

Objetos en JavaScript

La clase Object. Clases de tipos primitivos

Math. Array. Date

RegExp. Error. JSON

- La clase Object. Clases de tipos primitivos
- Math. Array. Date
- RegExp. Error. JSON

En la mayoría de los lenguajes OOP (Java, C++, C#, PHP 7...) el concepto central es el de **CLASE**, a partir del cual se instancian los objetos

Clases y objetos



CLASE



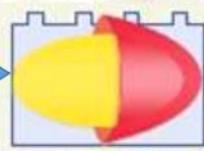
Entidad teórica (abstracta) que describe el comportamiento de los objetos

Atributos ---> posibles datos
Métodos ----> servicios que proporciona

OBJETOS

Fragmento de código

Atributos:
datos con un valor determinado



Métodos:
servicios que proporciona

INSTANCIACION
(en tiempo de ejecución)

Encapsulación
Polimorfismo
Herencia

En JavaScript la situación es completamente diferente

Tipos referencia: las “clases” de JS

Js

Todos los datos que no se ajustan a los tipos simples son de **tipo referencia** (i.e. clases)

En OOP, los objetos son **instancias** de la **clase** a la que pertenecen.

Sin embargo, en JavaScript **no se define el concepto de clase**

Los tipos de referencia se asemejan a las clases de otros lenguajes de programación y es frecuente referirse a ellos como **clases**

Objeto

Una colección de **propiedades** (elementos),
cada una con un nombre y un valor

Dato primitivo (número,
cadena o booleano)
Objeto
Función (método)

Técnicamente es una referencia al área de memoria donde se almacenan el conjunto de elementos que lo componen.

La clase Object(): Declaración de Objetos

JS

Los objetos en JavaScript se pueden crear declarándolos previamente.

mediante la palabra reservada **new** y el nombre de un tipo de datos (el equivalente a la clase que se va a instanciar), con lo que se simulará un método constructor

```
var variable = new <clase>(<valor>);
```

```
var var1 = new Object();
var var2 = new Array();
var var3 = new String("hola mundo");
```

Existen una serie de tipos predefinidos, equivalentes a “clases”, que luego veremos; además se pueden definir “clases” propias, definiendo el constructor adecuado

La inicialización del objeto (asignación de valores) suele realizarse a continuación de su declaración

Inicialización de Objetos (1)

Js

Los objetos en JavaScript se inicializan agregando sucesivamente **propiedades** con su correspondiente **valor**. Para ello pueden utilizarse diferentes notaciones.

1 Notación tradicional

Cada propiedad se indica con su nombre entre [] y a continuación se le asigna su valor

```
var stooge = new Object();
stooge['first_name'] = "Jerome";
stooge['last_name'] = "Howard";
```

```
var modulos = new Object();
modulos.titulos = new Object();
modulos.titulos['rss'] = "Lector RSS";
modulos.titulos['email'] = "Gestor de email";
modulos.titulos['agenda'] = "Agenda";
```

Inicialización de Objetos (2)

Js

2

Notación de puntos

Cada propiedad se indica con nombre objeto punto su nombre y a continuación se le asigna su valor

```
var stooge = new Object()  
stooge.first_name = "Jerome",  
stooge.last_name = "Howard";
```

```
var modulos = new Object();  
modulos.titulos = new Object();  
modulos.titulos.rss = "Lector RSS";  
modulos.titulos.email = "Gestor de email";  
modulos.titulos.agenda = "Agenda";
```

Inicialización de Objetos (3)

Notación JSON

Js

- 3 Como **objetos literales**, mediante el operador {}, que delimita los elementos (propiedades) que constituyen el objeto

Cada uno de estas propiedades se declara como **nombre : valor**

Esta forma de referir los objetos (y arrays) como literales se conoce como **JSON** (JavaScript Object Notation), y algunas veces aparece como una alternativa al formato XML

```
var empty_object = {};
// var stooge = new Object()
var stooge = {
    "first_name": "Jerome",
    "last_name": "Howard"
};
```

En JSON es frecuente prescindir de la declaración.

Objeto vacío

Objeto sencillo

```
var modulos = new Object();
modulos.titulos = {rss: "Lector RSS", email:
"Gestor de email", agenda: "Agenda"};
```

Objeto cuya única propiedad es a su vez otro objeto

Literales v. Constructores

Js

```
var aLiteral = [1, 2, 3, 4];
var aContructor = new Array(1, 2, 3, 4);
console.log(aLiteral);           -   -
                                [1, 2, 3, 4]
console.log(aContructor);       [1, 2, 3, 4]
console.dir(aLiteral);          ► Array[4]
console.dir(aContructor);       ► Array[4]

var oLiteral = {"nombre" : "Pepe", "edad" : 43}
var oContructor = new Object()
oContructor.nombre = "Pepe"
oContructor.edad = 43
console.log(oLiteral);
console.log(oContructor);
console.dir(oLiteral);
console.dir(oContructor);

                                Object {nombre: "Pepe", edad: 43}
                                Object {nombre: "Pepe", edad: 43}
                                ► Object
                                ► Object
```

Utilización de objetos

Js

Los objetos son útiles para recopilar y organizar datos; como pueden contener otros objetos, es fácil representar en ellos estructuras de árbol o de grafo.

Los valores correspondientes a una propiedad de un objeto pueden recuperarse utilizando dos formas posibles de notación

con el sufijo [] → `stooge["first-name"] // "Jerome"`

notación de puntos → `flight.departure.IATA // "SYD"`

De cualquiera de ambas formas se puede actualizar una propiedad, asignándole (=) un nuevo valor.

Como datos de tipo referencia que son, los objetos se pasan siempre por referencia. Nunca se crea una copia de un objeto como consecuencia de un pase por valor-

Ejercicio (11a)

Objeto Vuelo:

Crear un objeto flight (o vuelo) que almacene el nombre de aerolínea, el número de vuelo, los datos de salida y de llegada, con los siguientes valores airline: Oceanic, number: 815. Datos de salida (departure): clave IATA: SYD, fecha y hora (time) 2004-09-22 14:55, ciudad: Sydney. Datos de llegada (arrival): clave IATA: LAX, fecha y hora (time) 2004-09-23 10:42", ciudad: Los Angeles.

Comprobar como puede usarse cualquiera de las notaciones, incluso mezclándolas

Objetivo: Conocer la técnica para declarar objetos e inicializar una serie de propiedades.

Control de flujo. Iteraciones con objetos

Js

```
for(indice in <objeto>) {  
    ...  
}
```



Una variación de `for` para objetos , incluyendo `arrays`

El índice `i` vale sucesivamente el nombre de cada propiedad del objeto

Si se usa en un objeto, itera los nombres de todas las propiedades que estén definidas como *enumerable* (excluye por ejemplo `length` o `constructor`)

La propiedad `Object.isEnumerable()` permite conocer el valor de este atributo de cualquiera de las propiedades

Dentro del bucle

- `i` es el nombre de la propiedad
- `objeto[i]` es su valor

Iteraciones con objetos.

Ejemplo de for - in

JS

```
var dias = {"Lunes":"Monday", "Martes":  
"Tuesday", "Miercoles":"Wednesday", "Jueves":  
"Thursday", "Viernes":"Friday",  
"S\u00e1bado":"Saturday", "Domingo" : "Sunday"};
```

```
for(var i in dias) {  
    alert(i+"="+dias[i]);  
}
```

Lunes = Monday

Martes = Tuesday

Domingo = Sunday

...

Ejercicio (11b)

Meses del año:

Crear un objeto llamado meses y que almacene el nombre de los doce meses del año en español y en un segundo idioma. Mostrar por pantalla los doce pares de nombres en una **tabla**, utilizando document.write.

Objetivo: Comprobar el uso de arrays y del bucle for ... in, que se utiliza específicamente para recorrerlos

Ejercicio (11c)

Objeto vuelo:

Con el objeto vuelo creado anteriormente, crear una función que muestre por pantalla una **lista de varios niveles** con todas sus propiedades, utilizando document.write.

Objetivo: Comprobar el uso de arrays y del bucle for ... in, que se utiliza específicamente para recorrerlos

Propiedades: atributos y métodos

Js

Los objetos son bastante más que estructuras de árbol (grafo) para recopilar y organizar datos

entre sus propiedades se incluyen funciones (métodos)

```
var person = {  
    name: "Nicholas",  
    age: 29,  
    job: "Software Engineer",  
    sayName: function() {alert(this.name);}  
}
```

El uso de los métodos es equivalente al de las otras propiedades , incorporando el () y dentro los parámetros que existan

```
person.sayName()
```

Ejercicio (11d)

Objeto “Vuelo”:

Con el objeto vuelo creado anteriormente, transformar en método la función que muestra por pantalla una **lista de varios niveles** con todas sus propiedades, utilizando document.write.

Objetivo: Comprobar nuevamente el uso de arrays y del bucle for ... in, que se utiliza específicamente para recorrerlos **Incorporar** a los objetos un **método** (una función) que muestra todas sus propiedades. **Evitar** que dicho método se **muestre a si mismo** (o en general, evitar que muestre métodos).

