ebimap
Edicom Business Integrator Mapping Tool

Edicom Business Integrator Mapping tool
Anexo - Sintaxis JavaScript

edicom
connecting business

Contenido

MANUAL JAVASCRIPT

1.1 VARIABLES

Las variables en JavaScript no tienen tipo asignado y cualquier valor puede almacenarse en cualquier variable. Las variables pueden ser declaradas con una sentencia `var`. Estas variables son visibles una vez han sido declaradas, pueden ser accedidas dentro de la función en la que se declararon. Las variables declaradas fuera de cualquier función y variables usadas por primera vez dentro de una función sin ser declaradas con la sentencia `var` son variables globales. Este es un ejemplo de declaración de variables y variables globales:

```
x = 0; // Una variable global
var y = 'Hola!'; // Otra variable global

function f(){
  var z = 'texto'; // Una variable local
  veinte = 20; // Variable global por que no se ha usado var
  return x; // Se puede usar x aquí porque es global
}
// El valor de la variable z ya no esta disponible
```

1.2 TIPOS DE DATOS BÁSICOS

NÚMEROS (CLASE NUMBER)

Los números en JavaScript se representan en binario como doubles en formato IEEE-754, con una precisión de 15 dígitos significativos. Como son números binarios no siempre representan exactamente números decimales, especialmente fracciones.

Esto es un problema a la hora de formatear números, puesto que JavaScript no tiene funciones nativas para formatear números. Por ejemplo:

```
alert(0.94 - 0.01); // muestra 0.9299999999999999
```

Como resultado, se debe usar el redondeo cuando se formatean los números para la su escritura. El método toFixed() no es parte de la especificación ECMAScript y varia según la implementación por lo que no debe ser usado..

Los números se pueden especificar usando cualquiera de estas notaciones:

```
345;    // Un numero entero
34.5;   // Numero en coma flotante
3.45e2; // Numero en coma flotante equivalente a 345
0377;   // un entero octal igual a 255
0xFF;   // un entero hexadecimal igual a 255
```

El constructor **`Number`** se puede usar para realizar conversiones numéricas explícitamente

```
var miCadena = "123.456";
var miNumero = Number( miCadena );
```

Cuando se usa como constructor, se crea un objeto de tipo *Number* con sus métodos asociados, (aunque no se suele usar):

```
miObjetoNumber = new Number( 123.456 );
```

VECTORES (CLASE ARRAY)

Un objeto *Array* es un mapa de enteros a valores. En JavaScript, todos los objetos pueden mapear de enteros a valores, pero los objetos de tipo *Array* tienen métodos especializados en el manejo de índices enteros (join, slice, y push...).

Los objetos de tipo *Array* tiene la propiedad `length` que se garantiza que tendrá un valor mayor que el mayor índice usado en el vector. Se actualiza automáticamente si se crea una propiedad con un índice mayor que el índice máximo. Si escribimos un valor más pequeño en la propiedad length eliminamos los índices mayores. La propiedad `length` es la única característica especial que distingue los objetos del tipo *Array* de los del tipo Object.

Se puede acceder a los elementos de un vector usando el método matricial para el acceso a las propiedades de los objetos:

```
miVector[1];
```

No es posible usar la notación con punto o cadenas como representación del entero:

```
miVector.1;     // Error sintáctico
miVector["01"]; // not es lo mismo que myArray[1]
```

Para la declaración de un vector se puede usar un literal o el constructor `Array`:

```
myArray = [0,1,,,4,5];          // vector con longitud 6 y 6
elementos,
                          //incluyendo 2 elementos no definidos
myArray = new Array(0,1,2,3,4,5);      // vector con longitud 6 y 6
elementos
myArray = new Array(365);              // vector vacío con longitud
365
```

Los objetos de tipo *Array* están implementados para que solo los objetos definidos consuman memoria. Estableciendo `myArray[10] = 'algo'` y `myArray[57] = 'algoMas'` solo se usa espacio para dos elementos, como en el resto de los objetos. Pero la propiedad `length` del objeto *Array* seguirá valiendo 58.

Se pueden usar los literales de declaración de objetos para crear matrices asociativos y multidimensionales o ambas cosas.

```
dog = {"color":"brown", "size":"large"};
dog["color"]; // devuelve "brown"
cats = [{"color":"brown", "size":"large"},
        {"color":"black", "size":"small"}];
cats[0]["size"]; // devuelve "large"

dogs = {"rover":{"color":"brown", "size":"large"},
        "spot":{"color":"black", "size":"small"}};
dogs["spot"]["size"]; // devuelve "small"
```

CADENAS (STRING)

Las cadenas en JavaScript son una secuencia de caracteres y puede crearse directamente insertando la serie de caracteres entre comillas dobles o simples.

```
var s = "Hola mundo!";
var t = 'Hola a todo el mundo.';
```

Se puede acceder a caracteres individuales usando la misma notación que en los vectores:

```
var h = s[0]; // Ahora h contiene 'H'
```

aunque si se quiere crear código compatible con ECMAScript es recomendable acceder al valor usando el método `charAt` de los objetos de tipo *String*:

```
var h = s.charAt(0); // Ahora h contiene 'H' - ECMAScript
compatible
```

Sin embargo en JavaScript las cadenas son inmutables (no se pueden modificar después de la creación):

```
s[0] = "H"; // ERROR
```

Aplicado el operador de igualdad ("= =") a dos cadenas devuelve true si las cadenas tienen el mismo contenido:

```
var x = "mundo";
var comparacion1 = ("Hola, " + x == "Hola, mundo"); // Ahora
comparacion1 contiene true
```

```
var comparacion2 = ("Hola, " + x == "hola, mundo"); // Ahora
comparacion1 contiene false
```

OBJETOS

Los objetos más básicos en JavaScript actúan como diccionarios. Estos diccionarios pueden tener cualquier tipo de valor asociado a una clave, que es una cadena. Los objetos con valor se pueden crear usando la notación literal:

```
var o = {nombre: 'Mi Objeto', propiedad: 42};
```

Las propiedades de los objetos puedes ser creadas, escritas o leídas usan la notación *Objeto.propiedad* o con la sintaxis de las vectores:

```
var nombre = o.name; // la variable nombre contiene 'Mi Objeto'
var propiedad = o['propiedad'];
// la variable propiedad contiene 42
```

Los objetos y vectores literales pueden crear estructuras de datos flexibles:

```
var miEstructura = {
  nombrecompleto: {
    nombre: "Juan",
    apellido: "Garcia"
  },
  edad: 65,
  aficiones: [ "tenis", "futbol" ]
};
```

1.3 OPERADORES

El operador '+' esta sobrecargado y puede ser usado para la concatenación de cadenas y la suma aritmética y convertir cadenas a números. También tiene un uso especial en las expresiones regulares.

```
// Concatenar dos cadenas
var a = 'Esto';
var b = ' y eso';
alert(a + b);  // muestra 'Esto y eso'

// Sumar dos números
var x = 2;
var y = 6;
```

```
alert(x + y); // muestra 8

// Sumando una cadena y un numero actúa como una concatenación
alert( x + '2'); // muestra 22

// Convierte una cadena a un numero
var z = '4';   // z es una cadena (el dígito 4)
alert( z + x); // muestra 42
alert( +z + x);// muestra 6
```

