PROGRAMACIÓN WEB – JS-I

Lenguaje de programación Javascript

JavaScript

Dr. José Raúl Romero Salguero
jrromero@uco.es

Contenidos

1. Aspectos básicos
2. Constructores básicos
3. Objetos en Javascript
4. DOM
5. Alertas y validación
6. Programación basada en eventos con Javascript

1.
Aspectos básicos

Interacción con Javascript

Acceso al contenido:

Se puede usar Javascript para acceder a cualquier elemento, atributo o texto desde una página HTML

Modificar contenido:

Se puede usar Javascript para agregar y / o eliminar elementos, atributos, texto de una página HTML

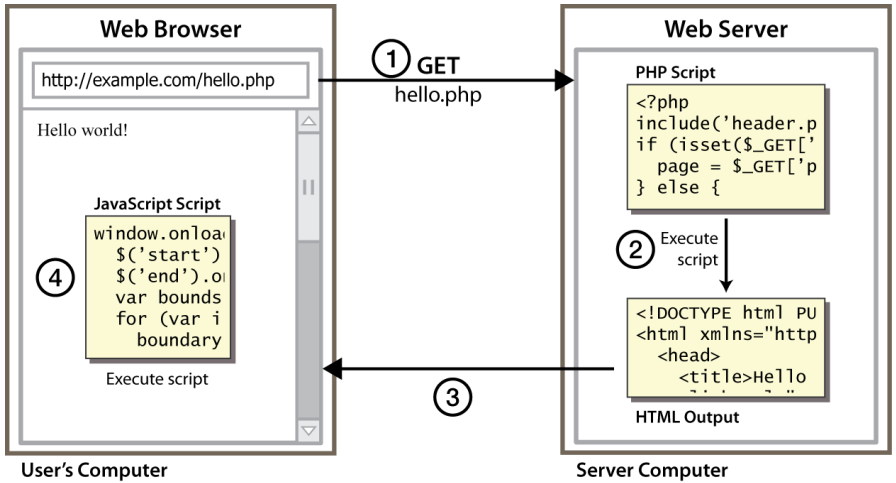
Incluir reglas al programa:

Se puede usar Javascript para regular la forma en que el usuario/navegador debe seguir para acceder o modificar el contenido de una página

Reaccionar a los eventos:

Se puede usar Javascript para indicar un *script* que debe ejecutarse cuando ocurre un evento específico

Secuencias de comandos en lado de cliente



Client-side script: el código se ejecuta en el navegador *después* de que la página se devuelve desde el servidor

Objetos y eventos

Cada cosa o concepto físico puede representarse como un **objeto** del mundo real

Cada objeto puede tener:

- **Propiedad** – par nombre/valor para cada característica
- **Método** – qué hace el objeto, su código (con el nombre en infinitivo)
- **Evento** – la forma en que el usuario interactúa con los objetos y que pueden cambiar los valores de sus propiedades (utilizando métodos)
  - Notificaciones de aquello que acaba de suceder, por ejemplo, cuando el conductor presiona el pedal del acelerador
  - Método de activación de eventos para responder a lo que acaba de suceder, por ejemplo, acelerar

2.
Constructores básicos

Comentarios

```
// comentario en una línea simple
/* comentario en varias líneas */
```

Idéntico a la sintaxis de comentarios de Java
Recordemos: sintaxis de comentarios depende del lenguaje:

HTML: `<!-- comment -->`

CSS/JS/PHP: `/* comment */`

Java/JS: `//comment`

PHP: `#comment`

```
/**
* Represents a book.
* @constructor
* @param {string} title - The title of the book.
* @param {string} author - The author of the book.
 */
function Book(title, author) { }
```

Es recomendable utilizar formato JSDoc para la generación automática de documentación a partir de comentarios: https://jsdoc.app/

Variables y tipos

Variables y tipos

```
var nombre = expression;                                    JS
```

```
var edad = 32;
var peso = 127.4;
var nombreCliente = "Pepe Pérez";                           JS
```

- Las variables se declaran con la palabra clave ***var*** (*case sensitive*)
  - Las variables no tienen tipo (conversion automática)
  - Identificadores pueden contener **letras**, **dígitos**, **$**, _ (no pueden empezar por un dígito)
- Los tipos no están especificados, pero JS tiene tipos ("tipados libremente")
  - **Number, Boolean, String, Array, Object, Function, Null, Undefined**
  - Puede averiguar el tipo de una variable llamando a ***typeof operando***

Valores especiales: `null` y `undefined`

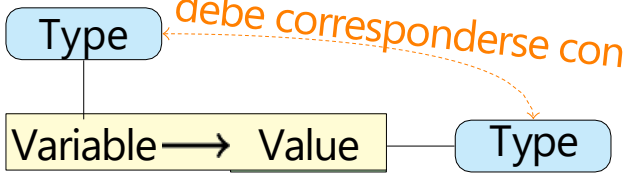
```
var ned = null;
var benson = 9;
var caroline;

//   ned --> null
//   benson --> 9
//   caroline --> undefined
```
JS

