este mismo resultado pero según la hora universal, entonces los métodos a utilizar serán `Date.prototype.getUTC«Unit»()` y `Date.prototype.setUTC«Unit»(number)`.
3. El método `Date.prototype.getTime()` devuelve el valor numérico correspondiente a la hora para una fecha determinada según la hora universal. Podemos utilizar este método para asignar una fecha y una hora a otro objeto Date, teniendo en cuenta que es funcionalmente equivalente al método `valueOf()`:

1
```
var fecha = new Date('2015-02-27');
    Date {Fri Febr 27 2015 01:00:00 GMT+0100 (CET)}
 fecha.getDate() // El resultado será 27
 fecha.getDay() // El resultado será 5
```

2
```
var hoy = new Date();
var year = hoy.getUTCFullYear();
```

3
```
var hoy = new Date();
var mes = hoy.getUTCMonth();
```

088

```
var aniversario = new Date(2015, 12, 21);
var copiar = new Date();
copiar.hora(aniversario.getTime());
```

4. Este script nos podría servir para construir un objeto del tipo Date con valores idénticos de tiempo. 4

5. Otro método constructor de prototipo para Date es `Date.prototype.setTime(valorHora)`, con el cual podemos obtener la fecha en milisegundos desde el 1 de enero de 1970 a las 00:00:00 UTC.

6. Por su parte, el método `Date.prototype.valueOf()` es, como hemos indicado unos pasos más arriba, el equivalente a `getTime()`. En concreto, este método se utiliza cuando una fecha se convierte en un valor numérico y devuelve el valor primitivo del objeto `Date`. Cabe señalar que este método normalmente es invocado de forma interna por Javascript, no explícitamente en el código. 5

7. El método `Date.prototype.getTimezoneOffset()` para el constructor `Date` devuelve la diferencia entre la hora local y la hora universal en minutos. Hay que tener en cuenta que el resultado obtenido por este método será positivo si la zona horaria local se encuentra dentro del UTC, mientras que dicho resultado será negativo si la zona horaria se encuentra fuera del UTC. Por ejemplo, si se indica como zona horaria local UTC+10 (Australian Eastern Standard Time), el resultado obtenido será -600: 6

```
var fecha = new Date();
var diferencia = fecha.getTimezoneOffset();
```

8. Por último queremos señalar que `Year` ya no se utiliza como unidad, sino que en su defecto deberá escribir `FullYear`.

4

```
var aniversario = new Date(2015, 12, 21);
var copiar = new Date();
copiar.hora(aniversario.getTime());
```

6

```
var fecha = new Date();
var diferencia = fecha.getTimezoneOffset();
```

5

```
var fecha = new Date(56, 6, 17);
var mifecha = fecha.valueOf();
// El método valueOf() asigna el valor numérico -424713600000
a la variable miFecha
```

Recuerde que el valor devuelto por este método son milisegundos.

Formatos para la fecha

JAVASCRIPT PERMITE EL USO DE CARACTERES especiales para programar la aparición de la fecha actual en pantalla en diferentes formatos.

1. En este ejercicio veremos un ejemplo de cómo podríamos definir un script para que la fecha mostrada en pantalla apareciera en distintos formatos. Una vez generadas las variables correspondientes a los diferentes elementos que formarán la fecha actual, será necesario añadir como parámetro la palabra formato, que corresponderá a una nueva variable que gestionará las cadenas de caracteres para el formato: 1

```
function fechaCompleta(fecha,formato) {
...
}
```

2. La nueva variable formato podría tener la siguiente sintaxis: 2

```
formato=formato.replace(/#DIASEMANA#/g,nombreDia);
```

3. Veamos parte por parte qué significa esta sentencia. `formato` es el nombre de la variable, cuyo contenido será ejecutado por el método `replace()`. Este método toma la expresión regular indicada y sustituye todo cuanto coincide con otra cadena de caracteres. En este ejemplo, la cadena de caracteres es #DIASEMANA#, que será sustituida en el documento por el contenido de la variable `nombreDia`, es decir, por el día de la semana actual. Ahora se trata de ir añadiendo nuevas líneas para la

1
```
function fechaCompleta(fecha, formato) {
 var diaSemana=fecha.getDay()-1;
  if (diaSemana <0){
    diaSemana=6
  }
 var dias = ['Lunes', 'Martes', 'Miércoles', 'Jueves',
 'Viernes', 'Sábado', 'Domingo'];
 var nombreDia=dias[diaSemana];
 var dia=fecha.getDate();
 var mes=fecha.getMonth();
```

2
```
 var mes=fecha.getMonth();
 var mesActual=mes+1;
 var meses = ['enero', 'febrero', 'marzo', 'abril', 'mayo',
 'junio', 'julio', 'agosto', 'septiembre', 'octubre', 'noviembre', 'diciembre'];
 var nombreMes=meses[mes];
 var year=fecha.getFullYear()
 formato=formato.replace(/#DIASEMANA#/g,nombreDia);
}
```

variable `formato` con cada una de las variaciones de formato, siguiendo la estructura anterior:

```
formato=formato.replace(/#diasemana#/g,nombreDia.slice(0,3));
```

4. Con esta nueva cadena conseguiremos que el día de la semana muestre en este caso sólo 3 caracteres, gracias al método `slice()`:

```
formato=formato.replace(/#DIA#/g,dia);
```

5. Ésta será la variable para el día del mes, es decir, en formato numérico. Podríamos definir el formato para los meses de esta manera:

```
formato=formato.replace(/#MESES#/g,nombreMes);
formato=formato.replace(/#meses#/g,nombreMes.slice(0,3));
formato=formato.replace(/#MES#/g,nombreMes);
```

6. El primer formato reflejará el mes en letras, el segundo, con sólo 3 letras y el tercero lo hará en números. Para indicar el código para el año, podríamos escribir lo siguiente: 3

```
formato=formato.replace(/#YEAR#/g,year);
formato=formato.replace(/#year#/g,year.toString().slice(-2));
```

7. Sepa que existen otras formas para conseguir mostrar la fecha actual en distintos formatos; nosotros hemos elegido ésta por ser más gráfica. Los métodos `Date.parse()` y `new Date()` se utiliza para parsear los formatos, mientras que el método `Date.prototype.toISOString()` genera una cadena de caracteres con el formato de fecha completo:

```
new Date().toISOString()
'2015-12-21T14:23:47.320Z'
```

3

```
var dias = ['Lunes', 'Martes', 'Miércoles', 'Jueves',
'Viernes', 'Sábado', 'Domingo'];
var nombreDia=dias[diaSemana];
var dia=fecha.getDate();
var mes=fecha.getMonth();
var mesActual=mes+1;
var meses = ['enero', 'febrero', 'marzo', 'abril', 'mayo',
'junio', 'julio', 'agosto', 'septiembre', 'octubre', 'noviembre', 'diciembre'];
var nombreMes=meses[mes];
var year=fecha.getFullYear()
formato=formato.replace(/#DIASEMANA#/g,nombreDia);
formato=formato.replace(/#diasemana#/g,nombreDia.slice(0,3));
formato=formato.replace(/#DIA#/g,dia);
formato=formato.replace(/#MESES#/g,nombreMes);
formato=formato.replace(/#meses#/g,nombreMes.slice(0,3));
formato=formato.replace(/#MES#/g,nombreMes);
formato=formato.replace(/#YEAR#/g,year);
formato=formato.replace(/#year#/g,year.toString().slice(-2));
}
```

Formatos para mostrar la hora actual

JAVASCRIPT PERMITE UTILIZAR DIFERENTES FORMATOS PARA mostrar la hora actual. En este ejercicio nos ocuparemos de describir dichos formatos y de mostrar algunos ejemplos de uso de los mismos en este lenguaje de programación.