OPERADORES ARITMÉTICOS

Operadores binarios

```
+     Suma
-     Resta
*     Multiplicación
/     División (devuelve un valor en coma flotante)
%     Resto (devuelve el resto entero)
```

Operadores unários

```
-     Negación unaria (Cambia el signo)
++    Incremento (puede ser prefijo o postfijo)
--    Decremento (puede ser prefijo o postfijo)
```

ASIGNACIÓN

```
=     Asignación
+=    Suma y asignación
-=    Resta y asignación
*=    Multiplicación y asignación
/=    División y asignación
%=    Resto y asignación
var x = 1;
x *= 3;
alert( x );  // muestra: 3
x /= 3;
alert( x );  // muestra: 1
x -= 1;
alert( x );  // muestra: 0
```

OPERADORES DE COMPARACIÓN

```
==    Igual
!=    Distinto
>     Mayor que
>=    Mayor o igual que
<     Menor que
<=    Menor o igual que
===   Idéntico (igual y del mismo tipo)
!==   No idéntico
```

OPERADORES BOOLEANOS

JavaScript tiene tres operadores lógicos booleanos: *&&* (AND lógico), *||* (OR lógico), y *!* (NOT lógico).

En el contexto de un operador booleano, todos los valores se evalúan a cierto, excepto el valor booleano *false*, el numero 0, una cadena de longitud 0, o cualquiera de estos valores especiales *null*, *undefined*, o *NaN*. La función puede usarse para realizar esta conversión de manera explícita:

```
Boolean( false );     // devuelve false
Boolean( 0 );         // devuelve false
Boolean( 0.0 );       // devuelve false
Boolean( "" );        // devuelve false
Boolean( null );      // devuelve false
Boolean( undefined ); // devuelve false
Boolean( NaN );       // devuelve false
// SOLO las cadenas vacías devuelven false
Boolean("false");     // devuelve true
Boolean("0");         // devuelve true
```

El operador NOT unário ! , primero evalúa su operador en un contexto boolean y después devuelve el valor boolean opuesto:

```
var a = 0;
var b = 9;
!a; // se evalúa a true,  al igual que (Boolean( a ) == false)
!b; // se evalúa a false, al igual que (Boolean( b ) == true)
```

El doble uso del operador ! se puede usar para normalizar un valor booleano:

```
var arg = null;
arg = !!arg; // arg tiene el valor false, en vez de null

arg = "finished"; // cadena no vacía
```

```
arg = !!arg; // arg tiene el valor true
```

En las implementaciones iniciales los operadores *&&* y *||* se comportaban de forma similar a C y siempre devolvían un valor booleano:

```
x && y; // devuelve true si x AND y se evalúa a cierto:
(Boolean( x ) == Boolean( y ) == true), false en el resto de los
casos

x || y; // devuelve true si x OR y se evalúa a cierto, si no false
```

En las nuevas implementaciones devuelven uno de los operandos:

```
expr1 && expr2; // devuelve expr1 si se evalúa a falso, si no
devuelve expr2

expr1 || expr2; // devuelve expr1 if se evalúa a cierto, si no
devuelve expr2
```

En JavaScript hay que tener en cuenta que las operaciones lógicas se evalúan de izquierda a derecha hasta que se pueda determinar la respuesta.

- **a || b** es automáticamente cierto si **a** es cierto. Así que no hay razón para evaluar **b**.
- **a && b** es falso si **a** es falso. Así que no hay razón para evaluar **b**.

```
&&    and

||    or

!     not (negación lógica)
```

OPERACIONES A NIVEL DE BIT

Operadores binarios

```
&     And

|     Or

^     Xor

<<    Desplazamiento a la izquierda (rellenando con ceros)

>>    Desplazamiento a la derecha (propagación de signo)

>>>   Desplazamiento a la derecha (rellenando con ceros)

      Para números positivos, >> y >>> devuelven el mismo
resultado.
```

Operadores unários

```
~     Not (invierte los bits)
```

OPERADORES CON CADENAS

```
=     Asignación
+     Concatenación
+=    Concatenación y asignación
```

Ejemplos

str = "ab" + "cd"; // "abcd"

str += "e"; // "abcde"

1.4 ESTRUCTURAS DE CONTROL

INSTRUCCIÓN "*IF ... ELSE*"

```
if (expr)
{
  sentencias;
}
else if (expr)
{
  sentencias;
}
else
{
  sentencias;
}
```

El operador condicional

En vez de usar "`if (){...} else {...}`", se puede usar una sintaxis abreviada:

```
var result = (param == condition) ? sentencia : otherwise;
```

es lo mismo que:

```
if (param == condition)
{
  result = sentencia;
}
else
{
  result = otherwise;
```

```
}
```

LA SENTENCIA "*SWITCH*"

```
switch (expr) {
  case VALOR:
    sentencias;
    break;
  case VALOR:
    sentencias;
    break;
  default:
    sentencias;
    break;
}
```

- `break;` es opcional; sin embargo es recomendarlo usarlo en la mayoría de los casos, porque si no lo usamos la ejecución continuara en el cuerpo de siguiente bloque case.
- Se pueden usar cadenas en los valores del case.
- Las llaves son obligatorias.

EL BUCLE "*FOR*"

```
for (expr-inicial; expr-condicion; expr evaluada después de cada
iteración) {
  sentencias;
}
```

EL BUCLE "*FOR ... IN ...*"

```
for (var nombre-propiedad in nombre-objeto) {
  sentencias usando nombre-objeto[nombre-propiedad];
}
```

- Iterará sobre todas las propiedades enumerables de un objeto
- **No es usable para vectores.** Podría iterar no solo sobre los índices de la vector, sino que también lo haría sobre otras propiedades visibles.

EL BUCLE "*WHILE*"

```
while (cond-expr) {
```

```
  sentencias;
}
```

EL BUCLE "*DO ... WHILE*"

```
do {
  sentencias;
} while (cond-expr);
```

EL BLOQUE "*WITH*"

```
with(document) {
  var a = getElementById('a');
  var b = getElementById('b');
  var c = getElementById('c');
};
```

- Se observa la ausencia de la palabra *document*, antes de cada invocación al método `getElementById()`.

