ceviche.js

Prueba los ejemplos en línea

Capítulo 1: Entendiendo JavaScript

> *La empresa La Buena Espina es una cadena de restaurantes de comida peruana que logró crecer gracias al boom gastronómico local. De tener un local familiar ahora tienen varios restaurantes en todo el país, por lo que decidieron lanzar un sitio web donde muestren información sobre su carta y sus locales.*
>
> *El dueño de La Buena Espina te ha pedido personalmente realizar el sitio y quiere que visitarla sea una experiencia tan buena como su comida, así que es tu deber como desarrollador crear una aplicación con contenido fácilmente mantenible y de un aspecto visual impactante.*

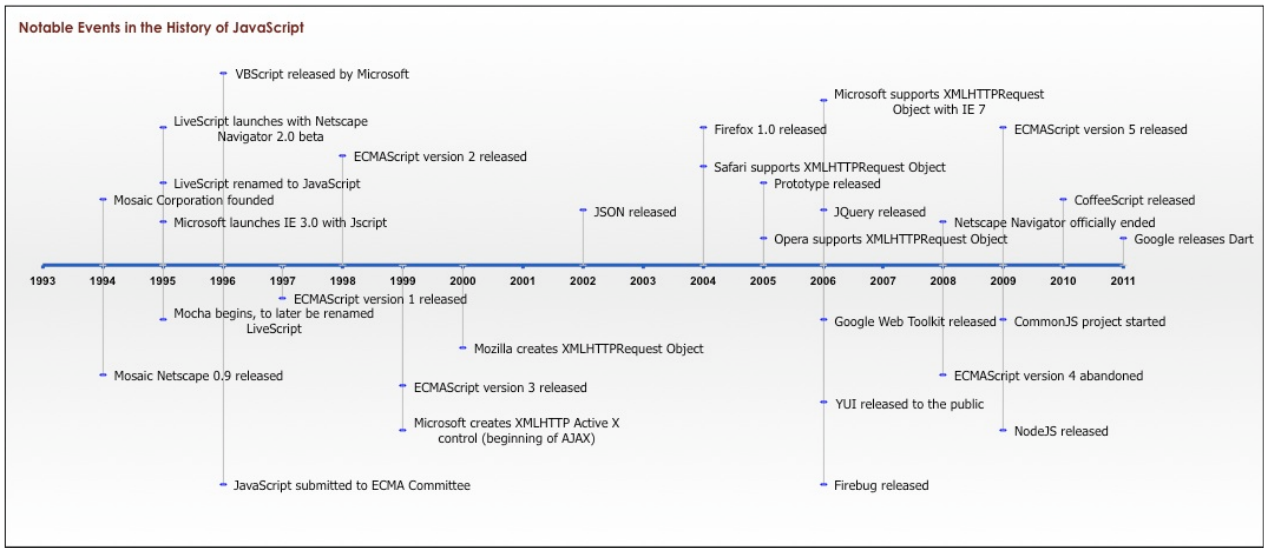
Entendiendo JavaScript

JavaScript es un lenguaje de programación dinámico orientado a objetos creado en 1995 por Brendan Eich. El uso del nombre *Java* fue una decisión comercial debido al auge que tenía dicho lenguaje en aquel entonces, pero no están relacionados más allá de algunas similitudes en la sintaxis.

Al momento de su creación existieron diferentes implementaciones del mismo, haciendo caótico su uso. Esto, sumado al nombre, que ocasionaba confusiones con respecto a su funcionamiento, y algunos errores de diseño, hizo que se volviera un lenguaje subestimado y mal usado por mucho tiempo.

Para 1997 la ECMA, una organización creada para desarrollar estándares de comunicación e información, realizó una especificación estándar llamada ECMAScript, la cual debe ser implementada por todos los navegadores. JavaScript en sí no es sinónimo de ECMAScript, si no una implementación de esta, al igual que ActionScript o JScript. ECMAScript actualmente está en su

versión 5.1, existiendo ya avances de la versión 6.