Ampliación: incorporar el uso de **this**

Ejercicio (11e)

Objeto “Meses del año”:

Con el objeto “meses” creado anteriormente, transformar en método la función que muestra por pantalla una **tabla**, con todos los meses y su traducción.

Objetivo: Comprobar nuevamente el uso de arrays y del bucle for ... in, que se utiliza específicamente para recorrerlos **Incorporar** a los objetos un **método** (una función) que muestra todas sus propiedades. **Evitar** que dicho método se **muestre a si mismo** (o en general, evitar que muestre métodos).

Ampliación: incorporar el uso de **this**

Atributos y métodos de Object()

Js

Los objetos siempre tienen un mínimo número de elementos, al derivar del prototipo más genérico, Object()

`oObjeto = new Object()` → Objeto "vacío"
`oObjeto = {}`

```
▼ Object [Object]
  ▷ __proto__: Object
  ▷ __defineGetter__: function __define
  ▷ __defineSetter__: function __define
  ▷ __lookupGetter__: function __LookupGetter () { [native code] }
  ▷ __lookupSetter__: function __Lookup
  ▷ constructor: function Object() { [native code] }
  ▷ hasOwnProperty: function hasOwnProp
  ▷ isPrototypeOf: function isPrototype
  ▷ propertyIsEnumerable: function prop
  ▷ toLocaleString: function toLocaleSt
  ▷ toString: function toString() { [native code] }
  ▷ valueOf: function valueOf() { [native code] }
  ▷ get __proto__: function __proto__() { [native code] }
  ▷ set __proto__: function __proto__() { [native code] }
```

prototype : Referencia a las propiedades heredables

constructor : Referencia a la función que instancia el objeto

toString() : "[object nombre_del_objeto]"

valueOf() : El propio objeto

Tipos Referencia (Clases) predefinidos

Js

String	→ Objetos genéricos correspondientes a los tipos primitivos
Number	→ Objetos genéricos correspondientes a los tipos primitivos
Boolean	→ Objetos genéricos correspondientes a los tipos primitivos
Object	→ Objeto genérico
Function	→ Objetos que tiene código ejecutable asociado a ellos
Method	→ Objetos que tiene código ejecutable asociado a ellos
Constructor	→ Objetos que tiene código ejecutable asociado a ellos
Array	→ Objetos en los que los elementos están ordenados numéricamente (desde el 0)
Date	→ Objetos que representan fechas y horas
RegExp	→ Objetos que representan expresiones regulares
Error	→ Objetos que representan errores de sintaxis o en tiempo de ejecución

Objetos de tipos primitivos



JavaScript definen objetos para cada uno de los tipos de datos primitivos.

- objetos String
- objetos Number
- objetos Boolean

Sólo los objetos null y undefined carecen de la capacidad de invocar métodos

Almacenan los mismos valores de los tipos de datos primitivos y añaden **propiedades y métodos** para manipular sus valores.

Es posible, aunque poco frecuente crear de forma explícita estos objetos, invocando los constructores String(), Number(), or Boolean()

```
var S = new String(s);
var N = new Number(n);
var B = new Boolean(b);
```

Como veremos, Los números, cadenas y booleans se asocian automáticamente a objetos de estas clases que les permiten comportarse como si tuvieran métodos

Objetos de tipos primitivos

JavaScript definen objetos para cada uno de los tipos de datos primitivos

- objetos String
- objetos Number
- objetos Boolean

Sólo los objetos null y undefined carecen de la capacidad de invocar métodos

Almacenan los mismos valores de los tipos de datos primitivos y añaden propiedades y métodos para manipular sus valores.

Es posible, aunque poco frecuente crear de forma explícita estos objetos, invocando var N = new Number(n); los constructores String(), Number(), var B = new Boolean(b); or Boolean()

Como veremos, Los números, cadenas y booleans se asocian automáticamente a objetos de estas clases que les permiten comportarse como si tuvieran métodos

Tipos primitivos: literales v. objetos



```
var sTexto = "Hola Mundo";
var oTexto = new String("Hola Mundo");
console.log(sTexto);
console.log(oTexto);
console.dir(oTexto);
                                Hola Mundo
                                String {0: "H", 1: "o", 2: "l", 3:
                                ▶ String

var nNumero = 22;
var oNumero = new Number(22);
console.log(nNumero);
console.log(oNumero);
console.dir(oNumero);
                                22
                                Number {[[PrimitiveValue]]: 22}
                                ▶ Number

var bLogico = true;
var oLogico = new Boolean(true);
console.log(bLogico);
console.log(oLogico);
console.dir(oLogico);
                                true
                                Boolean {[[PrimitiveValue]]: true}
                                ▶ Boolean
```

Objetos envolventes



Los números, cadenas y booleanos se asocian automáticamente y de forma dinámica a objetos envolventes (*Wrapper Objects*)

Para determinar el número de caracteres de una cadena de texto escribimos

```
var texto = "hola mundo";
var longitud = texto.length;
```

La propiedad *length* sólo está disponible en los objetos *String*, por lo que en principio no debería poder utilizarse en un dato primitivo de tipo cadena de texto.

- JavaScript convierte el tipo de dato primitivo al tipo de referencia *String*, como si hiciera una llamada *new String(texto)*
- Como el objeto hereda propiedades y métodos de los objetos *String*, se puede obtener el valor de la propiedad *length*.
- Una vez que la propiedad ha devuelto el resultado, se descarta el objeto creado de forma transitoria.

Todo este proceso se realiza de forma automática y transparente para el programador.

Objetos envolventes

Los números, cadenas y booleanos se asocian automáticamente y de forma dinámica a objetos envolventes (*Wrapper Objects*)

Para determinar el número de caracteres de una cadena de texto escribimos

```
var texto = "hola mundo";
var longitud = texto.length;
```

La propiedad *length* sólo está disponible en los objetos *String*, por lo que en principio no debería poder utilizarse en un dato primitivo de tipo cadena de texto.

- JavaScript convierte el tipo de dato primitivo al tipo de referencia *String*, como si hiciera una llamada *new String(texto)*

- Como el objeto hereda propiedades y métodos de los objetos *String*, se puede obtener el valor de la propiedad *length*.

- Una vez que la propiedad ha devuelto el resultado, se descarta el objeto creado de forma transitoria.

Todo este proceso se realiza de forma automática y transparente para el programador.

Detalles de los objetos envolventes



String()
probablemente el objeto más complejo de JavaScript, con decenas de métodos y utilidades, que se utilizan gracias a la creación automática de objetos envolventes de este tipo

Boolean()
en general no se utilizan porque su comportamiento no siempre es idéntico al de los tipos de datos primitivos ya que en una operación lógica, cualquier objeto que exista se convierte a true, independientemente de su valor.

Number()
maneja decimales y enteros de forma consistente.

```
var var1 = new Number(16.2345);
var var2 = var1.valueOf(); // variable2 = 16.2345
var var3 = var1.toFixed(2); // variable3 = 16.23
```

Ejemplos ya conocidos



Hasta ahora al crear instrucciones hemos usado indistintamente

operadores → *typeof*
funciones → *isNaN()*
 → *parseInt()*
 → *parseFloat()*

E incluso las propiedades de los objetos conocidas como métodos
 → *[window.] alert()*
 → *[window.] prompt()*
 → *[window.] confirm()*

 → *función.arguments.length*
 → *función.arguments.callee.length*
 → *document.write()*
 → *var.toString()*



En estos últimos, muchas veces consecuencia de los objetos envolventes profundizaremos a continuación

Operaciones con números

Una vez creado el objeto envolvente Number, podemos recoger su valor con varios métodos

```
var x = 1.1;
```

Number.valueOf(): Devuelve un string con el número

```
x.valueOf() => "1.1"
```

Number.toFixed(n): devuelve un string redondeando a n decimales

```
x.toFixed(2) => "1.10"
```

Number.toExponential(n) : devuelve un string redondeando la mantisa a n decimales.

```
x.toExponential(2) => "1.10e+0"
```

Number.toPrecision(n) : devuelve un string redondeando a n dígitos

```
x.toPrecision(2) => "1.1"
```

Operaciones con cadenas (1)

Js

Gracias al tema ya comentado de los objetos envolventes, las operaciones con cadenas se basan en el gran número de métodos de los que dispone la clase String()

`length`, es el **atributo** que almacena la longitud de una cadena de texto

`concat()` o el operador `+`, se emplea para concatenar varias cadenas de texto

`split(separador)`, convierte una cadena de texto en un **array** de cadenas de texto. Es el opuesto de la función `join(separador)`, propia de los **arrays**

`toUpperCase()`, transforma todos los caracteres de la cadena a sus correspondientes caracteres en mayúsculas

`toLowerCase()`, transforma todos los caracteres de la cadena a sus correspondientes caracteres en minúsculas

Ejercicio (12a)

Comprobar si una cadena mezcla mayúsculas y minúsculas:

Definir una función que muestre información sobre una cadena de texto que se le pasa como argumento. A partir de la cadena que se le pasa, la función determina si esa cadena está formada sólo por mayúsculas, sólo por minúsculas o por una mezcla de ambas. Incluir algunos ejemplos predefinidos y la posibilidad de que el usuario introduzca una cadena para realizar la prueba.