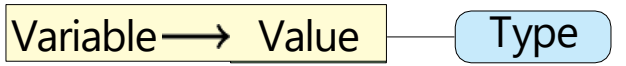
- `undefined`: no ha sido declarado, no existe
- `null`: existe, pero se le asignó específicamente un valor vacío o nulo

Javascript: Sistema de tipos

- JS es un lenguaje escrito de forma dinámica y flexible
- Lenguaje de programación estáticamente tipado:
  - Cada **variable** está vinculada a un **tipo** particular
  - Cada **variable** solo puede almacenar un **valor de tipo coincidente**



- Lenguaje de programación escrito dinámicamente:
  - Cada **variable** puede almacenar un **valor de tipo arbitrario**
  - Cada **variable** puede almacenar valores de diferentes tipos en diferentes momentos



Javascript: Sistema de tipos

- Lenguaje de programación fuertemente tipado:
  - En la invocación de una operación, cada **valor de argumento** debe ser de **tipo coincidente**
  - Los valores deben **convertirse explícitamente** al tipo coincidente (a menos que los tipos estén relacionados)

```
2.1 + 5 + Integer.parseInt("7")        // Java
```

- Lenguaje de programación libremente tipado:
  - En la invocación de una operación, cada **valor de argumento** se **convertirá implícitamente** al **tipo coincidente**

```
2.1 + 5 + "7"                  // Javascript
```

Javascript: Sistema de tipos

- Cada **valor** es de un **tipo** particular (o ninguno)

  519 1.9e3 son de tipo número (y solo de ese tipo)
  ’519’ "1.9e3" son de tipo cadena (y solo de ese tipo)

- Pero el **tipo de una variable** no necesita ser declarado.

```
var x; // declara x
```

- El **tipo de una variable** depende del valor que almacena actualmente y el tipo puede cambiar si se le asigna un valor de un tipo diferente.

```
x = 519; // x de tipo number
x = ’519 ’; // x de tipo string
```

Javascript: Sistema de tipos

- Las **declaraciones de funciones** no especifican el tipo de sus parámetros

```
function add(x, y) { return x + y; }
```

- En las **invocaciones a una función**, los tipos de argumentos se ajustarán automáticamente (si es posible)

```
add(519,1.9e3)     //  number 2419
add('519',"1.9e3") //  string '5191.9 e3'
add(519, '1.9e3')  //  string '5191.9 e3' '
add(true,1.9e3)    //  number 1901
```

- **Ventaja**: Programación más flexible
- **Desventaja**: Potencialmente se producen más errores

Coerción de tipos

- Javascript convierte automáticamente un valor al tipo apropiado según lo requiera la operación invocada (**coerción de tipos**)

```
5  * "3"  //  15
5  + "3"  //  "53"
5 && "3"  //  "3"
```

- El valor **`undefined`** se convierte de la siguiente manera:

Tipo	Default	Tipo	Default	Tipo	Default
bool	false	string	'undefined'	number	NaN

```
undefined || true //          true
undefined +  "-!" //   " undefined -!"
undefined +   1   //           NaN
```

Evaluación de código

- JS tiene la capacidad para generar, cargar e interpretar código fuente en tiempo de ejecución a través de una función **`eval`**.

```
var  x =  2;
var  y =  6;
var str = "if  (x > 0) {y /  x} else {-1}";
console.log (eval( str ));      //  Salida:  3
x = 0;
console.log (eval( str ));      //  Salida: -1
```

Tipo `Number`

```
var coste = 99;
var medianGrade = 2.8;
var creditos = 5 + 4 + (2 * 3);
```

- Los enteros y los números reales son del mismo tipo (**int** vs. **double**)
- Mismos operadores: + - * / % ++ -- = += -= *= /= %= **=
- Precedencia de operadores similar a Java

Tipo número: `NaN` e `Infinity`

- El tipo número de JS incluye constantes:
  - **`NaN`** (distingue entre mayúsculas y minúsculas) "no un número"
  - **`Infinity`** (distingue entre mayúsculas y minúsculas) "infinito"

- Las constantes **`NaN`** e **`Infinity`** se utilizan como **valores de retorno** para aplicaciones de funciones matemáticas que no devuelven un número

  - `Math.log(0)` devuelve `-Infinity`
  - `Math.sqrt(-1)` devuelve `NaN`
  - `1/0` devuelve `Infinity`
  - `0/0` devuelve `NaN`