JavaScript Timeline. http://tom-barker.com/

JavaScript es un lenguaje que está influenciado por muchos otros lenguajes. Tiene similitudes con Java (y por lo tanto, algo de C/C++), y un poco de Self y Scheme, logrando hacer de él un lenguaje imperativo (se le dice al computador qué hacer y cómo, como C/C++ y Java), pero con conceptos de programación funcional (los programas son escritos en forma de funciones aritméticas, gracias a Self y Scheme). Además, tiene sus propias características:

- **Es un lenguaje de tipado dinámico**, esto quiere decir que una variable puede ser tanto un número como una cadena de caracteres o un objeto sin necesidad de una conversión especial.
- **Es orientada a objetos**, pero con la particularidad que no tiene clases. Las clases son reemplazadas por funciones y los prototipos permiten manejar herencia simple.
- **Las funciones también son objetos**, por lo que tienen atributos y métodos, además de poder ser asignados a variables y ser devueltos por otras funciones.
- **Permite evaluar sentencias en tiempo de ejecución**, así que se pueden crear y ejecutar sentencias (y funciones) a partir de datos ingresados en el programa en ejecución.

Si bien su propósito inicial fue el de ser un lenguaje de scripting para web, actualmente es usado en muchos otros entornos, desde realizar scripts para

Adobe Photoshop y manejar bases de datos como CouchDB hasta servir como interfaz para manejar hardware o levantar aplicaciones del lado del servidor con Node.js.

Sintaxis básica

Tipos

Números

JavaScript no tiene tipos de datos específicos para números enteros y flotantes (como `short`, `long`, `float` o `double`), solo tiene números, los cuales pueden tener o no punto flotante.

Las operaciones aritméticas básicas están soportadas, así como el operador módulo (`%`)

```
1 + 10;
// 11

0.5 + 12.2;
// 12.7

11 - 9;
// 2

19.8-0.5;
// 19.3

10 / 2;
// 5

310/15.5;
// 20
```

```
3 * 4;
// 12

21.2 * 7;
// 148.4

6 % 4;
// 2

12 % 7.5;
// 4.5
```

Así mismo, JavaScript tiene un objeto `Math`, el cual contiene operaciones matemáticas adicionales, así como constantes. Se debe tener en cuenta que las operaciones trigonométricas (`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`) trabajan con ángulos en radianes.

```
Math.E;
// 2.718281828459045

Math.PI;
// 3.141592653589793

Math.pow(6, 2); // potencia de 6 elevado a 2
//36

Math.tan(45 * (Math.PI / 180)); // Convertimos 45 grados a radianes
// 0.9999999999999999
```

En el último ejemplo se puede ver que el resultado da `0.9999999999999999` cuando debería ser `1`. Esto sucede porque internamente el lenguaje guarda los números en formato binario con un número limitado de dígitos.

En formato binario, una fracción tiene su representación finita solo si el denominador tiene al 2 como único factor primo. Es por esto que, por ejemplo, expresar 1/10 (0.1) en base 2 ($0.0001100110011001100110011..._2$) no tiene una representación finita, ya que el denominador (10) está compuesto por dos factores, siendo ambos números primos: 2 y 5. Una explicación más detallada puede ser encontrada en What Every JavaScript Developer Should Know About Floating Points.

Cadenas

Las cadenas en JavaScript son valores que pueden ser escritos tanto con comillas simples como dobles:

```
"Bienvenidos a La Buena Espina".length;
// 29

// o...

'Bienvenidos a La Buena Espina'.length;
// 29
```

Las cadenas también son objetos, por lo que tienen propiedades (como `length`) y métodos:

```
'Bienvenidos a La Buena Espina'.toUpperCase();
// "BIENVENIDOS A LA BUENA ESPINA"

'La Buena Espina'.split('');
// ["L", "a", " ", "B", "u", "e", "n", "a", " ", "E", "s", "p",
"i", "n", "a"]
```

La propiedad `length` en una cadena es de solo lectura.

Booleanos

El tipo de dato lógico o booleano solo puede tener dos valores: `true` (verdadero) y `false` (falso). JavaScript tiene la particularidad de convertir implícitamente valores que son interpretados como verdaderos o falsos: `false`, `0`, `""` (o `''`), `NaN` (*Not A Number*), `null` y `undefined` son interpretados como `false` en una condicional, mientras que el resto de valores posibles son interpretados como `true`.

Variables

Las variables en JavaScript permiten guardar objetos para su posterior uso. Al ser JavaScript un lenguaje de tipado dinámico, una misma variable puede guardar diferentes tipos u objetos a lo largo de su ciclo de vida. Una variable es declarada utilizando la palabra reservada `var` y puede o no tener un valor inicial:

```
var siteTitle = 'Bienvenidos a La Buena Espina';
var currentUser;

siteTitle;
// "Bienvenidos a La Buena Espina"
currentUser;
// undefined
```

Arrays

Los arreglos (*arrays*) son un tipo especial de objetos y representan colecciones que pueden guardar cualquier tipo de dato en JavaScript y sus elementos pueden o no ser del mismo tipo.