Objetivo: Familiarizarnos con el uso de métodos propios de Strings como `toUpperCase()` o `toLowerCase()`.

Al mismo tiempo, profundizamos en el uso de funciones, el paso de parámetros y la devolución de valores..

Operaciones con cadenas (2)

Js

slice (inicio, final), substring(inicio, final) substr (inicio, desplazamiento)	'ciudades'.slice(2,5) => 'uda'
devuelven una porción de una cadena de texto, sin modificar el original	'ciudades'.substring(2,5) => 'uda' 'ciudades'.substr(2,5) => 'udade'

match (regExp), busca las coincidencias con una **expresión regular** y las devuelve como un array
search (expresión), busca una **cadena o expresión regular** y devuelve la posición en que comienza la coincidencia o -1 si no la hay
replace (expresión1, expresión 2), busca una **cadena o expresión regular** y devuelve una cadena en que la sustituye por la segunda expresión . No altera la cadena original

Operaciones con cadenas (3)

Js

charAt(posición), obtiene el carácter que se encuentra en la posición indicada Equivale a acceder al string como array	'ciudad'.charAt(2) => "u" 'ciudad'(2) => "u"
indexOf(carácter), calcula la posición en la que se encuentra el carácter indicado dentro de la cadena de texto.	'ciudad'.indexOf('da') => 3
lastIndexOf(carácter), calcula la última posición en la que se encuentra el carácter indicado dentro de la cadena de texto.	'ciudad'.lastIndexOd('d') => 5
charCodeAt(posición), obtiene el Unicode del carácter que se encuentra en la posición indicada String.fromCharCode(numero), convierte un valor numérico en el carácter Unicode correspondiente	'ciudad'.charCodeAt(2) => 117 String.fromCharCode(65) => 'A'

Operaciones con cadenas (4)

Js

Métodos "HTML": devuelven la cadena envuelta en determinadas etiquetas HTML

bold(), añade la etiqueta , correspondiente a la negrita.
italics(), añade la etiqueta <i>, correspondiente a la itálica
link(), crea un ancla para un hiperenlace
small(), añade la etiqueta de letra pequeña, <small>
sub(), añade la etiqueta de subíndice <sub>
sup(), añade la etiqueta de superíndice <sup>

anchor(), crea un ancla para un hiperenlace (descontinuada)
big(), añade la etiqueta <big> (descontinuada)
blink(), añade la etiqueta <blink> (descontinuada)
fixed(), añade la etiqueta de fuente fija, <tt> (descontinuada)
fontcolor(), aplica un determinado color a la fuente (descontinuada)
fontsize(), aplica un determinado tamaño a la fuente (descontinuada)
strike(), añade la etiqueta de tachado, <strike> (descontinuada)

Ejercicio (12b)

Comprobar E-mail.

Comprobar si una dirección de email contiene el carácter @ y el punto en su lugar.

Objetivo: Familiarizarnos con el uso de los métodos incluidos en los objetos String (), como indexOf() o lastIndexOf().

Al mismo tiempo, profundizamos en el uso de funciones, el paso de parámetros y la devolución de valores..

Ejercicios Extra (12c)

- Romper una cadena en dos mitades de igual longitud (o con 1 carácter más una de ellas si el total de caracteres es impar)
- Transformar una cadena en un nombre de variable válido: cambiar los espacios por guiones bajos y eliminar los caracteres no válidos y añadir un _ si el primer carácter es un número
- Recodificar una cadena en base a un número fijo o en base a uno variable, e.g. dependiente de la posición
- Decodificarla y recuperar su valor original
- comprobar si una cadena es un palíndromo

Objetivo: Familiarizarnos con el uso de los métodos incluidos en los objetos String (). Al mismo tiempo, profundizamos en el uso de funciones, el paso de parámetros y la devolución de valores..

La clase Math()

 JS

Permite dotar a JS de funciones matemáticas avanzadas y de constantes predefinidas, como el número PI..

random()	Número al azar		
sqrt(n)	Raíz cuadrada	pow(n_1, n)	Potencia: n_1^n
exp(n)	Exponencial: e^n	log(n)	Logaritmo natural
abs(n)	Valor absoluto	round(n)	Redondear
ceil(n)	Redondeo superior	floor(n)	Redondeo inferior
max(n_1, n_2)	Máximo	min(n_1, n_2)	Mínimo
sin(n)	Seno	acos(n)	Arco seno
cos(n)	Coseno	asin(n)	Arco coseno
tan(n)	Tangente	atan(n)	Arco tangente
		atan2(x,y)	Arco cotangente

Constantes matemáticas

Js

Constante	Valor	Significado
Math.E	2.718281828459045	Constante de Euler, base de los logaritmos naturales y también llamado <i>número e</i>
Math.LN2	0.6931471805599453	Logaritmo natural de 2
Math.LN10	2.302585092994046	Logaritmo natural de 10
Math.LOG2E	1.4426950408889634	Logaritmo en base 2 de Math.E
Math.LOG10E	0.4342944819032518	Logaritmo en base 10 de Math.E
Math.PI	3.141592653589793	Pi, relación entre el radio de una circunferencia y su diámetro
Math.SQRT1_2	0.7071067811865476	Raíz cuadrada de 1/2
Math.SQRT2	1.4142135623730951	Raíz cuadrada de 2

Arrays: La función Array()

Js

Los arrays son un tipo de objeto, en el que la agrupación de diversas variables (propiedades) tiene un carácter secuencial, en lugar del carácter asociativo de la clase object(). En este caso cada propiedad no está asociada a un nombre sino a una posición numérica dentro del array, siempre contando desde 0. Finalmente, los arrays **no suelen incorporar métodos**.

La declaración de los arrays, como la de los otros tipos referenciados, puede hacerse mediante su **constructor**

```
var array1 = new Array();
```

Existiendo o no declaración explícita, la inicialización de los arrays puede hacerse con cualquiera de las tres formas de notación ya mencionadas respecto a los objetos

Inicialización de Arrays

Js

Los arrays se pueden inicializar utilizando diferentes notaciones.

1

Notación tradicional

```
var aArray1 = new Array();
aArray[0] = 2;
aArray[1] = "hola";
aArray[2] = true;
aArray[3] = 45.34;
```

Cada propiedad se indica con su nombre entre [] y a continuación se le asigna su valor

Opcionalmente la declaración puede indicar la longitud del array

```
var variable1 = new Array(10);
```

2

En los objetos se utiliza también la notación de puntos, en la que cada propiedad se indica con nombre objeto punto su nombre.
Este sistema no se suele usar al asignar valores a los arrays, pero si al referirnos a sus atributos y métodos

Arrays literales. JSON

Js

- 3 Como objetos literales, utilizando la notación JSON, mediante el operador [], que delimita conjunto de valores o elementos que constituyen el array

```
var array1 = [valor0, valor1, ..., valorN];  
  
var aArray1 = [2, "hola", true, 45.34];
```

En JSON es frecuente prescindir de la declaración.

Otra variación de JSON es incluir los literales en la declaración

```
var aArray1 = new Array(2, "hola", true, 45.34);
```

JSON (JavaScript Object Notation) aparece algunas veces como una alternativa al formato XML

Ejercicio (13)

La letra del DNI:

1. Solicitar del usuario su DNI y su letra, utilizando la función prompt()
2. Comprobar si el número es válido (entre 0 y 99.999.999) y calcular la letra que le corresponde.
3. Comprobar si la letra indicada por el usuario es la correcta y comunicarle el resultado

Objetivo: Utilizar bucles de control condicionales (if). Acceder a un elemento de un array

NOTA: El cálculo de la letra del DNI es un proceso matemático sencillo que se basa en obtener el resto de la división entera del número de DNI y el número 23. A partir del resto de la división, se obtiene la letra seleccionándola dentro del siguiente array de letras:

```
var letras = ['T', 'R', 'W', 'A', 'G', 'M', 'Y', 'F', 'P', 'D', 'X', 'B', 'N', 'J', 'Z', 'S',  
'Q', 'V', 'H', 'L', 'C', 'K', 'E', 'T'];
```

Ejercicio (13)

La letra del DNI

:

1.Solicitar del usuario su DNI y su letra, utilizando la función prompt()

2.Comprobar si el número es válido (entre 0 y 99.999.999) y calcular la

letra que le corresponde.

3.Comprobar si la letra indicada por el usuario es la correcta y

comunicarle el resultado

Objetivo: Utilizar bucles de control condicionales (if). Acceder a un

elemento de un array

NOTA: El cálculo de la letra del DNI es un proceso matemático sencillo

que se basa en obtener el resto de la división entera del número de

DNI y el número 23. A partir del resto de la división, se obtiene la letra

seleccionándola dentro del siguiente array de letras:

var letras = ['T', 'R', 'W', 'A', 'G', 'M', 'Y', 'F', 'P', 'D', 'X', 'B', 'N', 'J', 'Z', 'S',

'Q', 'V', 'H', 'L', 'C', 'K', 'E', 'T'];

Control de flujo. Iteraciones con arrays

Js

```
for(<indice> = <num>; <indice> < <límite>; <indice>++) {  
    <array>[i] ....  
...}
```

Se itera a lo largo del índice del array ,
al recorrerlo usando el índice del bucle
como índice del array

```
for(indice in <objeto>) {  
...}
```

Una variación de *for* para objetos ,
incluyendo *arrays*

i vale sucesivamente el nombre de cada propiedad
del objeto o el índice de cada elemento del array

Ejercicio (14a)

Array de números aleatorios.