Tipo número: `NaN` e `Infinity`

- Los operadores de igualdad y comparación se amplían para abarcar **`NaN`** e **`Infinity`**:

```
NaN == NaN                ~ false
Infinity == Infinity      ~ true
NaN == 1                  ~ false
NaN < NaN                 ~ false
1 < Infinity              ~ true
Infinity < 1              ~ false
NaN < Infinity            ~ false
```

```
NaN === NaN               ~ false
Infinity === Infinity     ~ true
Infinity == 1             ~ false
Infinity < Infinity       ~ false
1 < NaN                   ~ false
NaN < 1                   ~ false
Infinity < NaN            ~ false
```

Tipo número: `NaN` e `Infinity`

- Funciones para probar si un valor es o no `NaN`, `Infinity` or `-Infinity`:
  - `bool isNaN(value)`
- Devuelve `TRUE` *sii* el valor es `NaN`
  - `bool isFinite(value)`
- Devuelve `TRUE` *sii* el valor no es `NaN` o `Infinity/-Infinity`
- No hay ninguna función `isInfinite`
- En conversión a un valor booleano
  - `NaN` convierte a `false`
  - `Infinity` convierte a `true`
- En conversión a una cadena
  - `NaN` convierte a `'NaN'`
  - `Infinity` convierte a `'Infinity'`

Tipo `String`

```
var s = "Pepe Juan";
var fNombre = s.substring(0, s.indexOf(" "));  // "Pepe Juan"
var len = s.length;                            // 9
var s2 = 'Melvin Merchant';                    // can use "" or ' '
```

- **Métodos**: charAt, charCodeAt, fromCharCode, indexOf, lastIndexOf, replace, split, substring, toLowerCase, toUpperCase
  - **`charAt`** devuelve un `String` de una letra (no hay ningún tipo `char`)
- `length` es una propiedad (no es un método como en Java)
- Concatenación con **`+`** : `1 + 1` es `2`

  `"1" + 1` es `"11"`

Tipo `String`

```
var count = 10;
var s1 = "" + count;                          // "10"
var s2 = count + " bananas!";                 // "10 bananas!"
var n1 = parseInt("42 es la respuesta"); // 42
var n2 = parseFloat(“una cadena");        // NaN
```

- Las **secuencias de escape** se comportan como en Java : `\' \" \& \n \t \\`
- Para acceder a los caracteres de una cadena, use [*index*] or `charAt`:

```
var firstLetter = s[0];
var firstLetter = s.charAt(0);
var lastLetter = s.charAt(s.length - 1);
```

Tipo String - split, join

```
var s = "the quick brown fox";
var a = s.split(" ");          // ["the", "quick", "brown", "fox"]
a.reverse();                   // ["fox", "brown", "quick", "the"]
s = a.join("!");               // "fox!brown!quick!the"
```

- split rompe una cadena en un array usando un delimitador
  - También se puede usar con expresiones regulares rodeadas por /:

```
var a = s.split(/[ \t]+/);
```

- join combina un array en una sola cadena, colocando un delimitador entre los elementos

Tipo `Boolean`

```JS
var iLikeJS = true;
var ieMola = "IE6" > 0;    // false
if ("PW es genial!") {  /* true */ }
if (0) {  /* false */ }
```

- Las constantes "`true`" y "`false`" (*case sensitive*)
- Cualquier valor puede usarse como Boolean:
  - Valores falsos: `0`, `0.0`, `NaN`, `""`, `null`, y `undefined`
  - Valores verdaderos: cualquier otra cosa
- Convertir un valor en un Boolean explícitamente:
  - `var boolValue = Boolean(otherValue);`
  - `var boolValue = !!(otherValue);`

Coerción de tipos - `Boolean`

- Al convertir a `Boolean`, los siguientes valores se consideran `false`:
  - El mismo boolean `false`
  - El número `0` (cero)
  - El `string` vacío
  - `undefined`
  - `null`
  - `NaN`
- Cualquier otro valor se convierte en `true`, incluidos
  - Funciones
  - Objetos, en particular, matrices con elementos cero

Arrays

```
var frutas = ['Manzana', 'Banana'];
console.log(frutas.length);
```

```
var stooges = [];
stooges[0] = "Larry";
stooges[1] = "Moe";
stooges[4] = "Curly";
stooges[4] = "Shemp";
```

`Array` en Javascript es un objeto global (objeto tipo lista de alto nivel)