1.5 FUNCIONES

Una función es un bloque con una lista de parámetros (puede ser vacía) que suele tener un nombre dado. Una función puede devolver un valor.

```
function nombre-funcion(arg1, arg2, arg3) {
  sentencias;
  return expresion;
}
```

También existen las funciones anónimas:

```
var fn = function(arg1, arg2) {
  sentencias;
  return expresion;
};
```

Ejemplo:

```
function gcd(segmentA, segmentB) {
  while (segmentA != segmentB) {
    if (segmentA > segmentB) {
```

```
      segmentA -= segmentB;
    } else {
      segmentB -= segmentA;
    }
  }
  return segmentA;
}
```

El numero de parámetros pasados cuando llamamos a una función, no tiene porque corresponderse con el numero de parámetros en la definición de la función. Un parámetro que existe en la definición y que no existe en la llamada toma el valor `undefined`. Dentro de la función también se puede acceder a los parámetros a través de la lista `arguments` usando índices.

```
arguments[0], arguments[1], ... , arguments[n]
```

Los tipos de datos básicos (cadenas, enteros, …) se pasan por valor, mientras que los objetos se pasan por referencia.

1.6 OBJETOS

Normalmente por simplicidad los tipos de subdividen en *primitivas* y *objetos*. Los objetos son entidades que tienen entidad (solo son iguales a si mismos) y asocian nombre de propiedades a valores.

JavaScript tiene distintos tipos de objeto predefinidos: *Array*, *Boolean*, *Date*, *Function*, *Math*, *Number*, *Object*, *RegExp* y *String*. Pueden existir otros predefinidos por el motor de ejecución, no por el lenguaje.

CREANDO OBJETOS

Los objetos pueden crearse usando una declaración, un inicializador o una función constructor:

```
// Declaración
var anObject = new Object();

// inicializador
var objectA = {};
var objectB = {index1:'value 1', index2:'value 2'};

// función constructor
function MiObjeto(atributoA, atributoB) {
  this.atributoA = atributoA;
```

```
  this.atributoB = atributoB;
}

// crea un Objeto
obj = new MiObjeto('rojo', 1000);
```

CONSTRUCTORES

Los constructores son la forma de crear múltiples instancias o copias de una objeto. JavaScript es un lenguaje basado en el prototipado de objetos, es decir la herencia se produce entre objetos no entre clases (JavaScript no tiene clases). Los objetos heredan sus propiedades de sus prototipos.

Las propiedades y métodos pueden crearse en el constructor o pueden añadirse y eliminarse una vez se ha creado el objeto. Para hacer esto para todas las instancias creadas por una función constructor, se usa la propiedad `prototype` del constructor para acceder al objeto prototipo. La eliminación de los objetos no es necesaria, puesto que el recolector de basura del motor de script se encarga de liberar las variables que ya no se referencian.

Ejemplo: Manipulando un objeto

```
// función constructor
function MiObjeto(atributoA, atributoB) {
  this.atributoA = atributoA;
  this.atributoB = atributoB;
}

// crea un Objeto
obj = new MiObjeto('rojo', 1000);

// accedemos a un atributo del objeto
alert(obj.atributoA);

// accedemos a un atributo usando la notación ["xxx"]
alert(obj["atributoA"]);

// añade una nueva propiedad
obj.atributoC = new Date();

// elimina una propiedad del objeto
delete obj.atributoB;

// elimina el objeto
delete obj;
```

HERENCIA

JavaScript soporta jerarquías de herencia a través del prototipado. Por ejemplo:

```
function Base() {
  this.Sobreescrita = function() {
    alert("Base::Sobreescrita()");
  }

  this.= function() {
    alert("Base::FuncionBase()");
  }
}

function Derivado() {
  this.Sobreescrita = function() {
    alert("Derivado::Sobreescrita()");
  }
}

Derivado.prototype = new Base();

d = new Derivado();
d.Sobreescrita();
d.FuncionBase();
```

Mostrará lo siguiente:

```
Derivado::Sobreescrita()
Base::FuncionBase()
```

Otra forma de implementar la sobreescritura de un método es la siguiente:

```
// Objeto base

function Base(paramA) {
  this.paramA = paramA;
}

Base.prototype.Sobreescrita = function() {
  alert("Base::Sobreescrita()");
}
```

```
Base.prototype.FuncionBase = function() {
  alert("Base::FuncionBase()");
}

// Objeto derivado

function Derivado(paramA, paramB) {
  this.parent = Base;
  this.parent(paramA);
  this.paramB = paramB;
}

Derivado.prototype = new Base();
Derivado.prototype.Sobreescrita = function() {
  alert("Derivado::Sobreescrita()");
}

d = new Derivado();
d.Sobreescrita();
d.FuncionBase();
```

1.7 EXCEPCIONES “TRY ... CATCH ... FINALLY”

JavaScript incluye una sentencia `try ... catch ... finally` para la gestión de excepciones, que permite capturar errores en tiempo de ejecución.

La sentencia `try ... catch ... finally` captura excepciones producidas por un error o una sentencia throw. La sintaxis es la siguiente:

```
try {
  // sentencias que pueden producir excepciones
} catch(error) {
  // sentencias para la gestión de las excepciones
} finally {
  //sentencias que se ejecutaran al finalizar pase lo que pase
}
```

Inicialmente, las sentencias del bloque try se ejecutan. Si se lanza una excepción, el control de flujo del script pasa al bloque catch, con la excepción disponible como el parámetro error, si no se produce una excepción el bloque catch se ignora. Una vez termina el bloque catch o el bloque try termina sin que se lance ninguna excepción, se ejecutan las sentencias en el bloque finally. Este bloque se suele usar para liberar recursos, que podrían no liberarse correctamente si se produjera un error grave en al ejecución. Este es un ejemplo del funcionamiento de la sentencia `try...catch...finally`:

```
try {
  // Creamos una vector
  arr = new Array();
  // Llamamos que a una función que puede que fallar
  func(arr);
}
catch (...) {
  // Escribimos el error en el log
  logError();
}
finally {
  // Aunque se ha producido un error grave, podemos liberar la
variable
  delete arr;
}
```

La parte del `finally` puede omitirse:

```
try {
  sentencias
}
catch (err) {
  // manejar errores
}
```

También se puede omitir la parte del `catch`:

```
try {
  sentencias
}
finally {
  // ignoramos los errores potenciales y liberamos los recursos
}
```

Estas sentencias requieren al menos un `catch` o `finally`. Es un error definir un bloque `try` aislado, aunque no se necesite manejar el error:

```
try { sentencia; } // ERROR
```

El argumento del catch también es necesario aunque no se use:

```
try { sentencia; } catch(   ) { sentencia; } // ERROR
```

Existen extensiones del ECMAScript estándar que permiten múltiples bloques catch de forma similar a Java:

```
try { sentencia; }
catch ( e if e == "InvalidNameException"  ) { sentencia; }
catch ( e if e == "InvalidIdException"    ) { sentencia; }
catch ( e if e == "InvalidEmailException" ) { sentencia; }
catch ( e ) { sentencia; }
```

1.8 OTRAS CONSIDERACIONES

CASE SENSITIVE

JavaScript es sensible a mayúsculas y minúsculas. Por lo que es recomendable seguir unas convenciones en la nomenclatura de las funciones y objetos. Es frecuente empezar los nombres de los objetos con una letra mayúscula y las funciones o variables con minúscula.