Realizamos un script capaz de llenar una lista de longitud dada, entre 10 y 25, con números enteros aleatorios comprendidos entre 0 y 100.

Detalles:

- El cálculo debe hacerlo una función
- La presentación de los datos debe estar fuera de la función

Objetivo: Familiarizarnos con el uso de los objetos Math().
Comprobar la creación de un array empleando un bucle for

Ejercicio (14b)

Buscar en un array.

Tenemos una lista de nombres (e.g. reyes de un país) y deseamos determinar si un nombre en concreto está en la lista, y si es así en qué posición se encuentra. Si el nombre no está en la lista la función debe devolver un valor booleano (false).

```
'Fernando',
'Isabel',
'Juana',
'Carlos',
'Felipe',
'Luis',
'Jose',
"Amadeo",
'Alfonso',
'Juan Carlos'
```

Objetivo: Familiarizarnos con el uso de los objetos arrays



Operaciones con arrays (1)

Js

Nuevamente, las operaciones con arrays se basan en el gran número de métodos de los que dispone la clase Array(), que muchas veces son análogos a los vistos para las cadenas

`length`, calcula el número de elementos de un array

`concat()`, devuelve un array en el que se concatenan varios de ellos (+ los concatena como strings)

`join(separador)`, devuelve una cadena de texto en la que une todos los elementos de un array, separados por el argumento de la función.

`split(separador)`, es la función contraria a `join()`: convierte una cadena de texto en un array de cadenas de texto.

Estas tres funciones devuelven un valor, pero no modifican al array al que afectan

→ implementadas como inmutables

Operaciones con arrays (2)

Js

sort(), Ordena alfabéticamente los elementos de un array.
Por defecto el orden es alfabético y ascendente (A -> Z)
Opcionalmente podemos pasar como argumento una función
"comparadora", cuyos dos argumentos determinaran el orden, según
devuelva positivo, 0 o negativo.,.

```
var points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a-b});
```

reverse(), modifica un array colocando sus elementos en el orden inverso a su posición original:

Como resultado de aplicar estas funciones,
cambia el valor del array al que ha sido aplicada → implementadas
como mutables

Ejercicio (15a)

Reconocer un palíndromo:

Definir una función que determine si la cadena de texto que se le pasa como parámetro es un palíndromo, es decir, si se lee de la misma forma desde la izquierda y desde la derecha.

Objetivo: Familiarizarnos con las operaciones con arrays, como
`split()`, `reverse()` ...
Al mismo tiempo, profundizamos en el uso de funciones, el paso
de parámetros y la devolución de valores..

- Palíndromo:
- *La ruta nos aportó otro paso natural*
 - *Dábale arroz a la zorra el abad*

Ejercicio (15b)

Reconocer un array de objetos:

Crear un array de al menos 6 objetos, cada uno de ellos con las propiedades nombre y edad. Crear una función que lo reordene aleatoriamente. Mostrar el resultado por pantalla.
Crear una segunda función que lo reordene ascendentemente en función de la propiedad edad de cada una de los objetos

Objetivo: Familiarizarnos con las operaciones con arrays, como sort() y el uso de funciones para la ordenación.

Al mismo tiempo, profundizamos en el uso de funciones, el paso de parámetros y la devolución de valores..

Operaciones con arrays (3)

Js

slice (inicio,fin), extrae parte de un Array (desde el primer parámetro hasta el segundo, sin incluirlo) devolviéndolo en un nuevo objeto Array, sin modificar el original.

```
var aNombres = ["Luis", "Juan", "Jose", "Carlos"];
var aParte = aNombres.slice(1,3)
// El array aNombres: Luis,Juan,Jose,Carlos
// El array aParte: Juan,Jose
```

splice(inicio,cuantos, <valor>, <valor>,...), extrae parte de un array, eliminando, a partir de donde inicia la extracción, los valores indicados en el segundo parámetro, y en su lugar introduce los valores que le indicamos (si los hay). Si el segundo parámetro es 0, no extrae nada, sólo añade

```
var aNombres = ["Luis", "Juan", "Jose", "Carlos"];
var aParte = aNombres.splice(1,2, "Raul", "Alberto")
// El array aNombres: Luis,Raul,Alberto,Carlos
// El array aParte: Juan,Jose
```

Operaciones con arrays (4)

Js

Pilas y Colas

`push()`, añade un elemento (o n) al final del array.

`pop()`, elimina el último elemento del array y lo devuelve.

Permiten reproducir una pila (stack) de datos: *FIFO* ("First In, Last Out" o "primero en entrar, último en salir").

`shift()`, elimina el primer elemento del array y lo devuelve.

Junto con `push()`, permite reproducir una cola (heap) de datos: *FIFO* ("First In, First Out" o "primero en entrar, primero en salir").

`unshift()`, añade un elemento (o n) al principio del array.

Operaciones con arrays en JS 1.5

Js

- `map()`,
 - `filter()`,
 - `some()`,
 - `every()`,
 - `forEach()`,
 - `reduce()`,
 - `reduceRight()`,
- En todos los nuevos métodos de utiliza una **función callback**, es decir una función que es pasada como parámetro para que el método la utilice de la forma en que tiene previamente definida.

Los parámetros de dicha función son (element, index, array)

Operaciones con arrays en JS 1.5

Arrays en JS 1.5 (1)

Js

`map()` una proyección (como select en C#): el argumento es una función que transforma cada uno de los elementos.

```
array.map(function(i){return i.toUpperCase()});  
convertiría cada elemento del array en mayúsculas
```

```
var numbers = [1, 4, 9];  
var roots = numbers.map(Math.sqrt);  
document.write("roots is : " + roots );
```

mostraría las raíces cuadradas de los elementos del array

Un ejemplo más complejo, podría transformar en objetos cada uno de los elementos del array

```
•map()  
•filter()  
•some()  
•  
every()  
•  
forEach()  
•  
reduce()  
•  
reduceRight()  
,
```

mostraría las raíces cuadradas de los elementos del array

Un ejemplo más complejo, podría transformar en objetos cada uno de los elementos del array

ECMAScript 5.1 (ECMA-262)

Arrays en JS 1.5 (2)

Js

`filter()` tiene como argumento una función lambda que evalúa cada elemento del array y devuelve un booleano. Se devuelve un **nuevo array** sólo con los elementos que hayan dado verdadero en la función callback

```
array.filter(function(i){return i[0]==="a"});
```

```
function isBigEnough(element, index, array) {
    return (element >= 10);
}
var aDatos = [12, 5, 8, 130, 44]
var aFiltrado = aDatos.filter(isBigEnough);
console.log(aFiltrado);
```

Ejemplo que devuelve [12,130,44]

ECMAScript 5.1 (ECMA-262)

Ejercicio (16a)

Crear y modificar un array de objetos:

Crear un array de al menos 6 nombres, para luego convertirlo en un array de objetos, cada uno de ellos con las propiedades nombre y edad, a la que se le asigna un valor aleatorio entre 16 y 19 (años enteros). Mostrar el resultado por pantalla.

Crear una segunda función que elimine a los menores y muestre el resultado por pantalla.

Objetivo: Familiarizarnos con las operaciones con arrays, utilizando algunas de las novedades de JS 1.5, como `map()` y `filter()`.

Al mismo tiempo, profundizamos en el uso de funciones, el paso de parámetros y la devolución de valores..

Arrays en JS 1.5 (3)

Js

`some()` y `every()` utilizan funciones del mismo tipo para evaluar si el array en su conjunto las cumple, y devolver en consecuencia verdadero o falso.