Los arreglos tienen una propiedad llamada `length` y utilizan los corchetes (`[]`) para acceder a un elemento del arreglo a través de su índice. En JavaScript,

el índice de los arreglos empieza en 0 y el valor de `length` es igual al último índice del arreglo más uno.

Al ser objetos, los arreglos tienen una serie de métodos que sirven para manipularlos:

join

Concatena los elementos de un arreglo en una cadena usando el parámetro que recibe este método como separador.

```
var dateParts = [19, 8, 1990];
dateParts.length;
// 3

dateParts.join('/');
// "19/8/1990"

var menuCategories = ['Entradas', 'Segundos', 'Postres'];
menuCategories.join(', ');
// "Entradas, Segundos, Postres"
```

pop

Quita el último elemento del arreglo y retorna su valor.

push

Agrega un elemento al final del arreglo y retorna el nuevo tamaño.

```
var dateParts = [19, 8, 1990];
dateParts.pop();
// 1990

dateParts;
```

```
// [19, 8]
dateParts.push(1989);
// 3

dateParts;
// [19, 8, 1989]
```

`indexOf`

Busca en el arreglo el primer elemento que sea igual al parámetro que se le pasa y devuelve su índice.

```
var dateParts = [19, 8, 1990];
dateParts.indexOf(8);
// 1
```

`indexOf` está definido en Internet Explorer 9 y superiores, y en Firefox 1.5 y superiores.

`reverse`

Modifica el arreglo invirtiendo sus elementos y retorna el arreglo invertido.

```
var dateParts = [19, 8, 1990];
dateParts.reverse();
// [1990, 8, 19]

dateParts;
// [1990, 8, 19]
```

`concat`

Agrega elementos a una copia del arreglo original y devuelve la copia con los nuevos elementos agregados.

```
var dateParts = [1990, 8, 19];
dateParts.concat(9, 30, 0);
// [1990, 8, 19, 9, 30, 0]

dateParts;
// [1990, 8, 19]
```

slice

`slice` crea una copia del arreglo de acuerdo a los parámetros que esta función recibe: el primer parámetro es el índice del elemento en el que inicia la copia y el segundo argumento es el índice siguiente al elemento final de la copia. El segundo parámetro es opcional, y si se obvia, la copia se realizará desde el índice inicial hasta el final del arreglo original.

Esta función devuelve la copia del arreglo y deja el arreglo original intacto.

```
var array = [8, 20, 12, 9, 1];
array.slice(2, 4);
//[12, 9]

array;
// [8, 20, 12, 9, 1]
```

splice

`splice` modifica el arreglo original, de acuerdo a los parámetros que recibe. El primer parámetro es el índice del elemento donde se empezará a cortar, mientras que el segundo parámetro es el número de elementos que se cortarán del arreglo original, incluyendo el elemento inicial. Puede recibir uno o más parámetros opcionales, los cuales se insertan en la posición definida en el primer parámetro.

Esta función devuelve el nuevo arreglo cortado y modifica el arreglo original.

```
var array = [8, 20, 12, 9, 1];
array.splice(2, 1);
// [12]

array;
// [8, 20, 9, 1]

array.splice(1, 0, 15);
// []

array;
// [8, 15, 20, 9, 1]
```

Objetos

Los objetos literales son colecciones de pares nombre-valor donde la primera parte (el *nombre*) es único dentro del objeto y por lo general es una cadena, mientras que la segunda parte (el *valor*) puede ser de cualquier tipo, incluyendo otros objetos.

Se puede crear un objeto de tres formas:

A. De forma literal:

```
var object = {};

object;
// {}
```

B. Creando una instancia de `Object`:

```
var object = new Object({});
```

```
object;
// {}
```

La tercera forma utiliza `Object.create`, por lo que para entenderla tenemos que ver un poco más a detalle cómo se comportan las propiedades de un objeto literal.