ESPACIOS EN BLANCO Y PUNTO Y COMA

Los espacios, tabuladores y saltos de línea usados fuera de constantes de tipo string son espacios en blanco. A diferencia de C, los espacios en blanco pueden causar efectos laterales en la semántica del script. Debido a una técnica llamada "inserción de punto y coma", cualquier sentencia esta bien formada si cuando aparece un salto de línea esta completa (como si apareciera un punto y coma antes del salto de línea). Se recomienda terminar las sentencias con saltos de línea para evitar la aparición de efectos laterales.

COMENTARIOS

La sintaxis de los comentarios es la misma que en C++ o Java

```
// comentario
/* comentario
   multilínea */
```

ACCESO A REGISTROS EN EBIMAP

Para acceder al valor de campos o registro de EBIMAP se usa la notación de vectores para acceder las propiedades sin especificar el literal con el nombre del objeto.

```
var a = [UNH.0062] // a tiene el valor del campo UNH.0062
```

En JavaScript a diferencia que en EdiwinScript los registros/campos son objetos inmutables y solo pueden ser referenciados para su lectura. Para la creación y/o modificación de los valores del registro EbiMap pon al servicio del programador funciones especificas para estas tareas:

```
setfieldvalue(«nombrecampo»,«valor»[,«cond»]);
overwritefieldvalue(«nombrecampo»,«valor»[,«cond»]);
```

CAPITULO 2 - OBJETOS GLOBALES

2.1 EL OBJETO OBJECT

INTRODUCCIÓN

Crea un objeto contenedor.

SINTAXIS

```
new Object([value])
```

PARÁMETROS

value

Cualquier valor.

DESCRIPCIÓN

El objeto *Object* es un objeto contenedor para gestionar un valor dado. Si este valor es *null* o *undefined*, se creara y devolverá un objeto vacío, en caso contrario devolverá un objeto de un tipo correspondiente al valor dado.

Cuando llamamos la función *Object* (sin el operador *new*), Objeto se comporta de forma idéntica

PROPIEDADES ESTÁTICAS

prototype

Permite añadir propiedades a todas las instancias del objeto *Object*.

Todos los objetos en JavaScript heredan del objeto *Object*; todos los objetos heredan los Métodos y propiedades de *Object.prototype*, aunque pueden ser sobrescritos. Por ejemplo otros prototipos de constructores sobrescriben la propiedad *constructor* y proporcionan sus propios métodos *toString*. Los cambios sobre *Object.prototype* se propagan todos los objetos, si las propiedades y métodos de estos no están sobrescritos.

PROPIEDADES

constructor

Especifica la función que crea el prototipo del objeto.

MÉTODOS

eval

Evalúa una cadena de código JavaScript en el contexto del objeto especificado.

getPrototypeOf

Devuelve el prototipo de un objeto especificado.

hasOwnProperty

Devuelve un booleano indicando si un objeto contiene la propiedad especificada como una propiedade directa y no a través de herencia.

isPrototypeOf

Devuelve un booleano indicando si un objeto hereda de otro objeto.

propertyIsEnumerable

Devuelve un booleano indicando la propiedad interna de ECMAScript *DontEnum* esta activada para este objeto.

toSource

Devuelve una cadena que representa el código fuente de un objeto equivalente. Este valor puede usarse para crear un nuevo objeto.

toLocaleString

Devuelve una representación textual del objeto.

toString

Devuelve una representación textual del objeto.

valueOf

Devuelve el valor del objeto especificado.

2.2 EL OBJETO STRING

INTRODUCCIÓN

Una cadena es una secuencia de caracteres.

SINTAXIS

```
new String()

new String(string)
```

Literales en la forma:

```
'stringText'

"stringText"
```

PARÁMETROS

string

Cualquier cadena.

stringText

Cualquier secuencia de caracteres propiamente codificada.

DESCRIPCIÓN

Los objetos *String* se crean llamando al constructor *new String()*. El objeto *String* es un objeto contenedor para gestionar valores de tipo string. La función global *String()* también puede llamarse si el operador *new* devolviendo una primitiva de tipo string.

Gracias a que Javascript convierte automáticamente entre primitivas y objetos de tipo String, se puede llamar cualquiera de los métodos del objeto *String* en una primitiva de tipo string. JavaScript convierte automáticamente la primitiva a un objeto *String* llama al método y descarta el objeto temporal. Por ejemplo:

```
s_obj.length;       // 3
s_prim.length;      // 3
s_also_prim.length; // 3
```

```
'foo'.length;       // 3
"foo".length;       // 3
```

(Un literal de cadena se puede crear usando comillas dobles o simples.)

Los objetos *String* se pueden convertir en primitivas mediante el método `String.valueOf()`.

Las primitivas y los objetos *String* dan resultados distintos cuando se evalúan en JavaScript. Las primitivas se tratan como código fuente y los objetos se tratan como un objeto de secuencia de caracteres. Por ejemplo:

```
s1 = "2 + 2";               // creates a string primitive
s2 = new String("2 + 2");   // creates a String object
eval(s1);                   // returns the number 4
eval(s2);                   // returns the string "2 + 2"
eval(s2.valueOf());         // returns the number 4
```

Acceso a caracteres

Hay dos maneras de acceder individualmente a un carácter en una cadena. La primera es el método *charAt*.

```
return 'cat'.charAt(1); // returns "a"
```

La otra es tratar la cadena como un array de caracteres, donde cada indice corresponde a un carácter individual:

```
return 'cat'[1]; // returns "a"
```

Aunque el segundo método no es parte de la especificación ECMAScript; es una funcionalidad de JavaScript.

En ambos casos, no se puede dar valor aun carácter de forma individual. En el primer caso *charAt* devolverá un error, mientras con el acceso a través del indice no se producirá ningún error pero el contenido de la cadena permanecerá invariable.

Comparando cadenas

En JavaScript se pueden usar los operadores > y < de forma similar al comportamiento de la función *strcmp()* del lenguaje C/C++:

```
var a = "a";
var b = "b";
if (a < b) // true
  print(a + " es menor que " + b);
else if (a > b)
```

```
  print(a + " es mayor que " + b);
else
  print(a + " y " + b + " son iguales.");
```

También se pueden obtener resultados similares usando el método *localeCompare* heredado por las instancias de *String*.

Propiedades estáticas

prototype

Permite añadir propiedades a todas las instancias del objeto String

Métodos estáticos

fromCharCode

Devuelve una cadena creada usando la secuencia especificada de valores Unicode.

Instancias del objeto String

Todas las instancias del objeto *String* heredan de *String.prototype* . Se puede modificar el prototipo del constructor para afectar a las propiedades y métodos heredados por todas las instancias del objeto *String*.

Propiedades

constructor

Especifica la función que crea el prototipo del objeto *String*, por defecto es la función *String*.

length

Refleja la longitud de la cadena.