- `some()` devuelve verdadero si algún elemento del array lo devuelve
- `every()` devuelve verdadero si todos los elementos del array lo devuelven

```
function isBigEnough(element, index, array {  
    return (element >= 10);  
}  
var aDatos = [12, 5, 8, 130, 44]  
var isVal_some = aDatos.some(isBigEnough);  
var isVal_every = aDatos.every(isBigEnough);  
console.log(isVal_some);  
console.log(isVal_every);
```

true
false

ECMAScript 5.1 (ECMA-262)

Arrays en JS 1.5 (4)

Js

`forEach()` permite indicar cualquier función , booleana o no, que modifique o no los elementos, pero que en cualquier caso se aplica sobre cada uno de ellos.

```
function printBr(element, index, array) {  
    document.write("<br />[" + index + "] is " + element );  
}  
aDatos = [12, 5, 8, 130, 44];  
aDatos.forEach(printBr);  
  
function cuad(element, index, array) {  
    array[index] = element * element;  
}  
aDatos = [12, 5, 8, 130, 44];  
aDatos.forEach(cuad);  
console.log(aDatos);
```

[0] is 12
[1] is 5
[2] is 8
[3] is 130
[4] is 44

[144, 25, 64, 16900, 1936]

ECMAScript 5.1 (ECMA-262)

Arrays en JS 1.5 (5)

Js

[reduce\(\)](#), aplica una función simultáneamente a pares de valores del array, desde la izquierda a la derecha, sucesivas veces, hasta reducir el array a un único valor

[reduceRight\(\)](#), realiza el mismo proceso desde la derecha a la izquierda, de nuevo sucesivas veces, hasta reducir el array a un único valor

```
var aDatos = [1, 2, 3, 4];
var nTotal = aDatos.reduce(function(a, b){ return a * b; });
console.log(aDatos);
console.log(nTotal);
```

[1, 2, 3, 4]

24

ECMAScript 5.1 (ECMA-262)

Ejercicio (16b)

Crear y modificar un array de objetos:

Partimos de array del ejercicio anterior, que convertimos en un array de objetos, cada uno de ellos con las propiedades nombre y edad, a la que se le asigna un valor aleatorio entre 16 y 19 (años enteros) y de nuevo mostrar el resultado por pantalla.

Crear dos funciones que muestren un aviso cuando haya menores en la lista, usando respectivamente some() y every()

Objetivo: Familiarizarnos con las operaciones con arrays, utilizando algunas de las novedades de JS 1.5, como some() y every().

Al mismo tiempo, profundizamos en el uso de funciones, el paso de parámetros y la devolución de valores..

Objetos Date()

Js

Permite representar y manipular valores relacionados con fechas y horas.

```
var fecha = new Date(año, mes(*), dia, hora, minuto, segundo);
```

la fecha y hora se almacena como el número de milisegundos que han transcurrido desde el 1 de Enero de 1970 a las 00:00:00.

(*) los meses indican normalmente pero luego el sistema los devuelve de 0 (Enero) a 11 (Diciembre)

Formas de instanciaión

```
var fecha = new Date()
// :fecha actual, al instanciar la clase sin parámetros
var fecha = new Date(1000000000000000);
// "Sat Nov 20 2286 18:46:40 GMT+0100"
var fecha = new Date(2009, 5, 1, 19, 29, 39);
// 1 de Junio de 2009 (19:29:39)
```

Imprimir el valor de un Date()

```
toString()  
toDateString()  
toGMTString()  
toISOString()  
  
now = new Date()  
  
now.toDateString() // "Wed Mar 26 2014"  
  
now.toString() // "Wed Mar 26 2014 17:11:07  
GMT+0100 (Hora estándar romance)"  
  
now.toGMTString() // "Wed, 26 Mar 2014 16:11:07  
GMT"  
  
now.toISOString() // "2014-03-26T16:11:07.179Z"
```

Operaciones con la clase Date()

Js

`getTime()` – devuelve un número que representa la fecha como el número de milisegundos transcurridos desde la referencia de tiempos (1 de Enero de 1970).

`getMonth()` – devuelve el número del mes de la fecha (empezando por 0 para Enero y acabando en 11 para Diciembre)

`getFullYear()` – devuelve el año de la fecha como un número de 4 cifras.

`getYear()` – devuelve el año de la fecha como un número de 2 cifras.

`getDate()` – devuelve el número del día del mes

`getDay()` – devuelve el número del día de la semana (0 para Domingo, 1 para Lunes, ..., 6 para Sábado)

`getHours()`, `getMinutes()`, `getSeconds()`, `getMilliseconds()` – devuelve respectivamente las horas, minutos, segundos y milisegundos de la hora correspondiente a la fecha.

Ejercicio (17)

17a. Fecha en formato “en español”

Realizamos un script que muestre la fecha como
“Día de la semana, día del Mes del Año”
e.g. Miércoles, 15 del Abril del 2015

17b. Saludo según la hora.

Realizamos un script que muestre la hora, junto con otra a la que
añada aleatoriamente entre 0 y 12 horas, saludando en consecuencia
con esta segunda hora (Buenos Días – Tardes - Noches).

Objetivo: Familiarizarnos con el uso de los objetos Date().

A1

Objetos *RegExp*: creación

Js

Permiten el uso de las **expresiones regulares** en JS

Declaración / inicialización mediante su **función constructora**

```
var RegEx_nombre = new RegExp(" expresión regular $")
```

Declaración / inicialización **literal**

```
var RegEx_nombre = /expresión regular$/;
```

Dos formas de crear la misma expresión regular

```
var re = new RegExp("j.*n");
```

```
var re = /j.*n/;
```

Patrón: empieza por j y termina pon n, separadas ambas por cualquier carácter, 0 o más veces

Diapositiva 217

A1 JavaScript: The Definitive Guide
David Flanagan
O'Reilly, 2011
Alejandro; 03/04/2014

Objetos RegExp: propiedades

JS

```
▼ /[a-z]/ {  
  global: false  
  ignoreCase: false  
  lastIndex: 0  
  multiline: false  
  source: "[a-z]"  
  ► __proto__: /(?:)/  
  ...}
```

global: búsqueda una sola vez (false, default)

o repetida en toda la expresión (true)

ignoreCase: búsqueda sensible al caso (false, default) o no sensible (true)

multiline: búsqueda en una sola línea (false, default) o en varias (true)

lastIndex: posición en la que iniciar la búsqueda, por defecto 0.

Las 3 primeras pueden pasarse como
modificadores al crear la expresión
literal (en cualquier orden):

/<expresión>/gim

new RegExp("<expresión>", "gim");

Una vez establecidas las propiedades no se pueden modificar

Diapositiva 218

A1 JavaScript: The Definitive Guide
David Flanagan
O'Reilly, 2011
Alejandro; 03/04/2014

Objetos RegExp: métodos

Js

Únicamente dos métodos.

reciben una cadena como parámetro
buscan en ella el patrón de la expresión

test(): método booleano, que devuelve (true) si encuentra el patrón al menos 1 vez y (false) si no lo encuentra

(más parecido al método search() de los Strings)

exec(): devuelve un array con cada una de las coincidencias con el patrón de búsqueda

(equivale al método match() de los Strings)

Diapositiva 219

A1 JavaScript: The Definitive Guide
David Flanagan
O'Reilly, 2011
Alejandro; 03/04/2014

Uso de RegExp desde strings

Js

Métodos de los objetos String que admiten como parámetros objetos RegExp

Método match()

cadena.match(RegEx_nombre)

Devuelve un array con las coincidencias que ha encontrado, o un null si no hay coincidencias

Método search()

Cadena.search(RegEx_nombre)

Devuelve la posición del primer carácter de la primera coincidencia o -1 si no hay ninguna

Método replace()

cadena.replace(RegEx_nombre, cadena_nueva)

Método split()

cadena.split(RegEx_nombre) → array

Diapositiva 220

A1 JavaScript: The Definitive Guide
David Flanagan
O'Reilly, 2011
Alejandro; 03/04/2014

Expresiones regulares

Expresiones regulares secuencia de caracteres que forman un patrón de búsqueda, método por medio del cual se pueden realizar búsquedas dentro de cadenas de caracteres.



- AWK (Unix/Linux): comandos grep, sed ...
- Java, varias bibliotecas
- JavaScript soporte integrado: objetos **RegExp**
- Perl: el lenguaje que hizo crecer a las expresiones regulares
- PCRE: biblioteca de ExReg para C, C++, Visual Basic
- PHP dos tipos diferentes
- Python: soporte mediante su librería <regex>.
- .Net Framework: clases para utilizarlas

Meta caracteres

. * ? + [] () { } ^ \$ | \

. cualquier carácter excepto nueva línea

[] conjunto de caracteres válidos o no

\ caracteres especiales escapados

() múltiples elementos como grupo de captura

^ patrón al principio de la cadena evaluada

\$ patrón al final de la cadena evaluada

* lo precedente 0 o más veces

+ lo precedente 1 o más veces

? lo precedente 0 o una vez (= opcional)

{} número de veces de lo precedente

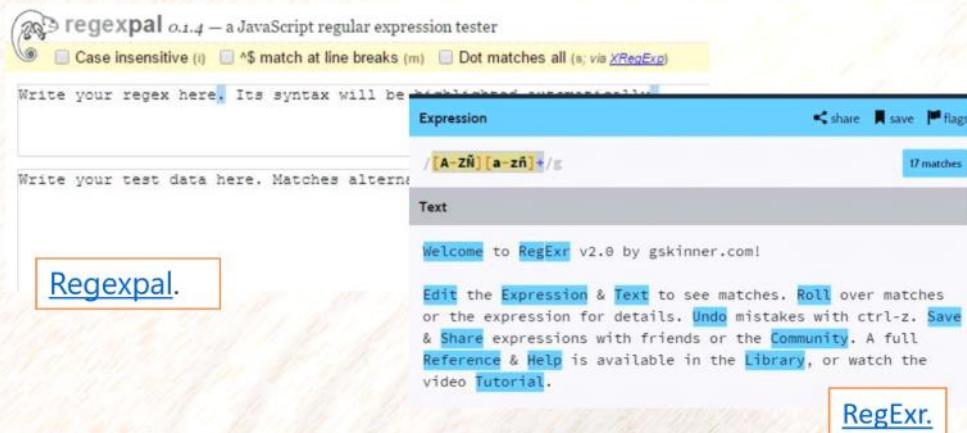
Caracteres especiales o escapados (1)