1. En este sencillo ejercicio trabajaremos con los formatos de hora disponibles en JavaScript. Para empezar, sepa que los formatos disponibles tienen las siguientes estructuras:

   ```
   THH:mm:ss.sss
   THH:mm:ss.sssZ
   THH:mm:ss
   THH:mm:ssZ
   THH:mm
   THH:mmZ
   ```

2. La letra T es un caracter literal, no un dígito, que indica el inicio del formato horario. Las letras `HH` se refieren a la hora, que puede ir de 00 a 23. Las letras `mm` indican los minutos en el formato digital y pueden ir de 00 a 59, y las letras `ss` indican lógicamente los segundos, que también se establecen el rango 0-59. Por su parte, las letras `sss` se refieren a los milisegundos y van de 000 a 999.

3. La letra `z` que aparece al final de algunos de los formatos especificados se refiere a la zona horaria y se trata de una información opcional. La Z corresponde a la zona UTC, mientras que también se pueden utilizar los signos + y - seguidos de una hora concreta.

4. Como ya hemos visto en ejercicios anteriores, el objeto Date de JavaScript no sólo permite mostrar la fecha actual en el

1

```
var hora = new Date();
var horas = hora.getHours();
var min = hora.getMinutes();
var seg = hora.get.Seconds();
document.write(horas + ":" + min + ":" + seg);
```

programa del usuario sino también la hora. Los métodos `getHours()`, `getMinutes()` y `getSeconds()` permiten mostrar la hora, los minutos y los segundos respectivamente:

```
var hora = new Date();
var horas = hora.getHours();
var min = hora.getMinutes();
var seg = hora.get.Seconds();
document.write(horas + ":" + min + ":" + seg);
```

5. Sin embargo, el script mostrado en el paso anterior no mostrará dos dígitos en el caso de que los minutos y los segundos sean menos que 10 (08, por ejemplo), lo que dará lugar a un formato un tanto extraño (10:2:5 en lugar de 10:02:05). Para ello, podemos utilizar una sencilla sentencia condicional que dicte que, si los minutos son menores que 10, se añada un 0 antes del valor correspondiente:

```
var min = hora.getMinutes();
 if (min<10) {
  min = '0' + min;
 }
```

6. Y podemos llevar a cabo el mismo proceso para definir los segundos en dos dígitos:

```
var seg = hora.get.Seconds();
 if (seg<10) {
  seg = '0' + seg;
 }
```

7. Sepa que también puede gestionar de un modo muy parecido el formato de las horas en 12 o 24:

```
if (horas<12) {
 formatoHora = 'am';
} else {
  formatoHora = 'pm';
}

horas = horas % 12;
 if (horas==0) {
  horas = 12;
 }
```

2

```
var hora = new Date();
var horas = hora.getHours();
var min = hora.getMinutes();
 if (min<10) {
  min = '0' + min;
 }
var seg = hora.get.Seconds();
 if (seg<10) {
  seg = '0' + seg;
 }
document.write(horas + ":" + min + ":" + seg);
```

Trabajar con valores temporales

AQUELLO QUE EN LA INTERFAZ API se denomina tiempo (time), la especificación ECMAScript denomina valores temporales (time value. Estos valores son un número primitivo que codifica una fecha como milisegundos desde las 00:00:00 horas del 1 de enero de 1970. Cada objeto del tipo fecha almacena su estado como un valor temporal en una propiedad interna.

1. En este último ejercicio dedicado al objeto y al constructor Date hablaremos de los valores temporales de ECMAScript. La propiedad interna que utiliza el objeto Date para almacenar su estado como valor temporal es la misma que utilizan las instancias de los constructores envolventes `Boolean`, `Number` y `String` para almacenar sus valores primitivos envolventes.
2. Muchos de los métodos descritos en ejercicios anteriores trabajan con valores temporales: `new Date()`, el cual utiliza valores temporales para generar una fecha; `Date.parse()`, 1 que devuelve un valor temporal tras parsear una cadena que contiene una fecha; `Date.now()` 2 que devuelve la fecha actual como valor temporal; `Date.UTC()`, 3 el cual interpreta los parámetros relativos al UTC y devuelve un valor temporal; `Date.prototype.getTime()`, que devuelve el valor temporal almacenado en el receptor `Date.prototype.setTime()`, y `Date.prototype.valueOf()`, que también devuelve el valor temporal almacenado en el receptor.

1

```
var fecha = "Noviembre 1, 1997 10:15 AM";
var result = Date.parse(fecha);
document.write(result);
// El resultado en milisegundos será 878404500000
```

2

```
var inicio = Date.now();
var respuesta = prompt("¿Cómo te llamas?", "");
var final = Date.now();
var tiempoEspera = (final - inicio) / 1000;
document.write("Has tardado " + elapsed + " segundos" + " en escribir " + response);
```

3

```
var nuevoDia = new Date("January 16, 2020");
var yearActual = newDay.getUTCFullYear();
var mesActual = newDay.getUTCMonth();
var diaMesActual = newDay.getUTCDate();

var t1 = Date.UTC(year, mes - 1, dia)
var t2 = Date.UTC(yearActual, mesActual, diaMesActual);

var diasPasados = (t2 - t1) / diasMilisegundos;

document.write(dias);
// El resultado será 10048
```

091

3. El rango de número en enteros en JavaScript es extremadamente amplio; podríamos decir que empieza aproximadamente 285.616 años antes de 1970 y termina aproximadamente 285.616 años después de 1970.