Métodos

charAt(i)

Devuelve el carácter en el indice especificado.

charCodeAt (i)

Devuelve un numero indicando el valor Unicode del carácter en el indice especificado.

concat(str1, str2)

Combina el texto de dos cadenas y devuelve una nueva cadena.

indexOf(str)

Devuelve el indice de la primera ocurrencia del valor especificado en la cadena, o -1 si no lo encuentra.

lastIndexOf(str)

Devuelve el indice de la ultima ocurrencia del valor especificado en la cadena, o -1 si no lo encuentra.

localeCompare

Devuelve un número que indica si una cadena de referencia viene antes o después, o es la misma que la dada, siguiendo un criterio de orden.

match

Compara la coincidencia de una expresión regular frente una cadena.

quote

Encapsula la cadena entre comillas dobles .

replace(expr, s)

Ejecuta una búsqueda de las coincidencia entre una expresión regular y la cadena especificada, reemplazando las coincidencias con la cadena s

search(expr)

Ejecuta una búsqueda de las coincidencia entre una expresión regular y la cadena especificada.

slice

Extrae la sección de una cadena y devuelve una nueva cadena.

split()

Divide una cadena en un array de cadenas separando la cadena en subcadenas.

substr(idx, len)

Devuelve una cadena con los caracteres empezando en el indice *idx* y con la longitud *len*.

substring(i1, i2)

Devuelve una cadena con los caracteres entre dos indices.

toLocaleLowerCase()

Devuelve la cadena convertida a minúsculas de acuerdo al idioma actual.

toLocaleUpperCase()

Devuelve la cadena convertida a mayúsculas de acuerdo al idioma actual.

toLowerCase()

Devuelve la cadena convertida a minúsculas

toSource()

Devuelve un literal representando el objeto especificado, este valor se puede usar para crear un nuevo objeto. Sobrescribe el método *Object.toSource*.

toString()

Devuelve la cadena que representa el objeto especificado. Sobrescribe el método *Object.toString*.

toUpperCase()

Devuelve la cadena convertida a mayúsculas

trim()

Recorta los espacios en blanco del inicio y el final de la cadena

trimLeft()

Recorta los espacios en blanco en la parte izquierda de la cadena

trimRight()

Recorta los espacios en blanco en la parte derecha de la cadena

valueOf()

Devuelve una primitiva con el valor del objeto especificado. Sobrescribe el método *Object.valueOf*.

EJEMPLOS

Extensión de las instancias de string con el método *repeat*

El siguiente ejemplo se crea una función *"str_rep()"*, y se usa la sentencia *String.prototype.repeat = str_rep* para añadir el método a todos los objetos de tipo *String*. Todas las instancias de String tendrán ese nuevo método, aunque ya hayan sido creadas. Entonces el ejemplo crea una nueva función alternativa y sobrecarga el método , en una instancia usando la sentencia *s1.repeat = fake_rep*. El método str_rep del resto de objetos permanece inalterado.

```
var s1 = new String("a");
var s2 = new String("b");
var s3 = new String("c");

// Create a repeat-string-N-times method for all String objects
function str_rep(n) {
   var s = "", t = this.toString();
   while (--n >= 0) {
     s += t
   }
   return s;
}
String.prototype.repeat = str_rep;
s1a=s1.repeat(3); // returns "aaa"
s2a=s2.repeat(5); // returns "bbbbb"
s3a=s3.repeat(2); // returns "cc"

// Create an method and assign it to only one String variable
function fake_rep(n) {
   return "repeat " + this + " " + n + " times.";
}

s1.repeat = fake_rep
```

```
s1b=s1.repeat(1); // returns "repeat a 1 times."
s2b=s2.repeat(4); // returns "bbbb"
s3b=s3.repeat(6); // returns "cccccc"
```

La función del ejemplo también funciona con objetos String no creados con el constructor String. El siguiente código devuelve *"zzz"*.

```
"z".repeat(3);
```

2.3 EL OBJETO NUMBER

INTRODUCCIÓN

El objeto *Number* es un objeto contenedor para gestionar un valor numérico.

SINTAXIS

```
new Number(value)
```

PARÁMETROS

value

El valor numérico inicial del objeto que se creará.

DESCRIPCIÓN

Los principales usos del objeto *Number* son:

Si un parámetro no puede convertirse en un numero, devuelve *NaN*.

En contexto de ni constructor (ej., si el operador *new*), *Number* puede usarse para la conversión de tipos.

PROPIEDADES ESTÁTICAS

MAX_VALUE

El mayor numero representable.

MIN_VALUE

El menor numero representable.

NaN

Valor especial para representar "no numérico".

NEGATIVE_INFINITY

Valor especial para representar el infinito negativo.

POSITIVE_INFINITY

Valor especial para representar el infinito positivo.

prototype

Permite añadir propiedades a todas las instancias del objeto *Number*.

Instancias del objeto Number

Todas las instancias del objeto *Number* heredan de *Number.prototype* . Se puede modificar el prototipo del constructor para afectar a las propiedades y métodos heredados por todas las instancias del objeto *Number*.

Propiedades

constructor

Especifica la función que crea el prototipo del objeto *Number*, por defecto es la función `Number()`.

Ejemplos

Usando el objeto *Number* para asignar valores a variables numéricas

El siguiente ejemplo usa las propiedades del objeto *Number* para asignar valores a distintas variables numéricas:

```
biggestNum = Number.MAX_VALUE;
smallestNum = Number.MIN_VALUE;
infiniteNum = Number.POSITIVE_INFINITY;
negInfiniteNum = Number.NEGATIVE_INFINITY;
```

```
notANum = Number.NaN;
```

Usando *Number* para convertir un objeto *Date*

El siguiente ejemplo convierte un objeto *Date* a un “x “valor numérico usando *Number* como una función:

```
var d = new Date("December 17, 1995 03:24:00");
print(Number(d));
```

Esto muestra *"819199440000"*.

2.4 EL OBJETO BOOLEAN

INTRODUCCIÓN

El objeto *Boolean* es un objeto contenedor para gestionar un valor booleano.

SINTAXIS

```
new Boolean(value)
```

PARÁMETROS

value

El valor numérico inicial del objeto que se creará.

DESCRIPCIÓN

El valor pasado como primer parámetro se convertirá a un valor booleano, si es necesario. Si se omite el valor o es *0, -0, null, false, NaN, undefined*, o la cadena vacía (""), el objeto tendrá un valor inicial *false*. El resto de valores incluyendo cualquier objeto o la cadena *"false"*, creará un objeto con un valor inicial *true*.

No hay que confundir los valores de true y false de la primitiva booleana con los valores del objeto *Boolean*.