Arrays

- Es posible asignar un valor a `arrayVar.length`
  - Si el valor es mayor que el anterior de `arrayVar.length`, la matriz se 'extiende' con elementos `undefined`
  - Si el valor es menor que el anterior de `arrayVar.length`, se eliminarán los elementos del array con un índice igual o mayor
- Asignar un array a una nueva variable **crea una referencia** al mismo:
  - ~ los cambios en la nueva variable afectan al array original
  - Los arrays también se pasan a funciones por referencia
- La función `slice` se puede utilizar para crear una copia del array:

```
object arrayVar.slice(start, end)
```

devuelve una copia de los elementos del array con índices entre `start` y `end`

Arrays - Funciones

- Javascript no tiene estructuras de datos “pila” o “cola”, pero tiene funciones de pila y cola para arrays:
  - `number array.push(value1, value2,...)` – agrega uno o más elementos al final; devuelve el número de elementos en el array resultante
  - `mixed array.pop()` extrae y devuelve el último elemento
  - `mixed array.shift()` extrae y devuelve el primer elemento
  - `number array.unshift(value1, value2,...)` inserta uno o más elementos al comienzo del array; devuelve el número de elementos del array resultante

Objeto `Math`

```JS
var rand1to10 = Math.floor(Math.random() * 10 + 1);
var three = Math.floor(Math.PI);
```

- Métodos: abs, ceil, cos, floor, log, max, min, pow, random, round, sin, sqrt, tan
- Propiedades: E, PI

Operadores

Operadores lógicos

- Relacionales: `>  <  >=  <=`
- Lógicos: `&&  ||  !`
- Igualdad: `==  !=  ===  !==`
  - La mayoría de los operadores lógicos convierten automáticamente los tipos

```
5 < "7"                 //true
42 == 42.0              //true
"5.0" == 5              //true
```

  - Los `===` y `!==` son pruebas estrictas de igualdad; comprueba tanto el tipo como el valor:

```
"5.0" === 5             //false
```

Constructores de flujo

Declaración `if/else`

```
if (condición) {
  sentencias;
} else if (condición) {
  sentencias;
} else {
  sentencias;
}
```
JS

- ¡**Javascript** permite casi cualquier cosa como condición!

Bucle `for`

```
for (inicialización; condición; incremento) {
  sentencias;
}
```

```
var sum = 0;
for (var i = 0; i < 100; i++) {
  sum = sum + i;
}
```

```
var s1 = "hello";
var s2 = "";
for (var i = 0; i < s1.length; i++) {
  s2 += s1[i] + s1[i];
}
// s2 stores "hheellllloo"
```

Bucle `while`

```
while (condición) {
  sentencia;
}
```

```
do {
  sentencias;
} while (condición);
```

También existe `break` y `continue`, si bien su uso debe hacerse solo cuando sea estrictamente necesario

Bucles - `break` y `continue`

- `break` detiene la ejecución del bucle y puede usarse también con `while-`, `do while-`, y `for`

```
while (v < 100) {
  if (v == 0) break ;
  v ++
}
```

- `continue` detiene la ejecución de la iteración actual y mueve la ejecución a la siguiente iteración

```
for  (x = -2;x <= 2;x++) {
    if (x == 0) continue;
    document.writeln ("10/" + x + "= " +
}   (10/x));
```

```
10  /  -2 = -5
10  /  -1 = -10
10  /  1 = 10
10  /  2 = 5
```

Funciones

Declaración de funciones

```
function name(param1, param2, …) {
  sentencias;
  [return valor;}
}
```

```
function myFunction() {
  alert("Hola!");
  alert("Como estas?");
}
```

- Las declaraciones de las funciones se pueden evaluar en respuesta a los eventos del usuario
- El nombre de la función distingue entre mayúsculas y minúsculas
- `fn_name.length` se puede usar dentro del cuerpo de la función para determinar el número de parámetros

Funciones anónimas

```
function(parameters) {
  statements;
}
```

- Javascript permite declarar funciones anónimas (sin nombre)
- Puede almacenarse como una variable, adjuntarse como un controlador de eventos, etc.

Funciones anónimas - Ejemplo

```
window.onload = function() {
  var ok = document.getElementById("ok");
  ok.onclick = okayClick;
};

function okayClick() {
  alert("siiiuuu");
}
```

- Lo siguiente también es legal (aunque más difícil de leer y con peor estilo):

```
window.onload = function() {
  document.getElementById("ok").onclick = function() {
    alert("siiiuuu");
  };
};
```

Arrays - función `forEach`

- La forma recomendada de iterar sobre todos los elementos de un array es con **`for`**

```
for (index = 0;index < arrayVar.length;index ++) {
    ... arrayVar[index ] ...
}
```

- Una alternativa interesante es el uso de la función **`forEach`**:

```
var callback = function (elem,index,arrayArg) {
    statements
}
array.forEach(callback);
```

- **`forEach`** toma una función como argumento e itera sobre todos los índices
- Pasa como parámetros el elemento actual (*`elem`*), el índice actual (*`index`*) y un puntero al array (*`arrayArg`*)
- Los valores de retorno de esa función se ignoran
- La función **puede tener efectos secundarios**