La forma B crea un objeto del mismo constructor del parámetro que se le pase:

```
new Object({});
// Object {}
new Object(1);
// Number {}
new Object("");
// String {}
new Object(true);
// Boolean {}
new Object([]);
// []
```

Existen dos formas para asignar y acceder al valor de una propiedad en un objeto:

```
var dish = {
  name: 'Ceviche simple',
  ingredients: [
    '1 kilo de pescado',
    '2 cebollas',
    '1 taza de jugo de limón',
    '1 ají limo',
    'sal'
  ],
  garnishes: [
    'lechuga (2 hojas por plato)',
```

```
      'maíz cancha',
      '4 porciones de yuca',
      '4 choclos sancochados',
      'camote sancochado en rodajas (2 por plato)'
    ],
    diners: 4
  };
```

A. Con punto:

```
dish.name;
// "Ceviche simple"

dish.name = 'Ceviche simple (estilo trujillano)';
// "Ceviche simple (estilo trujillano)"
```

B. Con corchetes:

```
dish['name'];
// "Ceviche simple"

dish['name'] = 'Ceviche simple (estilo trujillano)';
// "Ceviche simple (estilo trujillano)"
```

La forma B tiene una ventaja importante con respecto a la forma A, debido a que se accede a la propiedad con el nombre de esta propiedad en forma de cadena, permitiendo utilizar una variable cuyo valor sea definido dinámicamente:

```
var dinersPropertyName = 'name';

dish[dinersPropertyName];
// "Ceviche simple"
```

```
dish[dinersPropertyName] = 'Ceviche simple (estilo trujillano)';
// "Ceviche simple (estilo trujillano)"
```

Cada propiedad definida en un objeto tiene por defecto las siguientes características:

- Es enumerable: Saldrá listada si se recorre el objeto en un estructura repetitiva o utilizando el método `Object.keys`.
- Es configurable: Se podrá eliminar dicha propiedad. Así mismo, podrán cambiarse el resto de configuraciones (incluidas las mencionadas en esta lista) de la propiedad utilizando `Object.defineProperty`.
- Es grabable: Se podrá cambiar el valor de la propiedad.

El constructor `Object` tiene una serie de métodos que permiten crear y manipular objetos de forma más avanzada:

`Object.create`

Esta es una tercera forma de crear un objeto, permitiendo además definir sus propiedades y cuál será su *prototype*. El *prototype* de un objeto es otro objeto, el cual guardará las propiedades y métodos que compartirá con el objeto a crear.

```
var dish = Object.create(Object.prototype, {
  name: {
    value: '',
    writable: true,
    configurable: false
  },
  ingredients: {
    value: [],
    writable: true,
    configurable: false
  },
  garnishes: {
    value: [],
    writable: true,
```

```
    configurable: true
  },
  diners: {
    value: 1,
    writable: true,
    configurable: false
  }
});

dish.name = 'Ceviche simple';
dish.name;
// "Ceviche simple"

dish.ingredients.push('1 kilo de pescado');
// 1
```

Este método toma dos parámetros. El primero es el *prototype* del objeto, mientras que el segundo es un objeto plano que enumera las propiedades que tendrá el nuevo objeto. Este segundo parámetro tiene la misma forma que el segundo parámetro de `Object.defineProperties`.

`Object.create` es útil para definir prototipos, sobre todo si tienen propiedades especiales (por ejemplo, dinámicos o de solo lectura).

`Object.defineProperty`

Crea o modifica una propiedad en un objeto. A diferencia de la manipulación de propiedades mediante asignación con punto o corchetes, este método permite tener un control más avanzado de cómo se podrá manipular la propiedad.

Cuando se define una propiedad con `Object.defineProperty`, por defecto no es ni enumerable ni configurable ni grabable(excepto en Chrome, donde por defecto sí es grabable).

```
var siteTitle = {};
```

```
Object.defineProperty(siteTitle, 'internalValue', {
  writable: true,      // internalValue podrá cambiar de valor
  configurable: false, // internalValue podrá cambiar de valor pero
no ser eliminado del objeto
  enumerable: true     // internalValue aparecerá en Object.keys o
usando for..in
});
```

Este método recibe 3 parámetros: el objeto a modificar, la propiedad a definir y el descriptor de la propiedad. Un descriptor de propiedad puede ser de dos tipos:

- Descriptor de datos: para propiedades que tienen un valor
  - `value`: El valor asignado por defecto a la propiedad. Este valor puede ser de cualquier tipo.
  - `writable`: Valor booleano que define si la propiedad es grabable o no.
- Descriptor de acceso: para definir métodos de acceso (`get` y `set`).
  - `get`: Si está definida, esta función se ejecutará al intentar acceder a la propiedad.
  - `set`: Si está definida, esta función se ejecutará al intentar asignar un valor a la propiedad.