\t	tabulador.
\r	"retorno de carro" o "regreso al inicio" o sea el lugar en que la línea vuelve a iniciar.
\n	"nueva línea" el carácter por medio del cual una línea da inicio En Windows es necesaria una combinación de \r\n para comenzar una nueva línea; en Unix solamente se usa \n y en Mac_OS clásico solamente \r.
\e	tecla "Esc" o "Escape
\f	salto de página
\v	tabulador vertical
\x	caracteres ASCII
\u	caracteres unicode

Caracteres especiales o escapados (2)

\d	un dígito del 0 al 9; equivale a [0-9]
\w	cualquier carácter alfanumérico (letras del alfabeto latino y de números arábigos); equivale a [0-9 A-Z a-z]
\s	un espacio en blanco
\D	cualquier carácter que no sea un dígito del 0 al 9
\W	cualquier carácter no alfanumérico
\S	cualquier carácter que no sea un espacio en blanco
\A	Representa el inicio de la cadena. No un carácter sino una posición
\Z	Representa el final de la cadena. No un carácter sino una posición
\b	Marca el inicio y el final de una palabra
\B	Marca la posición entre dos caracteres alfanuméricos o dos no-alfanuméricos

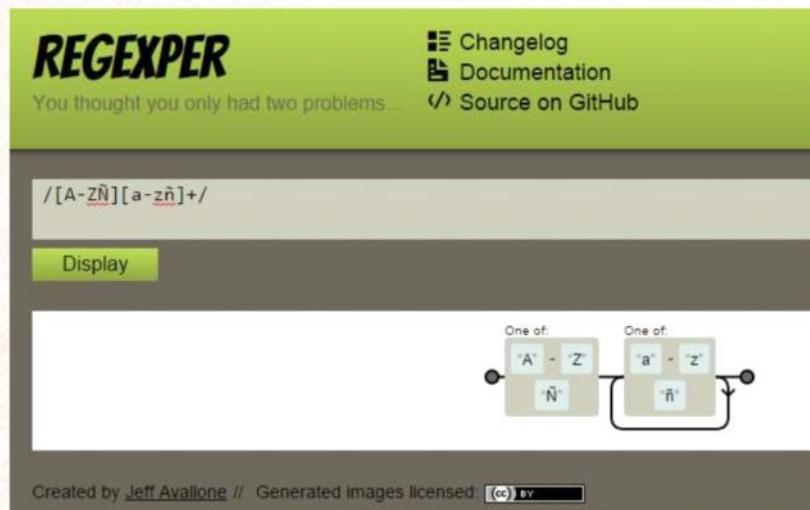
Regex: Herramientas (1)



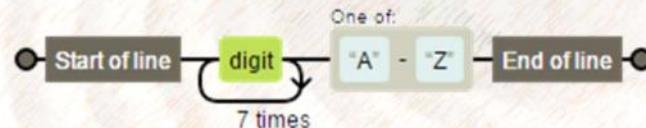
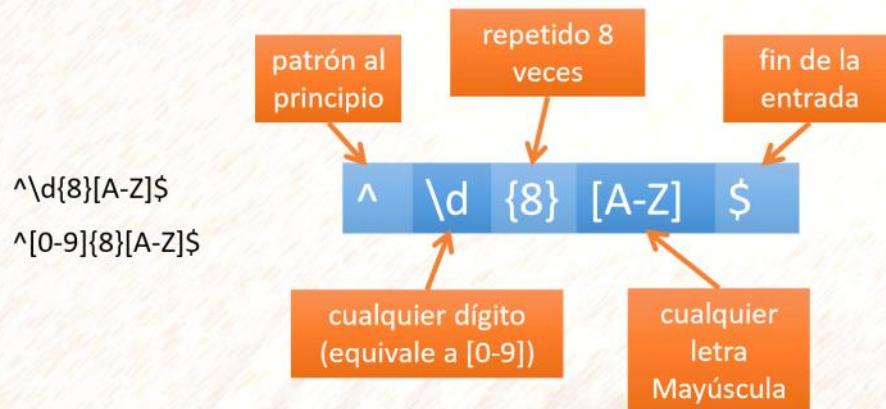
herramientas para probar expresiones regulares. Por un lado, tenemos el texto objetivo y, por otro, la expresión regex que se aplica en JavaScript

Regex: Herramientas (2)

[RegExper](#) nos permite construir el esquema que representa gráficamente el significado de la expresión regular



Expresiones regulares: DNI



Ejercicio (18a)

Replantearmos el ejercicio que comprobaba la **letra del DNI**:

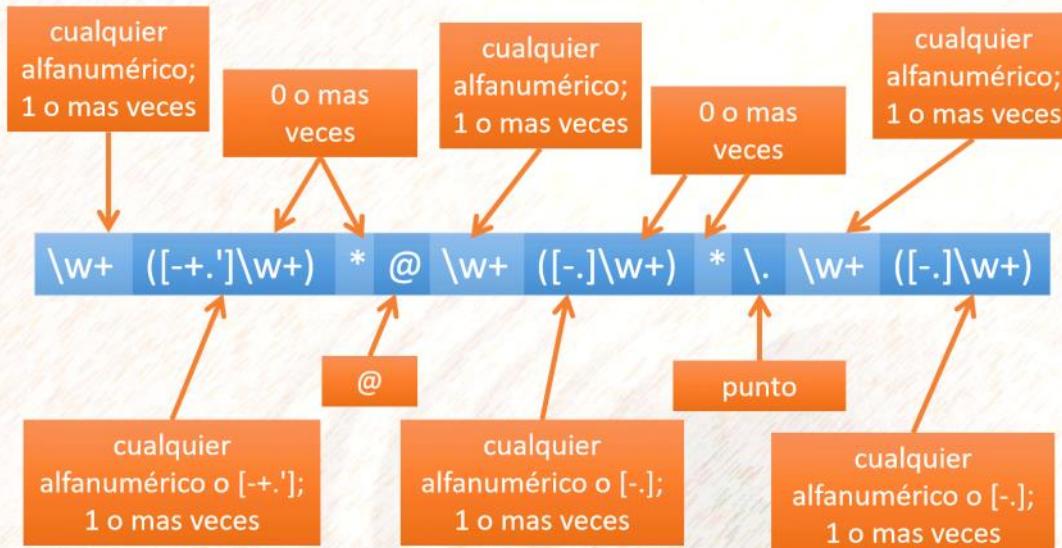
1. Solicitar del usuario su DNI y su letra, utilizando la función prompt()
2. Comprobar si el formato es valido y reducir lo a un número y una letra, empleando en ambas operaciones **expresiones regulares**
3. Comprobar si el número es válido (entre 0 y 99.999.999) y calcular la letra que le corresponde.
4. Comprobar si la letra indicada por el usuario es la correcta y comunicarle el resultado

Objetivo: Utilizar **expresiones regulares** para validar una entrada de datos

NOTA: Recordamos que el cálculo de la letra del DNI se basa en obtener el resto de la división entera del número de DNI y el número 23 y seleccionar la letra correspondiente dentro del siguiente array :

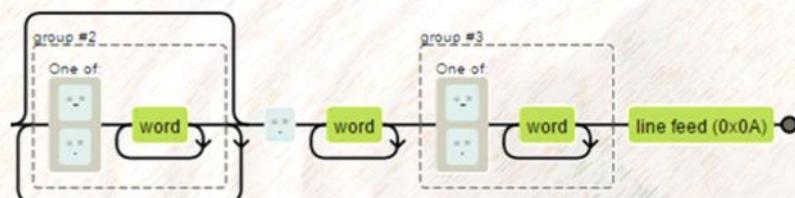
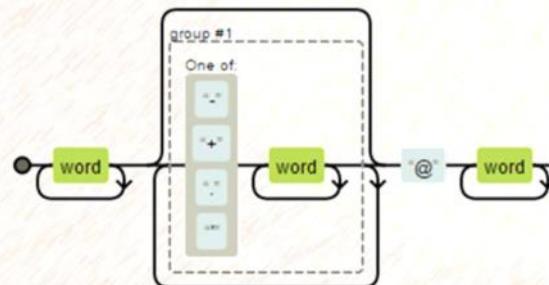
```
var letras = ['T', 'R', 'W', 'A', 'G', 'M', 'Y', 'F', 'P', 'D', 'X', 'B', 'N', 'J', 'Z', 'S', 'Q', 'V', 'H', 'L', 'C', 'K', 'E', 'T'];
```

Expresiones regulares: correo@



`\w+([-.\']\w+)*@\w+([-.\']\w+)*\.\w+([-.\']\w+)`

Expresiones regulares: correo@



\w+([-.\']\w+)*@\w+([-.\']\w+)*\.\w+([-.\']\w+)