Arrays – función forEach

```
var rewriteNames = function (elem , index , arr) {
  arr[index] = elem.replace(/(\w+)\s(\w+)/, "$2 ,$1");
}

var myArray = ['Dave Jackson','Ullrich Hustadt'];

myArray.forEach(rewriteNames);

for (i=0; i < myArray.length ; i++) {
  document.write ('['+ i+ '] = '+ myArray [i] + '  ' );
}
document.writeln ("<br/>");
```

```
[0] = Jackson , Dave [1] = Hustadt , Ullrich < br >
```

Variables globales

```
var count = 0;
function incr(n) {
  count += n;
}
function reset() {
  count = 0;
}

incr(4);
incr(2);
console.log(count);
```

count, **incr**, y **reset** son globales

- Se debe evitar el uso de variables globales
- Otros archivos JS pueden verlas y modificarlas

Variables globales

```
function everything() {
  var count = 0;
  function incr(n) {
    count += n;
  }
  function reset() {
    count = 0;
  }

  incr(4);
  incr(2);
  console.log(count);
}

everything();
```

1 símbolo global: **everything**

- El ejemplo anterior mueve todo el código a una función
- Las variables y funciones declaradas dentro de otra función son locales, no globales

Funciones anidadas

- Las declaraciones de funciones se pueden anidar en Javascript
- Las funciones internas tienen acceso a las variables de las funciones externas
- Por defecto, las funciones internas no se pueden invocar desde fuera de la función en la que se definen

```
function bubble_sort( array ) {
   function swap(i, j) {
      var tmp = array [i];
      array [i] = array [j];
      array [j] = tmp;
}
   if (!(array && array.constructor == Array))
      throw ("El argumento NO es un Array")
   for ( var i=0; i< array.length ; i++) {
      for ( var j=0; j< array.length - i; j++) {
         if ( array [j+1] < array [j]) swap(j, j+1)
   } }
   return array
}
```

El patrón *Module*

```JS
(function() {
    statements;
})();
```

- Envuelve todo el código de su archivo en una función anónima que se declara y se invoca inmediatamente
- ¡Se introducen **0** símbolos globales!
- Las variables y funciones definidas por su código no se pueden alterar externamente

Patron Module - Ejemplo

```
(function() {
  var count = 0;
  function incr(n) {
    count += n;
  }
  function reset() {
    count = 0;
  }

  incr(4);
  incr(2);
  console.log(count);
})();
```

Referencias a estilos CSS

```JS
function okayClick() {
  this.style.color = "red";
  this.className = "highlighted";
}
```

```CSS
.highlighted { color: red; }
```

- El código Javascript debe contener la menor cantidad de código **CSS** posible
- Si es necesario, JS deberá referenciar a clases / ID de elementos CSS
- En un archivo CSS, se definirán los estilos de esas clases / ID

Importación del *script*

- Cada vez que un navegador encuentra un elemento `script`, de forma predeterminada, deja de analizar el HTML restante hasta que el elemento `script` se haya descargado y procesado por completo
  - ~ Puede ocasionar mala experiencia de usuario (esperas) y errores
- "Solución segura": colocar los elementos del `script` al final del elemento `body`
- "Mejor solución": utilizar el atributo `async` o `defer` de `<script>` para cambiar el comportamiento predeterminado de descarga y procesado

Importación del *script*

```
<script src="jsLib1.js" async></script>
<script src="jsLib2.js" async></script>
```

- *Async* descarga de forma asíncrona el *script*, sin detener el análisis de HTML. Una vez descargado, detiene el renderizado del HTML y ejecuta el *script*. No se garantiza la ejecución de los *scripts* asíncronos en el orden de aparición en el documento

```
<script src="jsLib1.js" defer></script>
<script src="jsLib2.js" defer></script>
```

- *Defer* descarga de forma asíncrona el *script*, sin detener el análisis de HTML. La ejecución del *script* también es diferida, manteniendo el orden de aparición en el documento. No hay bloqueo en el renderizado

3.
Objetos en Javascript

Definición de objetos

Objetos Literales

- Javascript puede ser un lenguaje orientado a objetos, pero sin definir clases
- En su lugar, se debe declarar objeto literal, que luego servirá de base al resto

```
{ property1 : value1 , property2 : value2 , . . . }
```

donde `property1`, `property2` son nombres de propiedad y *`value1`*, *`value2`* son valores (expresiones)

```
var person1 = {
  age:     (30 + 2),
  gender: 'male',
  name:    {first : 'John', last : 'Doe'},
  interests: ['music', 'sports'],
  hello:  function () {
            return 'Hello! My name is' + ' '  + this.name.first + ' '
            + this.name.last;
  } };
```

```
person1.age              --> 32
person1['gender']        --> 'male'
person1.name.first       --> 'Bob'
person1 ['name']['last']--> 'Smith'
```

```
person1.fullname  --> " undefined undefined "
person1.fullname2 --> " undefined undefined "
```