Ambos tipos de descriptores comparten dos atributos con valores booleanos: `configurable` y `enumerable`.

`Object.defineProperties`

Crea o modifica propiedades para un objeto. En este caso el segundo parámetro es un objeto plano donde cada par nombre : valor corresponde al nombre de la propiedad y el descriptor de la misma.

```
var siteTitle = {};

Object.defineProperties(siteTitle, {
  internalValue: {
    writable: true,       // internalValue podrá cambiar de valor
```

```
    configurable: true,   // internalValue podrá cambiar de valor y
ser eliminado del objeto
    enumerable: true      // internalValue aparecerá en Object.keys
o usando for..in
  },
  toTitle: {
    writable: false,      // el método toTitle no podrá cambiar de
valor
    configurable: false,  // el método toTitle no podrá ser
eliminado ni cambiado de valor,
    enumerable: false,    // el método toTitle no aparecerá en
Object.keys o usando for..in
    value: function() {
      return this.internalValue.toUpperCase().split('').join(' ');
    }
  }
});
```

Object.preventExtensions

Bloquea la capacidad del objeto de tener nuevas propiedades.

```
var object = {};

Object.preventExtensions(object);
```

Object.freeze

Bloquea futuras modificaciones en un objeto.

```
var object = {};

Object.freeze(object);
```

Object.seal

Sella un objeto, negando la capacidad del objeto de tener nuevas propiedades y de tener propiedades configurables.

```
var object = {};

Object.seal(object);
```

Object.getOwnPropertyNames

Lista todas las propiedades (incluyendo métodos) de un objeto, sean estos enumerables o no.

```
var siteTitle = Object.create(String.prototype, {
  internalValue: {
    enumerable: false,
    writable: true,
    configurable: false
  }
});

Object.getOwnPropertyNames(superString);
// ["internalValue"]
```

Object.getPrototypeOf

Devuelve el *prototype* de un objeto.

```
var siteTitle = Object.create(String.prototype, {
  internalValue: {
    enumerable: false,
    writable: true,
```

```
    configurable: false
  }
});

Object.getPrototypeOf(siteTitle);
// String {}

siteTitle.internalValue = 'La Buena Espina';
// "La Buena Espina"
Object.getPrototypeOf(siteTitle.internalValue);
// TypeError: Object.getPrototypeOf called on non-object
```

Como se ve en el segundo ejemplo, `Object.getPrototypeOf` solo funciona para objetos mas no para valores primitivos.

`Object.isExtensible`

Verifica si el objeto permite agregar nuevas propiedades.

```
var object = {};

Object.preventExtensions(object);
Object.isExtensible(object);
// false
```

Este comportamiento es definitivo. Es decir, una vez que un objeto deja de ser extensible, no puede volver a su estado anterior.

`Object.isFrozen`

Verifica si un objeto está *congelado* (Ver `Object.freeze`)

```
var object = {};
```

```
Object.freeze(object);
Object.isFrozen(object);
// true
```

Object.isSealed

Verifica si el objeto está sellado. Un objeto está sellado si no es extensible (`Object.isExtensible` devolviendo `false`) y si ninguna de sus propiedades son configurables.

```
var object = {};

Object.seal(object);
Object.isSealed(object);
// true
```

Object.keys

Lista todas las propiedades de un objeto que sean enumerables.

```
var siteTitle = Object.create(String.prototype, {
  internalValue: {
    enumerable: false,
    writable: true,
    configurable: true
  }
});

Object.keys(siteTitle);
// []

Object.defineProperty(siteTitle, 'internalValue', {
  enumerable: true
});
```

```
Object.keys(superString);
// ["internalValue"]
```

En general, estos métodos no están disponibles para versiones anteriores a Internet Explorer 9 o las primeras versiones de Chrome y Firefox. En el caso de Opera, estos métodos están disponibles a partir de la versión 12.

Fechas

A diferencia de los arreglos y los objetos, que pueden crearse de forma literal, las fechas en JavaScript son instanciadas utilizando el constructor `Date`. Este constructor tiene diferentes modos, ya que puede tomar como argumentos una cadena, un número que represente una marca de tiempo en milisegundos, o valores separados por comas empezando por el año, mes y día, hasta llegar al milisegundo.

```
new Date();
// Mon Feb 03 2014 21:22:52 GMT-0500 (PET)
new Date('2014-02-04T02:23:16.198Z');
// Mon Feb 03 2014 21:23:16 GMT-0500 (PET)
new Date(1391480663373);
// Mon Feb 03 2014 21:24:23 GMT-0500 (PET)
new Date(2014, 2, 3, 21, 25, 12, 0);
// Mon Mar 03 2014 21:25:12 GMT-0500 (PET)
```

En el último ejemplo se puede ver que aún cuando el segundo parámetro, que corresponde al mes, tiene como valor `2`, genera una fecha con el mes de Marzo. Esto es porque en JavaScript, los valores numéricos de los meses empiezan en 0. Algo similar pasa con el parámetro que representa a los años, pues los valores del 0 al 99 son equivalentes respectivamente a los años del 1900 a 1999.