Cualquier objeto cuyo valor no es *undefined* o *null*, incluyendo un objeto *Boolean* cuyo valor es *false*, se evalúa a true cuando se pasa en una sentencia condicional. Por ejemplo la condicion en el siguiente sentencia se evalúa a *true*:

```
x = new Boolean(false);
if (x) {
  // . . . this code is executed
}
```

Este comportamiento no se aplica a las primitivas booleanas. Por ejemplo la condición en la siguiente sentencia se evalúa a *false*:

```
x = false;
if (x) {
  // . . . this code is not executed
```

```
}
```

No se debe usar un objeto *Boolean* para convertir un valor no booleano a un valor booleano, se debe usar la función Boolean para realizar esta tarea:

```
x = Boolean(expression);     // preferred
x = new Boolean(expression); // don't use
```

Si especificamos cualquier objeto, incluyendo un objeto Boolean cuyo valor inicial es *false*, para crear un nuevo objeto *Boolean*, El nuevo objeto *Boolean* tendrá un valor inicial *true*.

```
myFalse = new Boolean(false);   // initial value of false
g = new Boolean(myFalse);       // initial value of true
myString = new String("Hello"); // string object
s = new Boolean(myString);      // initial value of true
```

No se debe usar un objeto *Boolean* para reemplazar una primitiva booleana.

Propiedades estáticas

prototype

Permite añadir propiedades a todas las instancias del objeto *Boolean*

Instancias del objeto Boolean

Todas las instancias del objeto *Boolean* heredan de *Boolean.prototype* . Se puede modificar el prototipo del constructor para afectar a las propiedades y métodos heredados por todas las instancias del objeto *Boolean*.

Propiedades

constructor

Especifica la función que crea el prototipo del objeto Boolean, por defecto es la función *Boolean*.

Métodos

toSource()

Devuelve una cadena que representa el código fuente de un objeto *Boolean* equivalente. Este valor puede usarse para crear un nuevo objeto. Sobrescribe el método *Object.prototype.toSource*.

toString()

Devuelve una cadena "true" o "false" dependiendo del valor del objeto. Sobrescribe el método *Object.prototype.toString*

valueOf()

Devuelve el valor del objeto *Boolean*. Sobrescribe el método *Object.prototype.valueOf* .

EJEMPLOS

Creando objetos de tipo *Boolean* con el valor inicial a *false*

```
bNoParam = new Boolean();
bZero = new Boolean(0);
bNull = new Boolean(null);
bEmptyString = new Boolean("");
bfalse = new Boolean(false);
```

Creando objetos de tipo *Boolean* con el valor inicial a *true*

```
btrue = new Boolean(true);
btrueString = new Boolean("true");
bfalseString = new Boolean("false");
bSuLin = new Boolean("Su Lin");
```

2.5 EL OBJETO DATE

INTRODUCCIÓN

El objeto *Date* es un objeto contenedor para gestionar fechas y horas

SINTAXIS

```
new Date()
new Date(milliseconds)
new Date(dateString)
new Date(year, month, date [, hour, minute, second, millisecond])
```

PARÁMETROS

milliseconds

Número de milisegundos transcurridos desde el 01 de Enero de 1970 00:00:00 UTC.

dateString

Representación textual de la fecha debe estar en un formato reconocido por el método *Date.parse*.

year

Valor entero representando el año (formato YYYY)

month

Valor entero representando el mes, empezando con 0 para Enero a 11 para Diciembre.

date

Valor entero representando el día del mes.

hour

Valor entero representando la hora del día (formato 24 horas).

minute

Valor entero representando el minuto.

second

Valor entero representando el segundo.

millisecond

Valor entero representando los milisegundos.

DESCRIPCIÓN

Si no pasamos ningún parámetro, el constructor crea un objeto *Date* para la fecha actual de acuerdo a la hora local. Si se pasan solo algunos parámetros, los parámetros omitidos se interpretan con el valor 0. Los parámetros referentes a la fecha son obligatorios, mientras que los relativos a la hora son opcionales.

La fecha se mide en milisegundos desde la medianoche del 01 de Enero de 1970 UTC. Un día tiene 86.400.000 milisegundos. El rango del objeto Date es de -100,000,000 días a 100,000,000 días relativos al 01 de Enero de 1970 UTC.

El comportamiento del objeto *Date* es independiente de la plataforma. El objeto *Date* tiene métodos para la gestión de tiempo en UTC (universal) o en hora local. La hora local se refiere a la maquina donde se esta ejecutando el script.

Invocando *Date()* como función (sin el operador new) devolverá una cadena representando la fecha y hora actuales.

Propiedades estáticas

prototype

Permite añadir propiedades a todas las instancias del objeto *Date*.

Métodos estáticos

now

Devuelve un valor numérico correspondiente a la fecha y hora actuales.

parse

Analiza una representación textual de la fecha y devuelve el número de milisegundos desde el 01 de Enero de 1970 (hora local).

UTC

Acepta los mismos parámetros que la forma larga del constructor, y devuelve el número de milisegundos desde el 01 de Enero de 1970 UTC.

Instancias del objeto Date

Todas las instancias del objeto *Date* heredan de *Date.prototype*. Se puede modificar el prototipo del constructor para afectar a las propiedades y métodos heredados por todas las instancias del objeto *Date*.

Por compatibilidad todos los cálculos en formato de 2 dígitos, se basan en el año 2000, Se recomienda especificar siempre el año en cuatro dígitos. Existen métodos para trabajar con el año completo (4 dígitos): *getFullYear, setFullYear, getUTCFullYear* y *setUTCFullYear.*

Propiedades

constructor

Especifica la función que crea el prototipo del objeto *Date,* por defecto es la función *Date.*

MÉTODOS

getDate

Devuelve el día del mes para la fecha especificada de acuerdo a la hora local.

getDay

Devuelve el día de la semana para la fecha especificada de acuerdo a la hora local.

getFullYear

Devuelve el año de la fecha especificada de acuerdo a la hora local. (4 dígitos).

getHours

Devuelve la hora en la fecha especificada de acuerdo a la hora local.

getMilliseconds

Devuelve los milisegundos en la fecha especificada de acuerdo a la hora local.

getMinutes

Devuelve los minutos en la fecha especificada de acuerdo a la hora local.

getMonth

Devuelve el mes en la fecha especificada de acuerdo a la hora local.

getSeconds

Devuelve los segundos in la fecha especificada de acuerdo a la hora local.

getTime

Devuelve devuelve el número de milisegundos desde el 01 de Enero de 1970 UTC.

getTimezoneOffset

Devuelve el desplazamiento de la zona horaria en minutos para la configuración actual.

getUTCDate

Devuelve el día del mes en la fecha especificada de acuerdo a la hora universal (UTC).

getUTCDay

Devuelve el día de la semana en la fecha especificada de acuerdo a la hora universal (UTC).

getUTCFullYear

Devuelve el año en la fecha especificada de acuerdo a la hora universal (UTC). (4 dígitos).

getUTCHours

Devuelve las horas en la fecha especificada de acuerdo a la hora universal (UTC).

getUTCMilliseconds

Devuelve los milisegundos en la fecha especificada de acuerdo a la hora universal (UTC).

getUTCMinutes

Devuelve los minutos en la fecha especificada de acuerdo a la hora universal (UTC).

getUTCMonth

Devuelve el mes en la fecha especificada de acuerdo a la hora universal (UTC).