4. Seguidamente, le proporcionamos algunos ejemplos en los cuales convertimos una fecha en un valor temporal, utilizando para ello el método `getTime()`, estudiado en ejercicios anteriores: [4]

```
new Date('1970-01-01').getTime()
// La conversión daría 0
 new Date('1970-01-02').getTime()
// La conversión daría 86400000
 new Date('1960-01-02').getTime()
// La conversión daría -315532800000
```

5. En cambio, podemos utilizar el constructor `Date()` para realizar la operación inversa, es decir, para convertir un valor temporal en una fecha. Veamos unos cuantos ejemplos de ello: [5]

```
new Date(0)
// Date {Thu Jan 01 1970 01:00:00 GMT+0100 (CET)}
new Date(24 * 60 * 60 * 1000)
// Date {Fri Jan 02 1970 01:00:00 GMT+0100 (CET)}
new Date(-315532800000)
// Date {Sat Jan 02 1960 01:00:00 GMT+0100 (CET)}
```

6. JavaScript nos permite realizar otras conversiones en las que intervienen fechas. Por ejemplo, podemos convertir una fecha en un valor numérico mediante el método `Date.prototype.valueOf()`, el cual, como hemos indicado anteriormente, devuelve como resultado un valor temporal. [6]

4

```
new Date('1970-01-01').getTime()
// La conversión daría 0
 new Date('1970-01-02').getTime()
// La conversión daría 86400000
 new Date('1960-01-02').getTime()
// La conversión daría -315532800000
```

6

```
new Date('2015-05-18')
// Se convierte en new Date('2015-05-18')
true
```

5

```
new Date(0)
// Date {Thu Jan 01 1970 01:00:00 GMT+0100 (CET)}
new Date(24 * 60 * 60 * 1000)
// Date {Fri Jan 02 1970 01:00:00 GMT+0100 (CET)}
new Date(-315532800000)
// Date {Sat Jan 02 1960 01:00:00 GMT+0100 (CET)}
```

El objeto Math y sus propiedades

EL OBJETO DE JAVASCRIPT MATH ES un objeto intrínseco que proporciona funciones matemáticas y constantes básicas. Los parámetros que conlleva este objeto son dos: propiedades y métodos. Y es sobre las primeras que tratará en exclusiva este ejercicio, dejando para el siguiente el estudio de los métodos.

1. Las propiedades del objeto `Math` guardan valores que podrían ser necesarios en algún momento si se realizan cálculos matemáticos. Es probable que estas propiedades resulten un poco confusas a aquellas personas que no trabajan habitualmente con las matemáticas avanzadas; sin embargo, aquellos que las conozcan sabrán de su utilidad.
2. Siete son las propiedades o constantes del objeto `Math` de JavaScript: `Math.E`, `Math.LN2`, `Math.LN10`, `Math.LOG2E`, `Math.LOG10E`, `Math.PI`, `Math.SQRT1_2` y `Math.SQRT2`. Veamos una a una cuáles son sus características.
3. La propiedad `Math.E` es una constante matemática e, que representa el número de Euler. Dicho número es la base de los logaritmos naturales y su valor aproximado es 2.718. Debido a que `e` es una propiedad estática del objeto `Math`, siempre deberá utilizarse como `Math.e` y nunca crearse como objeto. 1

```
function getNapier() {
  return Math.E
}
```

4. Las propiedades `Math.LN2` y `Math.LN10` indican respectivamente los logaritmos naturales 2 y 10. 2 El primero tiene un

1

```
function getNapier() {
  return Math.E
}
```

2

```
function getNatLog2() {
   return Math.LN2
}
```

valor aproximado de 0.693 y el segundo, de 2.302. Estas propiedades no se pueden ni sobrescribir, ni numerar, ni configurar:

```
function getNatLog2() {
   return Math.LN2
}
```

5. Del mismo modo, las propiedades `Math.LOG2E` y `Math.LOG10E` representan la base 2 y 10 del logaritmo natural E. El valor aproximado de `Math.log2e` es de 1.443 y el de la propiedad `Math.LOG10E` es de 0.434. [3]

```
function getLog2e() {
  return Math.LOG2E
}
```

6. Otra de las propiedades del objeto matemático `Math` de JavaScript es `Math.pi`, que representa la relación entre la circunferencia de un círculo y su diámetro. Se trata del número pi, por lo que su valor aproximado es de 3.14159. [4]

```
var radio = 3;
var area = Math.PI * radio * radio;
document.write(area);

// El resultado será 28.274333882308138
```

7. Las dos últimas propiedades de Math son `Math.SQRT1_2` y `Math.SQRT2`, las cuales representan respectivamente la raíz cuadrada de 0,5 y la raíz cuadrada de 2. En el caso de `Math.sqrt1_2`, también representa de forma equivalente el valor 1 dividido entre la raíz cuadrada de 2.

8. El valor aproximado para la propiedad `Math.SQRT1_2` es de 0.707 y el valor para `Math.SQRT2` es de 1.414.

3

```
function getLog2e() {
  return Math.LOG2E
}
```

4

```
var radio = 3;
var area = Math.PI * radio * radio;
document.write(area);

// El resultado será 28.274333882308138
```

Funciones numéricas

EN EL EJERCICIO ANTERIOR HEMOS PODIDO describir las propiedades del objeto Math de JavaScript. Además de estas propiedades, el segundo parámetro incluido en estos objetos son los métodos, entre los cuales se encuentran las funciones numéricas y trigonométricas.

1. Aquellos lectores que disfruten con las matemáticas o que las conozcan y utilicen habitualmente, verán que este ejercicio no supone ninguna dificultad para ellos, puesto que los métodos que describiremos en estas páginas están basados en operaciones matemáticas típicas aunque ligeramente complejas.
2. El primer método del objeto Math que deseamos tratar es `Math.abs()`; esta función devuelve el valor absoluto de un número, es decir, el valor después de eliminar el signo + o - que le precedía.
3. Seguidamente, hablamos del método `Math.ceil()`, el cual devuelve el entero igual o inmediatamente superior de un número especificado:

```
Math.ceil(6.999) == 7
Math.ceil(6.001) == 7
Math.ceil(-6.001) == -6
Math.ceil(6.000) == 6
```

4. Otro método que representa una función numérica del objeto `Math` es `Math.exp()`. Este método devuelve el resultado de elevar el número E a un número especificado. Esta función re-

1

```
var caso1;
var valor1 = Math.abs(15);
var valor2 = Math.abs(-15);
 if (valor1 == valor2) {
   document.write("Los valores absolutos son iguales.");
 } else {
   document.write("Los valores absolutos son distintos.");
 }
// El resultado sería: Los valores absolutos son iguales
```

2

```
Math.exp(-1); // 0.36787944117144233
Math.exp(0);  // 1
Math.exp(1);  // 2.718281828459045
```

presenta la inversa de la función `Math.log()`, que trataremos más adelante en esta página.

5. La función numérica `Math.floor()` devuelve como resultado un número igual al especificado o bien inmediatamente inferior. Es decir, que se trata de la función inversa a `Math.ceil()`.

```
Math.floor(6.999) == 6
Math.floor(6.001) == 6
Math.floor(-6.001) == -7
Math.floor(6.000) == 6
```

6. Ya hemos mencionado la función numérica `Math.log()`; se trata de un método que devuelve el logaritmo neperiano de un número. [3] Representa la función inversa al método `Math.exp()`.