Objetos Literales

```
var person1 = {
  name:   {first : 'John', last : 'Doe'},
  hello:  function () {
            return 'Hello! My name is' + ' '  + this.name.first + ' '
            + this.name.last;
          },
  fullname: this.name.first + ' ' + this.name.last,
  fullname2: name.first + ' ' + name.last
};
```

En `person1.hello()` el contexto de ejecución de `hello()` es `person1`

~ `name.first` **no** se refiere a `person1.name.first`

y `this.name.first` se refiere a `person1.name.first`

Objetos Literales

```
<form name="miForm">
   <p>
      <label>Nombre del formulario:
         <input type="text" name="text1" value="Beluga">
      </label>
   </p>
   <p>
      <input name="button1" type="button" value="Mostrar Nombre del Formulario"
onclick="this.form.text1.value = this.form.name">
   </p>
</form>
```

En combinación con la propiedad `form`, `this` se referirá al formulario actual (“miForm”)

Constructores de objetos

- En lugar de definir una clase, podemos definir una función que actúa como constructor de objetos
  - Las variables declaradas dentro de la función serán propiedades del objeto
    - Cada objeto tendrá su propia copia de estas variables
    - Es posible hacer que tales propiedades sean privadas o públicas
  - Las funciones internas (inner functions) serán métodos del objeto
    - Es posible hacer tales funciones / métodos privados o públicos
- Cada vez que se llama a un constructor de objetos, con el prefijo **`new`**, luego
  - se crea un nuevo objeto,
  - la función se ejecuta con la palabra clave `this` que está vinculada a ese objeto.

Objetos: definición y usos

```
function EjemploObject () {
  instVar2 = 'B';       //privada
  var instVar3 = 'C';   //privada
  this.instVar1 = 'A';  //publica
  this.method1 = function () {   // metodo publico
    // El uso de variables públicas precedido por 'this'
    return 'm1[' + this.instVar1 + ']' + method3();   }

  this.method2 = function () {   // metodo publico
    // El uso de un método público precedido por 'this'
    return 'm2 [' + this.method1 () + ']';  }

  method3 = function () {        // metodo privado
    return 'm3 [' + instVar2 + ']' + method4 ();  }

  var method4 = function () {    // metodo privado
    return 'm4 [' + instVar3 + ']';          }
}

obj = new EjemploObject ();
```

Objetos: definición y usos

```
function EjemploObject () {
  instVar2 = 'B';       //privada
  var instVar3 = 'C';   //privada
  this.instVar1 = 'A';  //publica
  this.method1 = function () {   // metodo publico
    // El uso de variables públicas precedido por 'this'
    return 'm1[' + this.instVar1 + ']' + method3();   }

  this.method2 = function () {   // metodo publico
    // El uso de un método público precedido por 'this'
    return 'm2 [' + this.method1 () + ']';  }

  method3 = function () {        // metodo privado
    return 'm3 [' + instVar2 + ']' + method4 ();  }

  var method4 = function () {    // metodo privado
    return 'm4 [' + instVar3 + ']';          }
}

obj = new EjemploObject ();
```

Las variables de instancia (propiedades) pueden almacenar valores tipo cadena o funciones

Cada objeto almacena su propia copia de los métodos

Prototipo

Objetos: propiedad prototipo

- Todas las funciones tienen una propiedad `prototype` que puede contener propiedades y métodos de objetos compartidos.
  - ~ los objetos no almacenan sus propias copias de estas propiedades y métodos, sino que solo almacenan referencias a una sola copia

```
function EjemploObject () {
  this.instVar1 = 'A'; // propiedad publica
  instVar2 = 'B'; // propiedad privada
  var instVar3 = 'C'; // propiedad privada

  EjemploObject.prototype.method1 = function () { ... } // metodo publico
  EjemploObject.prototype.method2 = function () { ... } // metodo publico

  method3 = function () { ... } // metodo privado
  var method4 = function () { ... } // metodo privado
}
```

- Las propiedades y métodos `prototype` son **siempre públicos**

Objetos: propiedad prototipo

```
obj1 = new EjemploObject ();
obj2 = new EjemploObject ();
document.writeln(obj1.instVar4);  // undefined
document.writeln(obj2.instVar4);  // undefined

EjemploObject.prototype.instVar4 = 'A';
document.writeln(obj1.instVar4);  // 'A'
document.writeln(obj2.instVar4);  // 'A'

EjemploObject.prototype.instVar4 = 'B';
document.writeln(obj1.instVar4);  // 'B'
document.writeln(obj2.instVar4);  // 'B'

obj1.instVar4 = 'C'; // crea una nueva propiedad para obj1

EjemploObject.prototype.instVar4 = 'D';
document.writeln(obj1.instVar4);  // 'C' !!
document.writeln(obj2.instVar4);  // 'D' !!
```