Por defecto, si ningún parámetro es pasado al constructor, `Date` devuelve la fecha y hora actual de acuerdo a la zona horaria definida en el sistema.

Así mismo, el constructor `Date` tiene 3 métodos:

Date.now

Devuelve el valor correspondiente al número de milisegundos pasados desde el primero de Enero de 1970 a las 00:00:00 hasta la fecha actual en la zona horaria definida en el sistema.

```
var now = Date.now();
now;
// 1391576500965

new Date(now);
// Wed Feb 05 2014 00:01:40 GMT-0500 (PET)
```

Date.parse

Analiza una fecha en forma de cadena y devuelve el valor correspondiente al número de milisegundos pasados desde el primero de Enero de 1970 a las 00:00:00 en tiempo local. La fecha debe tener formato RFC2822, o ISO 8601 (disponible a partir de versiones superiores a Internet Explorer 9, Firefox 3, Safari 3.2 y Opera 10).

```
var date = Date.parse('Aug 19, 1990 09:30:00');
date;
// 651076200000

new Date(date);
// Sun Aug 19 1990 09:30:00 GMT-0500 (PET)
```

Date.UTC

Similar a la última forma de usar el constructor de `Date`, devuelve el valor correspondiente al número de milisegundos pasados desde el primero de Enero de 1970 a las 00:00:00 hasta la fecha actual en UTC.

```
var date = Date.UTC(2014, 2, 3, 21, 25, 12, 0);
date;
// 1393881912000
new Date(date);
// Mon Mar 03 2014 16:25:12 GMT-0500 (PET)
```

Se debe tener en cuenta que los valores numéricos devueltos por `Date.now`, `Date.parse` y `Date.UTC` no contienen información referente a la zona horaria, por lo que si se utilizan estos números en otros métodos de `Date` se tomará en cuenta la zona horaria local, a menos que la referencia indique lo contrario.

Las instancias de `Date` tienen métodos que permiten obtener información detallada de sus atributos:

getFullYear

Devuelve el año de una fecha (valor de 4 dígitos).
`setFullYear`: Asigna el año a una fecha.

Método equivalente en UTC: `getUTCFullYear` / `setUTCFullYear`.

getMonth

Devuelve el mes de una fecha (valor entre 0 y 11).
`setMonth`: signa el mes a una fecha.

Método equivalente en UTC: `getUTCMonth` / `setUTCMonth`.

getDate

Devuelve el día del mes (valor entre el 1 al 31) de una fecha.
`setDate`: Asigna el día del mes a una fecha.

Método equivalente en UTC: `getUTCDate` / `setUTCDate`.

getDay

Devuelve el día de semana (valor del 0 al 6) de una fecha. El número `0` en este caso representa al día domingo, el `1` al lunes y así sucesivamente.

Método equivalente en UTC: `getUTCDay`.

`getHours`

Devuelve la hora en formato de 24 horas (valor entre el 0 al 23) de una fecha.
`setHours`: Asigna las horas en formato de 24 horas a una fecha.

Método equivalente en UTC: `getUTCHours` / `setUTCHours`.

`getMinutes`

Devuelve los minutos (valor entre el 0 al 59) de una fecha.
`setMinutes`: Asigna los minutos a una fecha.

Método equivalente en UTC: `getUTCMinutes` / `setUTCMinutes`.

`getSeconds`

Devuelve los segundos (valor entre el 0 al 59) de una fecha.
`setSeconds`: Asigna los segundos a una fecha.

Método equivalente en UTC: `getUTCSeconds` / `setUTCSeconds`.

`getMilliseconds`

Devuelve los milisegundos (valor entre el 0 al 999) de una fecha.
`setMilliseconds`: Asigna los milisegundos a una fecha.

Método equivalente en UTC: `getUTCMilliseconds` / `setUTCMilliseconds`.