7. Otra de las interesantes funciones numéricas del objeto Math es `Math.pow()`. Este método, a diferencia de cuantos hemos visto hasta el momento en este ejercicio, trabaja con dos parámetros, devolviendo como resultado el primero de ellos elevado al segundo:

```
Math.pow(7, 2) == 41
Math.pow(36, 0.5) == 6
```

8. El método `Math.round()` se utiliza para redondear el número especificado al entero más próximo: [4]

```
Math.round(8.999) == 9
Math.round(8.001) == 8
Math.round(8.5) == 9
Math.round(-8.5) == -8
```

9. Y la última función numérica del objeto Date que deseamos considerar en este ejercicio es `Math.sqrt()`, que devuelve la raíz cuadrada de un número especificado, teniendo en cuenta que si este valor es negativo el resultado será el valor `NaN`:

```
Math.sqrt(256) == 16
```

3

```
var numeros = [ 45.3, 69.0, 557.04, 0.222 ];
for (i in numeros) {
    document.write(Math.log(numeros[i]));
    document.write("<br/>");
}
// Los resultados obtenidos para cada valor son:
// 3.8133070324889884
// 4.23410650459726
// 6.322637050634291
// -1.505077897109857 5
```

4

```
Math.round(8.999) == 9
Math.round(8.001) == 8
Math.round(8.5) == 9
Math.round(-8.5) == -8
```

Funciones trigonométricas

IMPORTANTE

Tenga en cuenta que al utilizar las funciones trigonométricas descritas en este ejercicio, se pueden producir errores de precisión, como por ejemplo obtener un decimal próximo a 0 en lugar de 0 u obtener un número muy grande en lugar de indeterminación por infinito.

JAVASCRIPT DISPONE DE MÉTODOS TRIGONOMÉTRICOS QUE aceptan y devuelven valores de ángulos como radianes. Las funciones trigonométricas se sirven de nuevo del objeto `Math`, el cual es importante saber que siempre tomará sus argumentos en radianes. También resultante imprescindible saber que siempre deberá utilizar este objeto como predefinido, nunca deberá crearlo desde cero.

1. En este ejercicio veremos otros métodos del objeto `Math`, esta vez para trabajar con funciones trigonométricas. Las funciones que trataremos son las siguientes: `Math.acos(x)`, `Math.asin(x)`, `Math.atan(x)`, `Math.atan2(y,x)`, `Math.cos(x)`, `Math.sin(x)` y `Math.tan(x)`. Veamos una a una en qué consisten y cómo se utilizan.
2. La función `Math.acos(x)`es una función trigonométrica inversa, la cual devuelve el arcocoseno de x. En este caso, el valor de x está expresado en radianes. Un ejemplo de uso de esta función sería la siguiente: 1

```
alert('Arcococseno de -1 es ' +Math.acos(-1));
//El valor Math.acos(-1) será 3.141592653589793
```

3. Similarmente a la función anterior, la función `Math.asin(x)` Devuelve el arcoseno de x, con el valor de x también expresado en radianes. 2
4. La siguiente función, `Math.atan(x)`, devuelve la arcotangente del valor numérico x; dicho valor también está expresado en radianes:

1

```
alert('Arcocoseno de -1 es ' +Math.acos(-1));
// El valor Math.acos(-1) será 3.141592653589793
```

Sepa que si el valor devuelto por esta función está fuera del rango especificado dentro de la función misma, dicho resultado será `NaN`.

2

```
alert('Arcoseno de 1 es (90 grados o pi/2) ' +Math.asin(1));
//El arcoseno de 1 es 1.570796326794896
```

```
Math.atan(1);  // 0.7853981633974483
Math.atan(0);  // 0
```

5. Éste sería un ejemplo de conversión de resultados con la función `Math.atan(x)`. De igual forma, la función `Math.atan2(y,x)` devuelve la arcotangente del cociente entre dos valores numéricos (x e y); en este tipo de funciones, el cociente representa un valor en radianes: 3

```
Math.atan2(90, 15); // 1.4056476493802699
Math.atan2(15, 90); // 0.16514867741462683
Math.atan2(±0, -0); // ±PI
Math.atan2(±0, +0); // ±0
Math.atan2(±0, -x); // ±PI for x > 0
```

6. Estos son sólo algunos ejemplos básicos de conversión realizados con esta función. La diferencia esencial entre ésta y la función `Math.atan(x)` es que la primera pasa los argumentos de `x` e `y` por separado, mientras que la segunda pasa la relación de ambos argumentos.

7. La siguiente función trigonométrica que deseamos describir es `Math.cos(x)`, la cual devuelve el coseno de x, con el valor de x expresado en radianes. El valor devuelto por esta función debe estar comprendido entre -1 y 1:

```
Math.cos(0);           // 1
Math.cos(1);           // 0.5403023058681398
Math.cos(Math.PI);     // -1
Math.cos(2 * Math.PI); // 1
```

8. Por su parte y del mismo modo, la función `Math.sin(x)` devuelve el seno de x y la función `Math.tan(x)` devuelve la tangente de x. 4 En ambos casos, el valor numérico de x se encuentra representado en radianes:

```
Math.sin(0); // 0
Math.sin(1); // 0.8414709848078965
Math.sin(Math.PI / 2); // 1
```

3

```
Math.atan2(90, 15); // 1.4056476493802699
Math.atan2(15, 90); // 0.16514867741462683
Math.atan2(±0, -0); // ±PI
Math.atan2(±0, +0); // ±0
Math.atan2(±0, -x); // ±PI for x > 0
```

En este caso, la función se encuentra expresada en grados en lugar de en radianes, como es habitual; sin embargo, hemos incluido otra función que realiza la conversión de grados a radianes antes de devolver la tangente.

4

```
function getTanDeg(deg) {
  var rad = deg * Math.PI/180;
  return Math.tan(rad);
}
```

Otras funciones matemáticas

EL OBJETO MATH, COMO HEMOS VISTO en los dos ejercicios anteriores, permite acceder a constantes de uso habitual como el número pi, o realizar cálculos trigonométricos, redondeos, logaritmos, potencias, etc. En este ejercicio veremos y describiremos otras funciones distintas a las numéricas y las trigonométricas (descritas en los ejercicios anteriores) que utiliza este objeto.

1. La primera de las funciones que deseamos tratar en este último ejercicio dedicado al objeto `Math` es la función `min()`, la cual devuelve el número más pequeño de entre los especificados entre paréntesis. Así de sencillo. JavaScript tiene en cuenta si los valores especificados son positivos o negativos para realizar esta comparación.