Otras expresiones regulares

URL:	<code>^(ht f)tp(s?)\:\:\/\/[0-9a-zA-Z]([-.\\w]*[0-9a-zA-Z])*(:(0-9)*)(\/?)([a-zA-Z0-9\-\.\?\\\\'\\\+\&\%\\$\#_]*))?</code> \$
Contraseña segura	<code>(?!^[0-9]*\$)(?!^[a-zA-Z]*\$)^([a-zA-Z0-9]{8,10})\$</code>
Fecha	<code>^\d{1,2}\.\d{1,2}\.\d{2,4}\$</code>
Número tarjeta de crédito	<code>^((67\d{2}) (4\d{3}) (5[1-5]\d{2})) (6011))(-\s?\d{4}){3} (3[4,7])\ \d{2}-?\s?\d{6}-?\s?\d{5}\$</code>
Número teléfono	<code>^[0-9]{2,3}-? ?[0-9]{6,7}\$</code>
Código postal	<code>^([1-9]{2} [0-9][1-9] [1-9][0-9])[0-9]{3}\$</code>

<http://webintenta.com/validacion-con-expresiones-regulares-y-javascript.html>

Objetos Error

Js

Declaración / inicialización mediante su **función constructora**

```
var oError = new Error (num, "texto del error")
```

Realmente hay varias
funciones constructoras

EvalError, RangeError,
ReferenceError, SyntaxError,
TypeError, and URIError

Declaración / inicialización **dinámica**

```
throw "texto del error"
```

Resultado

```
▼ Error at <anonymous>:2:8 at Object.InjectedScript._evaluate0
  Object.InjectedScript.evaluate (<anonymous>:694:21) □
    stack: (...)

▶ get stack: function () { [native code] }
▶ set stack: function () { [native code] }
▶ __proto__: d
```

Diapositiva 232

A1 JavaScript: The Definitive Guide
David Flanagan
O'Reilly, 2011
Alejandro; 03/04/2014

Ejercicio (18b)

Completamos el ejercicio que comprobaba la **letra del DNI**:

1. Solicitar del usuario su DNI y su letra, utilizando la función `prompt()`
2. Comprobar si el formato es valido y reducir lo a un número y una letra, empleando en ambas operaciones **expresiones regulares**
3. Comprobar si el número es válido (entre 0 y 99.999.999) y calcular la letra que le corresponde.
4. Comprobar si la letra indicada por el usuario es la correcta y comunicarle el resultado, Cualquier caso contrario se gestiona mediante excepciones y objetos **Error()**

Objetivo: Utilizar **Errores** para gestionar las **excepciones** en la entrada de datos

NOTA: Recordamos que el cálculo de la letra del DNI se basa en obtener el resto de la división entera del número de DNI y el número 23 y seleccionar la letra correspondiente dentro del siguiente array :

```
var letras = ['T', 'R', 'W', 'A', 'G', 'M', 'Y', 'F', 'P', 'D', 'X', 'B', 'N', 'J', 'Z', 'S',  
'Q', 'V', 'H', 'L', 'C', 'K', 'E', 'T'];
```

Declaración, inicialización y modificación

Objetos

- Notación tradicional
- Notación por puntos
- Objetos literales (JSON)

Tipos primitivos
como "clases"Objetos envolventes
(Wrapper Objects)

String() → Operaciones con cadenas
Boolean()
Number()

Tipos referenciados
("clases") derivados de
Object()

Array() → Creación
Bucles For...in
Operaciones con arrays
Date() → Operaciones con fechas

Notación JSON

Js

Como hemos visto, los objetos (y arrays) pueden ser referidos como **objetos literales**, mediante el operador {}, que delimita los elementos (propiedades) que constituyen el objeto, declarados como **nombre : valor**

Esta forma de referir los objetos se conoce como **JSON** (JavaScript Object Notation), la especificación RFC 4627, y se utiliza con diversos propósitos

- formato de **representación literal** de objetos en JS
- sustrato para la **serialización** de los objetos (conversión del objeto en *string*) para poder almacenarlo
- intercambio de datos entre aplicaciones, como alternativa al formato XML. Su tipo MIME oficial es application/json.

<http://json.org/json-es.html>

Ejemplos de JSON

```
var oSocio= new Object()
```

En JSON es frecuente prescindir de la declaración.

```
var oSocio= {  
    "nombre": "Antonio",  
    "apellido": "López"  
};
```

Creación de un objeto con sólo 2 propiedades

```
oSocio.direccion= {  
    calle: "Po. Reina Victoria",  
    poblacion: "Madrid",  
    pais: "España"};
```

Añadimos un objeto como propiedad del objeto ya creado

JSON: serialización

Js

Serialización

transformación reversible de un tipo u objeto
(en memoria) en un string equivalente

La serialización es un formato de intercambio de datos

- Almacenar datos en un fichero
- Enviar datos a través de una línea de comunicación
- Paso de parámetros en interfaces REST

`JSON.stringify(object)`

En JavaScript (desde
ECMAScript 5)

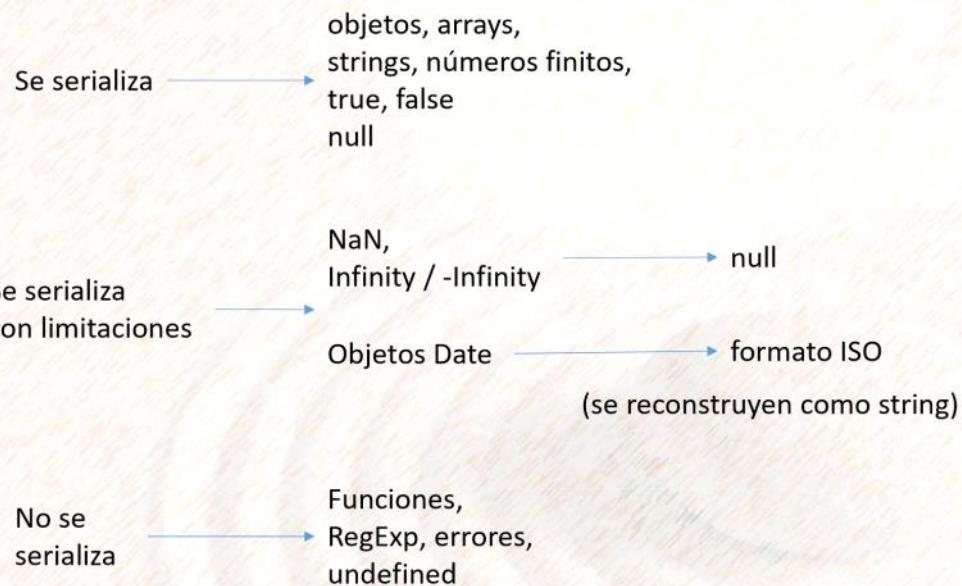
objeto JavaScript se
transforma a un string JSON

`JSON.parse(string_JSON)`

Un string JSON se transforma
en el objeto original

JSON: serialización en JS

Js



- Objetos, funciones y métodos
- Objetos definidos por el usuario: Constructores
- Prototipos. Ámbitos
- Relaciones entre clases. Herencia. Patrones

JS y el paradigma OOP

Objetos, funciones y métodos

Objetos definidos por el usuario: Constructores

Prototipos. Ámbitos

Relaciones entre clases. Herencia. Patrones

Paradigma OOP y JavaScript

Js

Hasta aquí, se ha descrito el funcionamiento de los programas en JS

- Lenguaje débilmente tipado
- Muy dependiente de variables globales
- Utiliza objetos profusamente, incluso para operar con números y cadenas

Las estructuras de programación han correspondido al paradigma tradicional de programación estructurada

Paradigma OOP en JavaScript

- Creación de objetos propios para poder modificar la estructura de los programas
- Aplicación de patrones y técnicas de diseño orientadas a objetos

OOP y JavaScript

Js

Particularidades
de la OOP
en JavaScript

- Basada en **prototipos** y con ausencia de clases
- Potente **notación literal** de objetos (de ahí el JSON)
- Operador **new** para invocar al método constructor
- **Herencia prototipada** a partir de otros objeto
- **Eventos**

Visiones del
paradigma de la OOP

- Enfoque clásico basado en **clases**
- Enfoque centrado en **prototipos**

Visiones de la OOP en JS (2)

Js

- Enfoque centrado en prototipos

- Al basarse en el uso de los **prototipos**, se adapta a la forma en que realmente trabaja en lenguaje
- Igualmente aprovecha la plasticidad de los **objetos literales** y de la **modificación dinámica** de los objetos

Se aleja de las formas de trabajo habituales en los lenguajes OOP y de los patrones desarrollados en dichos lenguajes

Visiones de la OOP en JS (1)

Js

- Enfoque clásico basado en clases
 - En el fondo sigue dependiendo del uso de los **prototipos**
 - Se enfrenta a la **carenica** de determinados mecanismos explícitos de la POO (class, constructur; extends...// publico / privado)
- ECMAScript 6 incorporará
- class
 - constructor
 - super
 - extends
 - Modules...
- alternativas como **TypeScript**, superset del lenguaje con características como
- tipado estricto
 - palabras reservadas
 - estructuras de datos como interfaces...