- La propiedad `prototype` se puede modificar "sobre la marcha"
  - ~ todos los objetos ya existentes obtienen nuevas propiedades/métodos
  - ~ la manipulación de propiedades / métodos asociados con la propiedad `prototype` debe hacerse con cuidado

```
obj3 = new EjemploObject ();

obj3.instVar4 == ??
```

Objetos: propiedad prototipo

```
// Se puede modificar prototype posterior a su declaracion
EjemploObject.prototype.instVar5 = 'E';
EjemploObject.prototype.setInstVar5 = function (arg) {
this.instVar5 = arg;
}

obj1 = new EjemploObject ();
obj2 = new EjemploObject ();
obj2.setInstVar5 ('F');

document.writeln (obj1.instVar5);    //  ??
document.writeln (obj2.instVar5);    //   ??
```

Variables y métodos de clase

```
function Circulo (radius) { this.r = radius; }

// 'class variable' - propiedad de Circulo
Circulo.PI = 3.14159;

// 'instance method'
Circulo.prototype.area = function () {
  return Circulo.PI * this.r * this.r; }

// ' class method ' - propiedad de Circulo
Circulo.max = function (cx , cy) {
  if (cx.r > cy.r) {     return cx;    } else {  return cy; } }

c1 = new Circulo (1.0); // Objeto
c1.r = 2.2;
c1_area = c1.area();

x = Math.exp (Circulo.PI);
c2 = new Circulo (1.2);
mayor = Circulo.max (c1, c2);
```

Las **propiedades de función** se pueden usar para emular las variables de clase de Java (**variables estáticas** compartidas entre objetos) y los métodos de clase

Variables estáticas privadas

```
var Persona = ( function () {
  var poblacion = 0;    // variable de clase estática privada
  return function (n) { // constructor -
    poblacion++;
    var nombre = n; // privada
    this.setNombre = function (n) { nombre = n; }
    this.getNombre = function () {return nombre;}
    this.getPob = function () {return poblacion;}
  }
} () )

per1 = new Persona ('Pedro');
per2 = new Persona ('Juan');
//per1.setNombre('David');
```

Para crear variables privadas estáticas compartidas entre objetos, podemos usar una función anónima de ejecución automática

```
console.log(per1.getNombre());  // ??
console.log(per2.getNombre());  // ??
console.log(per1.nombre);     // ??
console.log(per2.poblacion || per1.poblacion);  // ??
console.log(per1.getPob()); // ??
```

Bucle for/in-loop

El bucle for/in-loop nos permite pasar por las propiedades de un objeto

```
for ( var in object ) { statements }
```

Dentro del bucle podemos usar `object[var]` para acceder al valor de la propiedad `var`

```
var per1 = {
  edad: 32,
  genero: 'H',
  nombre: 'Juanito' }

for (prop in per1) {  document.writeln ('per1 [' + prop + '] es ' +
per1[prop] + '<br />'); }
```

per1 [edad] es 32
per1 [genero] es H
per1 [nombre] es Juanito

```
function Rectangulo(ancho, alto) {
  this.ancho = ancho;
  this.alto = alto;
  this.tipo = 'Rectangulo';
  this.area = function () {
return this.ancho * this.alto; }
}

function Cuadrado(largo) {
  this.ancho = this.alto = largo;
  this.tipo = 'Cuadrado'; }

// Cuadrado hereda de Rectangulo
Cuadrado.prototype = new Rectangulo();

var c1 = new Cuadrado(5);
document.writeln ("El area de c1 es " + c1.area());
```

Herencia

La propiedad `prototype` también se puede usar para establecer una **relación de herencia** entre objetos.

El area de c1 es 25

```
class Rectangulo {
  constructor(ancho, alto) {
    this.ancho = ancho;
    this.alto = alto;
    this.tipo = 'Rectangulo'; }

  get area() {
    return this.ancho * this.alto; }
}

class Cuadrado extends Rectangulo {
  constructor( largo ) {
    super(largo, largo);
    this.tipo = 'Cuadrado'; }
}

var c1 = new Cuadrado(5);
document.writeln("El area de c1 es " + c1.area );
```

‘Syntactic Sugar’ para clases

ECMAScript 2015 introdujo `class` como *syntactic sugar* para objetos basados en prototipos

El area de c1 es 25

Objetos predefinidos

Objetos predefinidos: RegExp

- Javascript tiene una colección de objetos predefinidos, que incluyen **Array**, **Date**, **RegExp**, **String**.
- Los objetos **RegExp** se denominan expresiones regulares.
- Las expresiones regulares son patrones que coinciden con cadenas.
- Las expresiones regulares se crean a través de:

```
/ regexp /              // literal de la expresión regular
new RegExp(' regexp ') // conversión de cadena a expr. regular
```

p

RegExp proporciona dos métodos:

`test(str)`	Prueba una coincidencia en una cadena, devuelve verdadero o falso
`exec(str)`	Ejecuta una búsqueda de una coincidencia en la cadena `str`, devuelve una matriz con una coincidencia o `null`

Objetos predefinidos: RegExp

- Las expresiones regulares más simples consisten en una secuencia de
  - ❑ caracteres alfanuméricos y
  - ❑ caracteres no alfanuméricos escapados por barra diagonal inversa:

  que coincide exactamente con esta secuencia de caracteres

```
/100\$/ matches "100 $" in " This 100 $ bill"
```

- Caracteres especiales y de control en expresiones regulares:

.	Coincide con cualquier carácter excepto el carácter de nueva línea \n
\n	Coincide con el carácter de nueva línea \n
\w	Coincide con un carácter de "palabra" (alfanumérico más "_")
\s	Coincide con un carácter de espacio en blanco
\d	Coincide con un carácter de dígito decimal

```
/\w\d/ empareja con "P5", "51" , y "19" en "COMP 519"
```

Objetos predefinidos: RegExp

^	Coincide con el inicio de entrada / línea
$	Coincide con el final de la entrada / línea
+	Coincide con la expresión anterior 1 o más veces
*	Coincide con la expresión anterior 0 o más veces
[*set*]	Coincide con cualquier carácter del conjunto (*set*) que consiste en caracteres, caracteres especiales y rangos de caracteres
[^*set*]	Coincide con cualquier carácter que no esté en el *set*

```
/^[a-z]+$/          Empareja con "abc", "x"
                    pero no con "0 abc", " abc1 ", " ab", " AB", ""

/^\s*[a-z]+\s*$/    empareja con "abc", "x", " ab"
                    pero no con"0abc","abc1","AB",  ""

/^[^a-z]+$/         Empareja con "AB","0123"
                    Pero no con"abc", "x", "0abc"
```

Existe una gama de caracteres especiales

Objetos predefinidos: String

- Un objeto **`String`** encapsula valores de tipo de dato cadena
- Las propiedades de un objeto **`String`** incluyen:
  - `length`: número de caracteres en la cadena (longitud)
- Los métodos de un objeto **`String`** incluyen:
  - `charAt(`*`index`*`)`: carácter en la posición *`index`* (desde 0).
  - `substring(`*`start`*`,`*`end`*`)`: devuelve la parte de una cadena entre las posiciones `start` (incluido) y `end` (excluido).
  - `toUpperCase()`: devuelve una copia de una cadena con todas las letras en mayúscula
  - `toLowerCase()`: devuelve una copia de una cadena con todas las letras en minúsculas

Objetos predefinidos : String y RegExp

- Los objetos `String` tienen métodos que usan expresiones regulares:
  - `search(`*`regexp`*`)` busca*`regexp`* en una cadena y devuelve la posición de inicio de la primera coincidencia si se encuentra, si no -1
  - `match(`*`regexp`*`)`
    - sin el modificador `g` devuelve los grupos coincidentes para la primera coincidencia o si no se encuentra ninguna coincidencia devuelve `null`
    - con el modificador `g` devuelve una matriz que contiene todas las coincidencias para toda la expresión
  - `replace(`*`regexp`*`,`*`replacement`*`)` reemplaza las coincidencias para *`regexp`* por *`replacement`*, y devuelve la cadena resultante

Objetos predefinidos : String y RegExp

```
name1 = 'Dave Shield'.replace (/(\w+)\s(\w+)/, "$2, $1");
regexp = new RegExp ("(\\w+)\\s(\\w+)");
name2 = 'Ken Chan'.replace(regexp, "$2, $1");

console.log(name1 + ' & ' + name2);
```

'Shield, Dave & Chan, Ken'

Objetos predefinidos: Date

- El objeto `Date` se puede usar para acceder a la fecha y hora (local).
- El objeto `Date` admite varios constructores:
  - `new Date()`: fecha y hora actual
  - `new Date(ms)`: establece la fecha en milisegundos desde el 1 de enero de 1970
  - `new Date(dateString)`: establece la fecha de acuerdo con `dateString`
  - `new Date(year, month, day, hours, min, sec, ms)`
- Los métodos más habituales para `Date`:
  - `toString()`: devuelve una representación de cadena del objeto `Date`
  - `getFullYear()`: devuelve una representación de cadena de cuatro dígitos del año (actual)
  - `parse()`: analiza una cadena de fecha y devuelve el número de milisegundos desde la medianoche del 1 de enero de 1970

Programación Web

Presentación de la asignatura__ **Curso 2019/20**