2. En el caso en que no se proporcione ningún argumento para esta función, el valor devuelto será `-Infinity`, mientras que si alguno de los argumentos especificados no es un valor numérico, el resultado será `NaN`. A continuación le mostramos algunos casos de uso de la función `min()`: 1

```
var caso1 = Math.min(5, 10); // Valor devuelto: 5
var caso2 = Math.min(0, 150, 30, 20, 38); // Valor devuelto: 0
var caso3 = Math.min(-5, 10); // Valor devuelto: -5
var caso4 = Math.min(-5, -10); // Valor devuelto: -10
var caso5 = Math.min(1.5, 2.5); // Valor devuelto: 1.5
```

3. Así como la función `min()` que acabamos de describir devuel-

1

```
var caso1 = Math.min(5, 10);
// Valor devuelto: 5
var caso2 = Math.min(0, 150, 30, 20, 38);
// Valor devuelto: 0
var caso3 = Math.min(-5, 10);
// Valor devuelto: -5
var caso4 = Math.min(-5, -10);
// Valor devuelto: -10
var caso5 = Math.min(1.5, 2.5); //
Valor devuelto: 1.5
```

La función `min()` del objeto `Math` es soportada por todos los navegadores web: Internet Explorer, Safari, Chrome, Opera y Firefox.

ve el valor más pequeño entre los argumentos proporcionados, la función `max()` devuelve como resultado el valor más alto. También igual que la función `min()`, en el caso en que no se proporcione ningún argumento para la función `max()`, el valor devuelto será `-Infinity`, mientras que si alguno de los argumentos especificados no es un valor numérico, el resultado será `NaN`. He aquí algunos ejemplos de uso y sus resultados: [2]

```
var caso1 = Math.max(5, 10); // Valor devuelto: 10
var caso2 = Math.max(0, 150, 30, 20, 38); // Valor devuelto: 150
var caso3 = Math.max(-5, 10); // Valor devuelto: 10
var caso4 = Math.max(-5, -10); // Valor devuelto: -5
var caso5 = Math.max(1.5, 2.5); // Valor devuelto: 2.5
```

4. La última función que queremos mencionar en este ejercicio es la función `random()`, la cual devuelve un número aleatorio superior a 0 (incluido) e inferior a 1. Debe saber que el generador de números aleatorios se inicializa automáticamente cada vez que se carga JavaScript. [3] Por ejemplo, si queremos obtener un número aleatorio situado entre el 1 y el 10, podemos escribir lo siguiente:

   ```
   Math.floor((Math.random() * 10) + 1);
   ```

5. Y el resultado que podría devolver dicha función podría ser:

   ```
   3
   ```

6. Ahora imagine que desea obtener un número aleatorio entre el 1 y el 100, es decir, en un rango más amplio de valores. Con el siguiente script podría conseguirlo rápidamente:

   ```
   Math.floor((Math.random() * 100) + 1);
   ```

7. Y obtener como resultado una cifra como

   ```
   40
   ```

2

```
var caso1 = Math.max(5, 10);
// Valor devuelto: 10
var caso2 = Math.max(0, 150, 30, 20, 38);
// Valor devuelto: 150
var caso3 = Math.max(-5, 10);
// Valor devuelto: 10
var caso4 = Math.max(-5, -10);
// Valor devuelto: -5
var caso5 = Math.max(1.5, 2.5);
// Valor devuelto: 2.5
```

Éste es un script utilizado para generar una batería de preguntas en un cuestionario, de manera que dichas preguntas aparezcan de forma aleatoria.

3

```
function cuest() {
 for (var i=0; i<preguntas.length; i++) {
  askQuestion(aleat);
  aleatorio = Math.floor(Math.random() *preguntas.length);
  aleat = preguntas[aleatorio];
 }
 var mensaje = 'Has contestado correctamente ' + puntos;
 mensaje += ' de ' +preguntas.length;
 mensaje += ' preguntas';
 var respuestas = document.getElementById('resultados');
 respuestas.innerHTML = mensaje;
}
```

Qué es el JSON

IMPORTANTE

En diciembre de 2005, el servidor Yahoo! empezó a dar soporte opcional a JSON en algunos de sus servicios web.

JSON ES EL ACRÓNIMO DEL TÉRMINO en inglés JavaScript Object Notation. Se trata de un formato ligero para el intercambio de datos, un subconjunto de la notación literal de objetos de JavaScript que no necesita utilizar el formato XML. Gracias a su simplicidad, el uso de este sistema de notación de objetos se ha generalizado mucho entre los programadores, sobre todo como alternativa a XML en AJAX.

1. En este ejercicio, realizaremos una aproximación al JSON [1] y trataremos de demostrar su relación directa con JavaScript. Empezaremos analizando por qué JSON presenta ciertas ventajas sobre el formato XML. Una de estas ventajas es que, como formato de intercambio de datos, resulta mucho más sencillo escribir un analizador sintáctico (conocido como parser) de JSON. [2]
2. Además, en JavaScript, un texto JSON puede ser analizado de forma sencilla mediante la función eval(), de la cual ya hemos hablado en este libro. Este análisis directo del formato JSON, junto a la ubicuidad de JavaScript en prácticamente todos los navegadores web, ha contribuido a que dicho formato haya sido aceptado por gran parte de la comunidad de desarrolladores AJAX.

1

Logotipo del formato JSON.

Los expertos dicen que, en teoría, es trivial analizar JSON en JavaScript usando la función JSON.parse() incorporada en el lenguaje. En la práctica, las consideraciones de seguridad por lo general recomiendan no usar eval() sobre datos crudos, sino que debería utilizarse un analizador JavaScript distinto para garantizar la seguridad.

2

```
caso1 = JSON.parse(json_datos);
```

3. El formato JSON se utiliza normalmente en entornos en los cuales el tamaño del flujo de datos entre cliente y servidor es muy importante, como es el caso de servidores como Yahoo o Google, que atienden a millones de usuarios.[3] También, cuando el origen de los datos es 100x100 fiable y en aquellos casos en que no sea tan importante no disponer de procesamiento XSLT para modificar los datos en el cliente.

4. No siempre el formato JSON aparece como opuesto o aventajado ante al formato XML, sino que en muchas ocasiones se requiere una complementación de ambos formatos, como podría ser en una aplicación de cliente que proporcionara información del tiempo en el protocolo estándar SOAP junto a mapas procedentes de Google Maps.