`getTime`

Devuelve el número de milisegundos transcuridos desde el primero de Enero de 1970 a las 00:00:00 UTC hasta una fecha determinada.
`setTime`: Reemplaza una fecha por la fecha correspondiente a un número de milisegundos transcuridos desde el primero de Enero de 1970 a las 00:00:00 UTC.

`getTimezoneOffset`

Devuelve la diferencia en minutos entre la zona horaria del sistema y el UTC. Si la zona horaria es negativa (por ejemplo, UTC-05:00 para el caso de Perú), el valor devuelto por `getTimezoneOffset` será positivo (es decir, 5 * 60).

Así mismo, tienen métodos que permiten convertir las fechas a cadenas:

`toDateString`

Devuelve la porción de fecha (ignorando hora, minutos, segundos, milisegundos y zona horaria).

`toTimeString`

Devuelve la porción de hora (hora, minutos, segundos, milisegundos y zona horaria).

`toISOString`

Devuelve la fecha completa en formato ISO 8601.

`toJSON`

Similar a `toISOString`, devuelve la fecha completa en formato ISO 8601.

`toUTCString`

Similar a `toDateString`, devuelve la fecha completa en UTC.

`toLocaleDateString`

Devuelve la porción de fecha (ignorando hora, minutos, segundos, milisegundos y zona horaria) en un formato local.

`toLocaleTimeString`

Devuelve la porción de hora (hora, minutos, segundos, milisegundos y zona horaria) en un formato local.

`toLocaleString`

Devuelve la fecha completa (fecha y hora) en un formato local.

Funciones

Las funciones en JavaScript permiten crear operaciones reutilizables.

El número de parámetros pasados a una función no es estricta. Si una función está definida para aceptar 3 argumentos y se le pasan 2 parámetros, el tercer parámetro será asignado a `undefined`, mientras que si a la misma función se le pasan 4 parámetros, el cuarto parámetro será ignorado. Es decir, si se tiene una función `sum`:

```
function sum(a, b, c) {
  var result = a + b;

  if (c) {
    result += c;  // similar a: result = result + c;
  }

  return result;
};

sum(1, 2);
// 3
sum(1, 2, 3, 4);
// 6
```

Tipos vs Objetos

En JavaScript, como en muchos otros lenguajes, existen los llamados **tipos**, los cuales corresponden a los valores más básicos que tiene un lenguaje. Los tipos en JavaScript son:

- Undefined
- Null
- Boolean
- String
- Number
- Object

Podemos identificar el tipo de un valor mediante el operador `typeof`:

```
typeof undefined
// "undefined"
typeof null
// "object"
typeof true
// "boolean"
typeof false
// "boolean"
typeof "hola"
// "string"
typeof 10
// "number"
typeof 1.6
// "number"
typeof {}
// "object"
```

Debido a un error de diseño, el tipo de `null` es `object`. Este error está explicado a profundidad en The history of “typeof null”.

Los objetos, por otro lado, son valores que pueden ser creados de dos formas: utilizando constructores propios del lenguaje o utilizando nuevos constructores creados por el desarrollador o por terceros. Los constructores propios del lenguaje son:

- Boolean
- String

- Number
- Object
- Array
- Date
- RegExp
- Function
- Error

Si bien no existe un operador que devuelva el constructor del objeto, se puede comparar el constructor del objeto con otros constructores y saber si el objeto es una **instancia** del mismo:

```
var object = {};
var array = [];
var date = new Date();
var regexp = /(.*)/;

function func() {};

var string = "";
var number = 10;
var bool = true;

object instanceof Object;
// true
array instanceof Array;
// true
date instanceof Date;
// true
regexp instanceof RegExp;
// true
func instanceof Function;
// true

string instanceof String;
```

```
// false
number instanceof Number;
// false
bool instanceof Boolean;
// false
```

Valor primitivos

Como se puede ver en los 3 últimos ejemplos, `instanceof` devuelve false aún cuando sabemos que `string` es una cadena, `number` es un número y `bool` contiene un valor lógico. Esto sucede porque `instanceof` no trabaja bien con los denominados valores primitivos. Cada valor primitivo tiene un constructor asociado:

Valor primitivo	Constructor
string	String
number	Number
boolean	Boolean

Los valores primitivos pueden ser confundidos con objetos debido a que tienen acceso a propiedades y métodos (como `"hola".length`). Internamente, el intérprete crea una instancia del constructor asociado al valor primitivo y le da el valor de este, para luego acceder a la propiedad o método que se requiera.