getUTCSeconds

Devuelve los segundos en la fecha especificada de acuerdo a la hora universal (UTC).

getYear

Devuelve el año en la fecha especificada de acuerdo a la hora local (2 dígitos). Se recomienda usar el método *getFullYear*.

setDate

Establece el día del mes para una fecha especificada de acuerdo a la hora local.

setFullYear

Establece el año para una fecha especificada de acuerdo a la hora local (4 dígitos).

setHours

Establece las horas para una fecha especificada de acuerdo a la hora local.

setMilliseconds

Establece los milisegundos para una fecha especificada de acuerdo a la hora local.

setMinutes

Establece los minutos para una fecha especificada de acuerdo a la hora local.

setMonth

Establece el mes para una fecha especificada de acuerdo a la hora local.

setSeconds

Establece los segundos para una fecha especificada de acuerdo a la hora local.

setTime

Establece el valor del objeto Date con el número de milisegundos desde el 01 de Enero de 1970 UTC especificados por el parámetro.

setUTCDate

Establece el día del mes para una fecha especificada de acuerdo a la hora universal (UTC).

setUTCFullYear

Establece el año para una fecha especificada de acuerdo a la hora universal (UTC). (4 dígitos).

setUTCHours

Establece la hora para una fecha especificada de acuerdo a la hora universal (UTC).

setUTCMilliseconds

Establece los milisegundos para una fecha especificada de acuerdo a la hora universal (UTC).

setUTCMinutes

Establece los minutos para una fecha especificada de acuerdo a la hora universal (UTC).

setUTCMonth

Establece los meses para una fecha especificada de acuerdo a la hora universal (UTC).

setUTCSeconds

Establece los segundos para una fecha especificada de acuerdo a la hora universal (UTC).

setYear

Establece el año para una fecha especificada de acuerdo a la hora local. Se recomienda usar el método *setFullYear*.

toDateString

Devuelve la fecha en un formato textual.

toGMTString

Convierte una fecha a string, usando la convención Internet GMT. Se recomienda usar el método *toUTCString*.

toLocaleDateString

Devuelve la fecha en un formato textual, usando la configuración local.

toLocaleFormat

Convierte una fecha a string, usando un formato dado.

toLocaleString

Devuelve una cadena que representa el objeto *Date*, usando la configuración local. Sobrescribe el método *Object.toLocaleString*.

toLocaleTimeString

Devuelve la hora en un formato textual, usando la configuración local.

toSource

Devuelve una cadena que representa el código fuente de un objeto *Date* equivalente. Este valor puede usarse para crear un nuevo objeto. Sobrescribe el método *Object.prototype.toSource*.

toString

Devuelve una cadena que representa el objeto *Date*. Sobrescribe el método *Object.prototype.toString*.

toTimeString

Devuelve la hora en un formato textual.

toUTCString

Convierte una fecha a texto, usando la convención UTC.

valueOf

Devuelve una primitiva con el valor del objeto *Date*. Sobrescribe el método *Object.prototype.valueOf*.

EJEMPLOS

Distintos métodos para asignar fechas

En el siguiente ejemplo se muestran distintos métodos para asignar fechas:

```
today = new Date();
birthday = new Date("December 17, 1995 03:24:00");
birthday = new Date(1995,11,17);
birthday = new Date(1995,11,17,3,24,0);
```

Calculando el tiempo transcurrido

En el siguiente ejemplo se muestra como determinar el tiempo transcurrido entre 2 fechas:

```
// using static methods
var start = Date.now();
// the event you'd like to time goes here:
doSomethingForALongTime();
var end = Date.now();
var elapsed = end - start; // time in milliseconds
// if you have Date objects
var start = new Date();
// the event you'd like to time goes here:
doSomethingForALongTime();
var end = new Date();
```

```
var elapsed = end.getTime() - start.getTime();
// time in milliseconds
```

2.6 EL OBJETO MATH

INTRODUCCIÓN

Es un objeto global predefinido que tiene propiedades para funciones y contantes matemáticas

DESCRIPCIÓN

A diferencia de otros objetos globales, *Math* no tiene constructor. Todas las propiedades y métodos son estáticos. Se puede referenciar a la constante *pi* como *Math.PI* e invocar la función seno como *Math.sin(x)*, donde ***x*** es el argumento del método. Las constantes están definidas con la precisión total de los números en JavaScript.

Se recomienda usar la sentencia with cuando una sección usa varias llamadas a constantes y métodos de la clase Math. Por ejemplo:

```
with (Math){
   a = PI * r*r;
   y = r*sin(theta);
   x = r*cos(theta);
}
```

PROPIEDADES ESTÁTICAS

E

La constante de Euler y base del logaritmo neperiano, aproximadamente 2.718.

LN2

Logaritmo neperiano de 2, aproximadamente 0.693.

LN10

Logaritmo neperiano de 10, aproximadamente 2.302.

LOG2E

Logaritmo en Base 2 de E, aproximadamente 1.442.

LOG10E

Logaritmo en Base 10 de E, aproximadamente 0.434.

PI

Relación entre la longitud de una circunferencia y su diámetro, aproximadamente 3.14159.

SQRT1_2

La raíz cuadrada de 1/2, aproximadamente 0.707.

SQRT2

La raíz cuadrada de 2, aproximadamente 1.414.

MÉTODOS ESTÁTICOS

abs(x)

Devuelve el valor absoluto de un número.

acos(x)

Devuelve el arco coseno (en radianes) de un número.

asin(x)

Devuelve el arco seno (en radianes) de un número.

atan(x)

Devuelve la arco tangente (en radianes) de un número.

atan2(x_1,...,x_n)

Devuelve la arco tangente del cociente de sus argumentos.

ceil(x)

Devuelve el menor entero mayor o igual que un número.

cos(x)

Devuelve el coseno (en radianes) de un número.

exp(x)

Devuelve E^x, donde *x* es el argumento, y E es la contante de Euler, la base del logaritmo neperiano.

floor(x)

Devuelve el mayor entero menor o igual que un número.

log(x)

Devuelve el logaritmo neperiano (base E) de un número.

max(x_1,...,x_n)

Devuelve el mayor de cero o más números.

min(x_1,...,x_n)

Devuelve el menor de cero o más números.

pow(x, exp)

Devuelve la base *x* elevada al exponente *exp*, x^{exp}.

random(x)

Devuelve un número pseudoaleatorio entre 0 y 1.

round(x)

Devuelve el valor de un número redondeado al entero más próximo.

sin(x)

Devuelve el seno (en radianes) de un número.

sqrt(x)

Devuelve la raíz cuadrada de un número.

tan(x)

Devuelve la tangente de un número.

toSource()

Devuelve la cadena "Math".