5. A continuación le mostramos un ejemplo de cómo se crearía un menú básico con JSON:

```
{"menu": {
  "id": "file",
   "value": "Archivo",
   "popup": {
     "menuitem": [
        {"value": "Nuevo", "onclick": "CrearNuevo()"},
      {"value": "Abrir", "onclick": "AbrirArchivo()"},
     {"value": "Cerrar", "onclick": "CerrarArchivo()"}
      ]
    }
  }
}
```

6. Vea la imagen 4 para visualizar la estructura del mismo menú escrita en XML.[4]

Yahoo y Google son sólo dos ejemplos de servidores que utilizan JSON como analizador sintáctico.

```
<menu id="file" value="Archivo">
    <popup>
      <menuitem value="Nuevo" onclick="CrearNuevo()" />
      <menuitem value="Abrir" onclick="AbrirArchivo()" />
      <menuitem value="Cerrar" onclick="CerrarArchivo()" />
    </popup>
</menu>
```

096

IMPORTANTE

Resulta interesante saber que, aun estando altamente difundido en los medios de programación, el término JSON en realidad se encuentra mal descrito, puesto que es sólo una parte de la definición del estándar ECMA-262 en que se basa Javascript. Por esta razón, ni Yahoo ni Google utilizan el nombre JSON, sino LJS (*Literal JavaScript*).

Métodos utilizados por el formato JSON

SI EN EL EJERCICIO ANTERIOR HEMOS realizado una presentación del formato JSON, en éste veremos algunos de los métodos utilizados por dicho formato. En concreto trataremos los métodos `JSON.stringify()`, `toJSON()` y `JSON.parse()`.

1. Empezamos este ejercicio con el método `JSON.stringify()`. Este método sencillamente convierte un valor al formato JSON. Según los parámetros especificados en el método, también puede realizarse un reemplazo de valores tras la conversión o bien se incluirán las propiedades especificadas mediante el array `replacer`.
2. La conversión realizada por el método `stringify()` cuenta con las siguientes características: los objetos `Boolean`, `Number`, y `String` son convertidos a sus valores primitivos, según la conversión semántica tradicional; si durante la conversión, el método se encuentra el valor `undefined`, una función o un símbolo, estos valores se omitirán, en el caso en que se encuentren en un objeto, o bien serán devueltos como `null`, en el caso en que formen parte de una matriz o array.
3. Seguidamente proporcionamos algunos ejemplos de uso básico del método JSON.stringify(): 1

```
JSON.stringify({}); // El valor devuelto es '{}'
JSON.stringify(true); // El valor devuelto es 'true'
JSON.stringify('foo'); // El valor devuelto es '"foo"'
JSON.stringify([1, 'false', false]); // El valor devuelto es '[1,"false",false]'
JSON.stringify({ x: 5 }); // El valor devuelto es '{"x":5}'
```

1

Estos son otros ejemplos un poco más complejos de uso de `JSON.stringify()`.

```
JSON.stringify({ x: 5, y: 6 });
// El valor devuelto es '{"x":5,"y":6}'
o bien '{"y":6,"x":5}'
JSON.stringify([new Number(1), new String('false'), new Boolean(false)]);
// El valor devuelto es '[1,"false",false]'
```

4. El método puede utilizar el parámetro opcional replacer, tal y como hemos comentado anteriormente. Este parámetro puede ser tanto una función como una matriz. Como función, toma dos parámetros: la clave y el valor que debe ser convertido. En el caso en que el parámetro replacer actúe como matriz, los valores de dicha matriz indicarán los nombres de las propiedades en el objeto que quedará incluido en la cadena de caracteres JSON resultante (vea la estructura de la función replacer en la imagen 2 para poder entender el resultado de la conversión:

```
JSON.stringify(caso1, ['semana', 'mes']);
//El resultado es '{"semana":32,"mes":3}' puesto que sólo mantiene las propiedades de los objetos "semana" y "mes".
```

5. El método `JSON.stringify()` utiliza el también método `toJSON()` para realizar las conversiones. Se trata de un miembro integrado del objeto `Date` de JavaScript que devuelve una cadena de fecha con formato ISO para la zona horaria UTC (indicada mediante el sufijo Z).

6. El último método que queremos analizar en este ejercicio es `JSON.parse()`. Este método analiza una cadena de texto como JSON y la convierte opcionalmente. Es preciso tener en cuenta que si la cadena de caracteres indicada para analizar no es válida, el intérprete lanzará una excepción del tipo `SyntaxError`. Vea un ejemplo de uso básico de este método:

```
JSON.parse('{}'); // El valor devuelto es {}
JSON.parse('true'); // El valor devuelto es true
JSON.parse('"foo"'); // El valor devuelto es "foo"
JSON.parse('[1, 5, "false"]'); // El valor devuelto es [1, 5, "false"]
JSON.parse('null'); // El valor devuelto es null
```

2

```
function replacer(clave, valor) {
  if (typeof valor === "string") {
    return undefined;
  }
  return value;
}
var caso1 = {fundación: "Mozilla", modelo: "caja",
semana: 32, medio: "coche", mes: 3};
var json = JSON.stringify(caso1, replacer);
//El resultado de la conversión será
{"semana":32,"mes":3}
```

3

```
var contacto = new Object();
contacto.nombre = "Ana";
contacto.apellido = "Luna";
contacto.telefono = ["21548563"];

contacto.toJSON = function(key)
 {
    var cambio = new Object();
    for (var val in this)
    {
        if (typeof (this[val]) === 'string')
            cambio[val] = this[val].toUpperCase();
        else
            cambio[val] = this[val]
    }
    return cambio;
};

var json = JSON.stringify(contacto);
```

Funciones globales no constructoras

A LO LARGO DE ESTE LIBRO hemos podido aprender numerosos aspectos acerca del uso y la aplicación de las funciones globales estándar en JavaScript: matrices, booleanos, números, funciones, fechas, objetos, expresiones regulares y cadenas de caracteres. Todas estas funciones han sido descritas como constructoras. Sin embargo, debe saber que existen un serie de funciones globales estándar que no son constructoras, y será acerca de éstas que tratará el siguiente ejercicio.

1. Las funciones globales no constructoras de JavaScript se suelen categorizar según su uso. Así encontramos funciones para codificar y descodificar texto y para categorizar y analizar números. Veamos en qué grupo podemos colocar cada una de estas funciones.
2. La función no constructora `encodeURI()` codifica una cadena de texto como identificador de recursos uniforme (URI) válido. El argumento que presenta esta función es obligatorio y representa un URI codificado.
3. Esta función tiene su función opuesta, `decodeURI()`. Si se codifica una cadena de texto con la función `encodeURI()` y se pasa este resultado por la función `decodeURI()`, el resultado será la cadena original.

```
var uriEncode = encodeURI ("http://www.Not a URL.com");
var uriDecode = decodeURIComponent(uriEncode);
document.write(uriEncode);
document.write("<br/>");
document.write(uriDecode);
//El resultado es http://www.Not%20a%20URL.com
//El resultado es http://www.Not a URL.com
```

1

```
var uriEncode = encodeURI ("http://www.Not a URL.com");
var uriDecode = decodeURIComponent(uriEncode);
document.write(uriEncode);
document.write("<br/>");
document.write(uriDecode);
//El resultado es http://www.Not%20a%20URL.com
//El resultado es http://www.Not a URL.com
```