Estructuras condicionales

Las estructuras condicionales en JavaScript son muy similares a las de otros lenguajes parecidos a C/C++. Es de este lenguaje que toma prestada la sintaxis de llaves, usada para delimitar los bloques en JavaScript.

`if..else`

`if` ejecutará las sentencias ubicadas en el primer bloque si la condición de `if` es `true`. Si se definen la sentencias `else` o `else if`, se ejecutarán sus respectivos bloques de sentencias en caso cumplan su condición lógica.

```
var randomNumber = 10;

if (randomNumber < 10) {
  'Menor a 10';
}
else {
  'Igual o mayor a 10';
}

// "Igual o mayor a 10"

if (randomNumber < 10) {
  'Menor a 10';
}
else if (randomNumber == 10) {
  'Igual a 10';
}
else {
  'Mayor a 10';
}

// "igual a 10"
```

switch

`switch` utiliza el operador `===` internamente para comparar el valor de `switch` con todos los casos definidos en él. Si el caso `default` está definido y ninguna de las comparaciones con los otros casos da `true`, se ejecutará el bloque definido para `default`.

```
var obj = '20';

switch(obj) {
  case '20':
```

```
    'obj es una cadena';
    break;
  case 20:
    'obj es un número';
    break;
  default:
    'obj tiene otro tipo de dato';
}

// "obj es una cadena"
```

En las estructuras condicionales se utilizan los operadores de comparación, como `==`, `!=`, `===` y `!==`. La diferencia entre `==` y `===` es que el primero solo compara valores, mientras que el segundo compara valores y tipos de datos. De igual forma pasa con `!=` y `!==`.

Si solo se evalua un valor en un operador condicional, este valor se convertirá implícitamente en valores booleanos `true` o `false`:

```
if (0) {
  '0 se convierte a true';
}
else {
  '0 se convierte a false';
}
// "0 se convierte a false"

var obj = {};

if (obj) {
  'obj se convierte a true';
}
else {
  'obj se convierte a false';
}
```

Estructuras repetitivas

Las estructuras repetitivas en JavaScript permiten recorrer arreglos u objetos, así como ejecutar operaciones mientras se cumpla una condición.

`for`

La sentencia `for` es similar a la usada en C o Java. Tiene 3 partes: una expresión inicial que define el inicio del bucle o repetición, la condición que debe darse para que el bloque de la sentencia se ejecute, y una expresión que incremente el valor utilizado en la condición de la segunda parte.

```
var counter;

for (counter = 0; counter < 5; counter++) { // inicio del bucle;
condición; expresión incremental
  console.log(counter);
}

// 0
// 1
// 2
// 3
// 4
```

`for..in`

Permite recorrer objetos a través de sus propiedades enumerables y es similar a `for`, excepto que en este caso tiene dos expresiones, separadas por la palabra reservada `in`. La primera expresión es una variable auxiliar que tendrá asignado el nombre de la propiedad que está leyéndose en cada iteración, mientras que la segunda expresión es el objeto que se va a recorrer.

```
var obj = {
```

```
  string: 'hola',
  number: 24,
  date: new Date()
};

var prop;

for (prop in obj) {
  console.log(prop + ' : ' +  obj[prop]);
}

// string : "hola"
// number : 24
// date : Wed Feb 05 2014 00:01:40 GMT-0500 (PET)
```

Si el objeto evaluado en una sentencia `for..in` tiene propiedades heredadas de otro objeto, estas también se iterarán.

`while`

`while` ejecuta las sentencias de su bloque mientras la condición pasada sea verdadera.

```
var counter = 0;

while (counter < 5) {
  console.log(counter);
  counter++;
}

// 0
// 1
// 2
// 3
// 4
```

`do..while`

`do..while` ejecuta las sentencias de su bloque hasta que la condición pasada sea falsa. A diferencia de `while`, `do..while` ejecuta al menos una las sentencias de su bloque.

```
var counter = 0;

do {
  console.log(counter);
} while (counter != 0);

// 0
```

En este ejemplo, se ve que `i != 0` dará `false`. Sin embargo, se ejecuta la sentencia `console.log(i)` una vez con el valor inicial.

Todas las estructuras repetitivas aceptan una sentencia llamada `break`, la cual interrumpe las iteraciones de la estructura. Así mismo, existe la sentencia `continue`, que salta a la siguiente iteración.

2. Capítulo 2: Funciones →

Acerca de | Disponible en Amazon.com