2.7 EL OBJETO ARRAY

INTRODUCCIÓN

El objeto *Array* es un objeto contenedor para gestionar datos matriciales

SINTAXIS

```
var arr1 = new Array(arrayLength);

var arr2 = new Array(element0, element1, ..., elementN);
```

Los literales array usan la siguiente sintaxis:

```
var lit = [element0, element1, ..., elementN];
```

PARÁMETROS

arrayLength

La longitud inicial del array. Se puede acceder a este valor usando la propiedad *length*. Si el valor especificado no es un numero se crea un array de longitud 1, que contendrá en el primer elemento el valor especificado. El tamaño máximo permitido es 4.294.967.295 (4GB).

$element_N$

Un valor para el elemento en esa posición del array. Cuando se usa esta nomenclatura, el array se inicializa con los valores de los elementos especificados, y la longitud se establece al número de parámetros.

DESCRIPCIÓN

Un array es un conjunto ordenado de valores asociados a un nombre de variable único. El objeto *Array* no se debería usar para crear un array asociativo, para ello se debería usar el objeto *Object*.

PROPIEDADES ESTÁTICAS

prototype

Permite añadir propiedades a todas las instancias del objeto *Array*.

Métodos estáticos

Aunque el objeto global *Array* define ninguna propiedad tiene métodos heredados.

Instancias del objeto Array

Todas las instancias del objeto *Array* heredan de *Array.prototype* . Se puede modificar el prototipo del constructor para afectar a las propiedades y métodos heredados por todas las instancias del objeto *Array*.

Ejemplos

Creación de un array

El siguiente ejemplo crea un objeto de tipo array usando un literal:

```
var coffees = ["Kenyan", "Columbian", "Kona"];
```

También se puede construir un array denso para dos o mas elementos empezando por el indice 0. usando el constructor *Array*:

```
var myArray = new Array("Hello", myVar, 3.14159);
```

Indexando un array

Asumimos que tenemos el siguiente array:

```
var myArray = new Array("Wind", "Rain", "Fire");
```

Se pueden referenciar los elementos de la siguiente forma:

- myArray[0] es el primer elemento
- myArray[1] es el segundo elemento
- myArray[2] es el tercer elemento

Especificando un único parámetro

Cuando especificamos un único valor numérico en el constructor *Array*, especificamos la longitud inicial del array. El siguiente código crea un array de cinco elementos:

```
var billingMethod = new Array(5);
```

El comportamiento del constructor *Array* depende si el parámetro es un numero o no.

- Si el valor especificado es un numero, el constructor convierte el numero a un entero de 32bits sin signo, y crea un array esta longitud. Inicialmente no contiene elementos pero puede tener una longitud distinta de cero.

- Si el valor especificado no es un numero, se crea un array de longitud 1, que en el primer elemento contiene el valor especificado.

El siguiente ejemplo crea un array de longitud 25 y asigna valor a los tres primeros elementos:

```
var musicTypes = new Array(25);
musicTypes[0] = "R&B";
musicTypes[1] = "Blues";
musicTypes[2] = "Jazz";
```

Incrementando la longitud de un array de forma indirecta

La longitud de un array se incrementa si asignamos valor aun elemento con un indice mayor que la longitud del array. El siguiente código crea un array de longitud 0 y después asigna valor al elemento 99. Cambiando la longitud de la matriz a 100.

```
var colors = new Array();
colors[99] = "midnightblue";
```

Creación de un array bidimensional

El siguiente ejemplo crea un tablero de ajedrez como un array de dos dimensiones.

```
var board =
[ ['R','N','B','Q','K','B','N','R'],
  ['P','P','P','P','P','P','P','P'],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  ['p','p','p','p','p','p','p','p'],
  ['r','n','b','q','k','b','n','r']];
print(board.join('\n') + '\n\n');
```

2.8 EL OBJETO REGEXP

INTRODUCCIÓN

El objeto *RegExp* es un objeto contenedor para gestionar datos expresiones regulares usadas para coincidencias de un patrón en una cadena.

SINTAXIS

```
var regex = new RegExp("pattern" [, "flags"]);

var literal = /pattern/flags;
```

PARÁMETROS

pattern

El texto de la expresión regular.

flags

Si se especifica, *flags* puede tener cualquiera de los siguientes valores:

g	coincidencia global
i	no sensible a mayúsculas y minúsculas
m	busca coincidencias sobre múltiples lineas

DESCRIPCIÓN

Cuando usamos la función constructor es necesario el uso de caracteres de escape propios de los strings:

```
var re = new RegExp("\\w+");
var re = /\w+/;
```

Los parámetros en formato litera no usan comillas para indicar la expresión, mientras que usando el constructor son necesarias las comillas:

```
/ab+c/i;
new RegExp("ab+c", "i");
```

Propiedades estáticas

prototype

Permite añadir propiedades a todas las instancias del objeto *RegExp*.

Instancias del objeto RegExp

Todas las instancias del objeto *RegExp* heredan de *RegExp.prototype*. Se puede modificar el prototipo del constructor para afectar a las propiedades y métodos heredados por todas las instancias del objeto *RegExp*.

Propiedades

constructor

Especifica la función que crea el prototipo del objeto *RegExp,* por defecto es la función *RegExp.*

global

Si la coincidencia del patrón en un string se aplicará globalmente.

ignoreCase

Si la coincidencia del patrón en un string sera sensible a mayúsculas y minúsculas.

lastIndex

El indice en el que se iniciara la búsqueda de la siguiente coincidencias.

multiline

Si la coincidencia del patrón en un string se aplicará a múltiples lineas.

source

El texto del patrón (expresión regular).

Métodos

exec(string)

Ejecuta una búsqueda de coincidencias sobre su parámetro string

test(string)

Busca una coincidencia sobre su parámetro string

toSource()

Devuelve una cadena que representa el código fuente de un objeto *RegExp* equivalente. Este valor puede usarse para crear un nuevo objeto. Sobrescribe el método *Object.prototype.toSource*.

toString()

Devuelve una cadena que representa el objeto *RegExp*. Sobrescribe el método *Object.prototype.toString*.

2.9 EL OBJETO FUNCTION

INTRODUCCIÓN

Todas las funciones en JavaScript son instancias del objeto *Function*.

SINTAXIS

```
new Function ([arg1[, arg2[, ... argN]],] functionBody)
```

PARÁMETROS

arg_1, arg_2, ... arg_N

Nombres usados en la función como nombres de los parámetros, Todos deben ser cadenas de caracteres correspondientes a un identificador valido de JavaScript o una lista de nombres separados por comas. Por ejemplo: *"x", "valor", o "a,b".*

functionBody

Una cadena de texto que contiene una serie de sentencias que conforman la definición de la función.

DESCRIPCIÓN

Los objetos *Function* creados con el constructor *Function* se evalúan cada vez que se usan, por este motivo es mas eficiente declarar la función directamente en el código, puesto que solo se evaluara una vez.

Todos los argumentos pasados a la función se trata como los nombre de los identificadores de los parámetros en la función creada en el ordena que se han pasado.

Invocar el constructor **`Function`** como función (sin el operador new), tiene el mismo efecto que evaluarlo como constructor.