4. La función `encodeURIComponent()` codifica una cadena de texto como un componente válido de un validador uniforme de recursos. A diferencia de la función descrita en pasos anteriores, `encodeURI()`, la función `encodeURIComponent()` codifica todos y cada uno de los caracteres de la cadena pasa como argumento, lo que significa que deberá tener especial cuidado si utiliza, por ejemplo, nombres de carpetas o de archivos en el texto a codificar; en dichos casos, la barra diagonal que suele separar estos elementos también se codificará y no serán válidos si se envían a un servidor web. Lo más aconsejable en estos casos es utilizar la función `encodeURI()`: [2]

```
var codificar = encodeURIComponent ("www.Not a URL.
com");
var descodificar = decodeURIComponent(uriEncode);
document.write(codificar);
// El resultado será www.Not%20a%20URL.com
```

5. Hablemos ahora de las funciones no constructoras para analizar y categorizar valores numéricos. De hecho, todas ellas han sido descritas en algún momento en este libro; sin embargo, haremos un rápido recordatorio de las mismas. Se trata de las siguientes funciones: `isFinite(número)`, `isNaN(valor)`, `parseFloat(cadena)` y `parseInt()`.

6. La función `isFinite()` permite especificar sin un número tiene o no el valor Infinity. La función `isNaN()` comprueba si un valor está considerado o no como numérico. La función `parseFloat()` convierte su argumento y devuelve como resultado un valor numérico con punto flotante, es decir, con decimales. [3] Por último, la función `parseInt()` convierte su argumento e intenta devolver como resultado un número entero basado en una raíz especificada. [4]

2

```
var codificar = encodeURIComponent ("www.Not a URL.com");
var descodificar = decodeURIComponent(uriEncode);
document.write(codificar);
// El resultado será www.Not%20a%20URL.com
```

3

```
parseFloat("3.14");
parseFloat("314e-2");
parseFloat("0.0314E+2"); v
var cadena = "3.14"; parseFloat(cadena);
parseFloat("3.14más caracteres no dígitos");
```

En todos estos casos, el valor devuelto por la función `parseFloat()` será 3.14.

En todos estos casos, el valor devuelto por la función `parseInt()` será 15.

4

```
parseInt("F", 16);
parseInt("17", 8);
parseInt("15", 10);
parseInt(15.99, 10);
parseInt("FXX123", 16);
parseInt("1111", 2);
parseInt("15*3", 10);
parseInt("12", 13);
```

Evaluar código dinámicamente

JAVASCRIPT PONE A DISPOSICIÓN DEL USUARIO diferentes métodos para evaluar el código escrito en este lenguaje de programación. Uno de estos métodos es la archiconocida función `eval()`, la cual convierte una cadena en un código válido y después la ejecuta. El segundo método es `new Function()`, que crea una función que devuelve la suma de sus parámetros.

1. En este ejercicio veremos cómo utilizar las funciones indicadas en la introducción de este ejercicio y trataremos de analizar cuál es la mejor opción para evaluar el código de JavaScript. Empecemos.
2. Muchas de las aplicaciones que utilizamos a diario no se podrían haber realizado sin el uso de la función `eval()`. ¿Cómo funciona `eval()`? Dicha función analiza una cadena de código JavaScript sin hacer referencia a ningún objeto en particular. Está considerada una función de alto nivel y no está asociada a ningún objeto.
3. El argumento pasado en la función `eval()` es una cadena (string). Si esta cadena es una expresión, la función la evalúa, mientras que si se trata de una o más sentencias JavaScript, `eval()` las ejecuta:

```
eval(new String("4 + 4"));
// devuelve un objeto del tipo String que contiene "4 + 4"
eval("4 + 4");
// Devuelve el valor 8, la suma de 4 + 4
```

4. Debe tener en cuenta que no es posible la aplicación de `eval()` con expresiones aritméticas; la razón es que Javascript evalúa este tipo de expresiones automáticamente. Y en el caso en que

1

```
eval(new String("4 + 4"));
// devuelve un objeto del tipo String que contiene "4 + 4"
eval("4 + 4");
// Devuelve el valor 8, la suma de 4 + 4
```

el argumento de la función `eval()` no sea una cadena, el resultado será la cadena original, es decir, no se producirá ningún cambio.

5. La función `eval()` presenta una serie de limitaciones que pasamos a detallar a continuación. `eval()` no puede ser invocada indirectamente con un nombre distinto a su original, ante el riesgo de que aparezca un error en tiempo de ejecución. El siguiente script, pues, no sería válido: 2

```
var caso1 = 2;
var caso2 = 4;
var resultado = eval;
resultado("caso1 + caso2");
```

6. También es importante no utilizar la función `eval()` para convertir los nombres de propiedades en propiedades. De hecho, esta aplicación de la función está desaconsejada y se apuesta por los denominados operadores de miembros para realizar esta misma acción.

7. En la introducción de este ejercicio hemos hablado también de la función `new Function()` para evaluar dinámicamente el código JavaScript. En este caso, esta función puede utilizarse para crear una nueva función que devuelve como resultado la suma de sus parámetros, especificados como argumentos: 3

```
var caso1 = new Function('x', 'y', 'devuelve x+y');
caso1(3, 4) // El valor devuelto sería 7
```

8. Los expertos opinan que es mejor utilizar `new Function()` en lugar de la función `eval()` para evaluar el código JavaScript. La razón es que los parámetros incluidos como argumentos para la función `Function()` proporcionan un esquema evidente para el código evaluado. Así, no es necesario basarse en la sintaxis de la función indirecta `eval()` para asegurarse de que el código evaluado puede acceder sólo a variables globales.

2

```
var str = "if (x == 5) {alert('z is 42'); z = 42;} else z = 0; ";
document.write("<P>z es ", eval(str));
```

En este ejemplo se utiliza la función `eval()` para evaluar la cadena de texto que contiene la variable `str`.

3

```
var caso1 = new Function('x', 'y', 'devuelve x+y');
caso1(3, 4) // El valor devuelto sería 7
```

Unicode y JavaScript

IMPORTANTE

Resulta interesante saber que la palabra Unicode procede de los tres objetivos perseguidos en el momento de su concepción por los tres responsables del lenguaje: universalidad, uniformidad y unicidad.

EN ESTE ÚLTIMO CAPÍTULO REALIZAREMOS UNA aproximación al lenguaje Unicode en relación con JavaScript. Son dos los procedimientos en los cuales JavaScript manipula el lenguaje Unicode: uno es de forma interna, durante el análisis del código, y otro de forma externa, en el momento de descargar un archivo.

1. Empezaremos este ejercicio definiendo el lenguaje Unicode y explicando brevemente su historia. En 1987, nace la idea de crear un conjunto de caracteres universal. Los responsables de esta idea fueron Joe Becker, Lee Collins y Mark Davis, ingenieros de Apple y Xerox, los cuales un año después lanzaron un primer borrador del nuevo lenguaje. 1
2. Hoy en día, Unicode es un estándar de codificación de caracteres diseñado para facilitar el tratamiento informático, transmisión y visualización de textos de múltiples lenguajes y disciplinas técnicas, además de textos clásicos de lenguas muertas.
3. Este estándar especifica un nombre e identificador numérico único para cada carácter o símbolo, el denominado *code point* o punto de código', además de otras informaciones necesarias para su uso correcto: direccionalidad, mayúsculas y otros atributos. Los caracteres alfabéticos, ideográficos y símbolos son tratados de forma equivalente; esto significa que pueden ser mezclados en un mismo texto sin la introducción de marcas o caracteres de control. 2
4. Los puntos de código de Unicode se identifican por un número entero. Según su arquitectura, un ordenador utilizará unidades de 8 (UTF-8), 16 (UTF-16) o 32 (UTF-32) bits para

1

Logotipo del estándar de codificación Unicode.

2

representar dichos enteros. Las formas de codificación de Unicode reglamentan la forma en que los puntos de código se transformarán en unidades tratables por el computador.

5. Unicode reemplaza los esquemas de codificación de caracteres existentes, muchos de los cuales están muy limitados en tamaño y son incompatibles con entornos plurilingües. Tras este reemplazo, Unicode se ha convertido en el estándar más extenso y completo, erigiéndose como dominante en la internacionalización y adaptación local del software informático. El estándar ha sido implementado en un número considerable de tecnologías recientes, que incluyen XML, Java, JavaScript y sistemas operativos modernos.

6. ¿Cómo manipula JavaScript Unicode en su codificación habitual? Como ya hemos indicado en la introducción de este ejercicio, este lenguaje de programación utiliza Unicode tanto internamente como externamente. Tenga en cuenta que internamente el código fuente JavaScript es tratado como una secuencia de unidades de código UTF-16. Además, si hablamos de identificadores, cadenas literales y expresiones regulares, cada una de las unidades de código utilizada también puede ser expresada mediante una secuencia de escape Unicode del tipo \uHHHH, donde HHHH representan cuatro dígitos hexadecimales: 3

```
var caso1\u006F\u006F = 'abc';
caso2 // 'abc'
```

7. En el caso del uso externo del estándar Unicode por parte de JavaScript, diremos que cuando se utiliza internamente el código UTF-16, el código fuente de JavaScript normalmente no se almacena en este formato. Si tiene interés para conocer más o absolutamente todo acerca de este estándar de codificación, no dude en visitar la siguiente dirección web: **www.unicode.org**. 4

3

```
var caso1\u006F\u006F = 'abc';
caso2 // 'abc'
```

Tenga en cuenta que es posible utilizar caracteres Unicode en cadenas de texto literales y en nombres de variables, sin para ello abandonar el rango ASCII en el código fuente.

4

Para continuar aprendiendo...

SI ESTE LIBRO HA COLMADO SUS EXPECTATIVAS

Este libro forma parte de una colección en la que se cubren los programas informáticos de más uso y difusión en todos los sectores profesionales.

Aunque el libro que tiene usted entre las manos es, en cuanto a su planteamiento, ligeramente distinto a la mayoría de los que componen esta colección, mucho más teórico, gran parte de ellos son principalmente prácticos. Si con éste le hemos ayudado a profundizar en el lenguaje de programación JavaScript, no se detenga aquí, en las páginas siguientes encontrará otros títulos de la colección que pueden ser de su interés.

DIBUJO VECTORIAL

Si su interés está más cerca del dibujo vectorial y la ilustración, entonces su libro ideal es "Aprender Illustrator CC con 100 ejercicios prácticos".

Illustrator es una excelente herramienta de dibujo asistido por ordenador. En los 100 ejercicios de este libro se estudian en profundidad las principales herramientas de creación y edición de esta aplicación.

Con este libro:

- Conozca las ventajas de trabajar en la nube con las aplicaciones de Creative Cloud
- Descubra cómo puede transformar sus ilustraciones con más libertad que nunca
- Sincronice colores desde su aplicación Kuler para iPhone con el nuevo panel Kuler de Illustrator
- Aproveche las mejoras en la localización de vértices en los distintos tipos de pinceles

PROGRAMACIÓN WEB

Si su interés está más cerca de los lenguajes de programación para web, entonces su libro ideal es "Aprender HTML5, CSS3 y JavaScript con 100 ejercicios prácticos".

HTML5, CSS3 y JavaScript se están consolidando como el estándar para el desarrollo de proyectos web. HTML5 aporta la estructura y contenidos de la página web; CSS3 define los estilos o el aspecto de los elementos, y JavaScript programa las acciones que realizarán determinados elementos en condiciones concretas.

Con este libro:

- Utilice las nuevas etiquetas semánticas de HTML5 para crear completos sitios web
- Defina colores, tipografías, sombras, etc. mediante estilos CSS3
- Cree efectos generados por eventos con los métodos y las propiedades de JavaScript
- Aprenda a programar un reproductor de vídeo

las acciones que realizarán determinados elementos en condiciones concretas.

Con este libro:

- Utilice las nuevas etiquetas semánticas de HTML5 para crear completos sitios web
- Defina colores, tipografías, sombras, etc. mediante estilos CSS3
- Cree efectos generados por eventos con los métodos y las propiedades de JavaScript
- Aprenda a programar un reproductor de vídeo

COLECCIÓN APRENDER...CON 100 EJERCICIOS PRÁCTICOS

DISEÑO Y CREATIVIDAD ASISTIDOS

- 3ds Max 2014
- 3ds Max 2012
- 3ds Max 2012 Avanzado
- AutoCAD 2014
- AutoCAD 2013
- Illustrator CC
- Illustrator CS6
- InDesign CC
- Dreamweaver CS5
- Dreamweaver CS6
- Maya 2014
- Maya 2012 Avanzado
- Photoshop CC
- Photoshop CS6
- Retoque fotográfico con Photoshop CC
- Retoque fotográfico con Photoshop CS6
- Integración entre Photoshop, Illustrator e InDesign

INTERNET

- Internet Explorer 8
- HTML5, CSS3 y JavaScript

OFIMÁTICA

- Access 2013
- Excel 2013
- Fórmulas y funciones para Excel 2010
- Office 2013
- PowerPoint 2013
- Outlook 2010
- Word 2014

SISTEMAS OPERATIVOS

- Las novedades de Windows 7
- Windows 8

EDICIÓN DE VIDEO

- Final Cut Pro 7
- Premiere Pro CS5.5
- After Effects CS5.5
- After Effects Avanzado CS5.5
- Pinnacle Studio