En resumen: Objetos en JavaScript

JS

- **Definición:** un objeto es una colección de propiedades (elementos), cada una con un nombre y un valor
- **Creación:** Los objetos se pueden crear declarándolos previamente mediante la palabra reservada **new** y el nombre de una **función constructora**, e.g. `Object()`
- **Creación:** Alternativamente es posible la declaración e inicialización **literal** de los objetos, mediante el operador `{}`, que encierra las propiedades iniciales del objeto. El resultado en ambos casos es idéntico
- **Modificación:** Una vez declarado de cualquiera de las maneras, la **asignación** de valores es **dinámica**, pudiendo realizarse en cualquier momento, modificando o añadiendo propiedades, indicadas mediante los operadores `[]` o `.` (punto).
- **Utilización:** Los mismos operadores permiten hacer **referencia** a las propiedades en cualquier momento, para hacer uso de los valores que almacenan. Disponemos para ello del bucle "for – in", para recorres sucesivamente todas las propiedades de un objeto.

Funciones y métodos

Js

Las **funciones** son una clase de objetos que tiene código ejecutable asociado a ellos. En consecuencia

Método → Función que es propiedad de un objeto

Constructor → Tipo especial de función que junto con el operador **new** inicializa un nuevo objeto.

No se definen funciones globales que operen con datos de distintos tipos

Los propios tipos de datos (simples u objetos) definen **métodos** que trabajan con los datos

Por ejemplo, para ordenar un array

no se le pasa el dato a la función	→	sort(a) // versión "estructurada"
se invoca el método del objeto	→	a.sort(); // version oop

Definición de métodos

Js

Los métodos se definen **asignando funciones al objeto**.

```
oObjeto.muestraId = muestraId
```

Para asignar una función externa como método se indica su nombre sin (). De otra forma se ejecutaría la llamada a la función y se asignaría como propiedad el resultado.

Los métodos de los objetos también se suelen definir de **forma anónima**, e.g. mediante la notación de puntos

```
oObjeto.muestraId =  
function() {alert("El ID del objeto es " + this.id);}
```

Si la función no está definida previamente, es posible crear una **función anónima** para asignarla al nuevo método del objeto.

this: Referencia
al propio objeto

Definición dinámica

Dinámicamente, las propiedades de objetos (incluidos los métodos)

- Pueden crearse
- Pueden destruirse

```
var oX new Object();
```

Operaciones sobre propiedades

- oX.c = 4
 - si propiedad oX.c existe, le asigna 4;
 - Si oX.c no existe, crea oX.c y le asigna 4
- **delete** x.c
 - si existe oX.c, la elimina; si no existe, no hace nada
- "c" **in** x
 - si oX.c existe, devuelve true, sino devuelve, false

En objetos anidados, una propiedad inexistente de un objeto anidado inexistente provocaría un error de ejecución

Ejercicio (19a)

Funciones, ámbitos y parámetros.

Recuperamos el ejercicio 10 en el que creábamos varias versiones de una sencilla que sumaba dos números.

Tomamos la opción en la que el parámetro era un objeto:

“Creamos una función que reciba como parámetro un objeto {var_1: 12, var_2: 5, resultado: 0}, sume los dos primeros atributos y almacene el resultado en el tercero”.

Reconvertimos la función como método incorporado al objeto.

Objetivo: Comprobar como se incorporan métodos a los objetos y la diferencia con las funciones independientes..

El operador *this*.

Js

sin this

```
oObjeto.muestraId =  
function() {alert("El ID del objeto es " + miObjeto.id);}
```

```
    alert(miObjetoHijo.muestraId)
```

this → Referencia al propio objeto
Su valor depende de la **forma de invocación**

```
oObjeto.muestraId =  
function() {alert("El ID del objeto es " + this.id);}
```

Es importante poder usar *this* para hacer referencia al objeto sin necesidad de conocer su nombre, independizando así el método del objeto en el que se crea

Opciones de invocación

Js

La diferencia entre funciones y métodos está en el patrón de invocación , que determina cual es el valor de this

- | | |
|-------------------------------------|---|
| Patrón de invocación
Function | <pre>var var1 = función_a (parámetros)</pre> <p>this es el objeto global del programa</p> |
| Patrón de invocación
Method | <pre>var var1 = objeto.función_a (parámetros)</pre> <p>this es el objeto en el que esta el método</p> |
| Patrón de invocación
Constructor | <pre>var var1 = new Función_a (parámetros)</pre> <p>this es el nuevo objeto creado</p> |
| Patrón de invocación
Apply | <pre>permite definir el valor de this</pre> |

Métodos *apply()* y *call()*

Js

Métodos del objeto **Function**, que Permiten ejecutar una función como si fuera un método de otro objeto: cambian el **binding** de la función ligándola al objeto, que pasa a ser **this** dentro de la función

```
function miFuncion(x) {  
    return this.numero + x;  
}  
var oObjeto = new Object();  
oObjeto.numero = 5;  
var resultado = miFuncion.call(oObjeto, 4);  
alert(resultado);
```

El primer parámetro del método es el objeto y la función se trata como si fuera un método del objeto.

Este procedimiento es necesario cuando un método se utiliza como función *callback*, e.g. al definir un manejador de objetos.

Diferencias `apply()` y `call()`

Js

La única diferencia entre los dos métodos es la forma en la que se pasan los argumentos a la función

- El método `call()` pasa independientemente cada parámetro de la función aplicada
- El método `apply()` es igual, salvo que en este caso los parámetros se pasan como un array.

```
function miFuncion(x, y) {  
    return (this.numero + x) * y;  
  
var oObjeto = new Object();  
oObjeto.numero = 5;  
var resultado_1 = miFuncion.call(oObjeto, 4, 3);  
var resultado_2 = miFuncion.apply(oObjeto, [4, 3]);  
console.log(resultado_1);  
console.log(resultado_2);
```

27
27

Métodos y patrón "Callback"



al pasar un método como argumento
a otra función (patrón Callback)

cambia el significado
del operador *this*

Inicialmente, cuando el método se ejecuta en el
objeto, *this* hace referencia a dicho objeto

En el marco de la función que ejecuta un método recibido como
parámetro, *this* hace referencia al entorno de la función, generalmente
el objeto global, window en el caso del navegador dicho objeto

El método debe ser aplicado (*apply* o *call*) con referencia al objeto original.
Las últimas versiones de JS incorporan un método *bind()* que realiza esta
operación

Ejemplo: call y funciones “callback”

```
obj1 = {  
    numero: 22,  
    calc : function (x, y) {  
        return (this.numero + x) * y;  
    }  
}  
  
function mostrar1(f) { console.log(f(3,2)); }  
function mostrar2(f) { console.log(f.call(obj1, 3, 2)); }  
  
mostrar1(obj1.calc);  
mostrar2(obj1.calc);  
mostrar1(obj1.calc.bind(obj1));
```

objeto que incluye un
método que luego
pasamos como callback

Ejercicio (19b)

Objeto Facturas:

Definir la estructura de un objeto que almacena una factura. Las facturas están formadas por la información de la propia empresa (nombre de la empresa, dirección, teléfono, NIF), la información del cliente (similar a la de la empresa), una lista de elementos (cada uno de los cuales dispone de descripción, precio, cantidad) y otra información básica de la factura (importe total, tipo de IVA, forma de pago).

Una vez definidas las propiedades del objeto, añadir un **método** que calcule el importe total de la factura y actualice el valor de la propiedad correspondiente. Por último, añadir otro método que muestre por pantalla el importe total de la factura en un formato HTML adecuado.

Objetivo: Conocer la técnica para declarar objetos e inicializar una serie de propiedades y métodos

Funciones constructoras

Js

Se pueden crear funciones constructoras propias.

```
function Factura(idFactura, idCliente) {  
    this.idFactura = idFactura;  
    this.idCliente = idCliente;  
}
```

La factura se inicializa mediante el identificador de factura y el de cliente.

```
var fac1 = new Factura(123, "A55");
```

La **aproximación clásica** en OOP se basa en la creación de funciones constructoras para representar las clases definidas por el usuario, dotarlas de sus métodos y propiedades y instanciar los objetos necesarios a partir de ellas.

Instanciación y el operador new

Js

Cuando declaramos un objeto mediante la palabra reservada **new** y el nombre de un tipo de datos o una clase personalizada, realmente estamos invocando una función que se ejecuta para crear (instanciar) el nuevo objeto, y como se trata de funciones, es posible incluir parámetros en la creación del objeto.

```
var variable = new <clase>(<parámetros>);
```

- se crea un **objeto genérico** del tipo *Object*
- se **asigna** al prototipo de ese objeto genérico la función constructora invocada mediante new
- se **ejecuta** la función constructora
- se devuelve el objeto inicialmente de tipo *Object* pero cuyo prototipo a cambiado a la clase constructora

La clave es el paso 2, que cambia el prototipo del objeto, para que apunte a una determinada función constructora, definiendo así el tipo del objeto

Ejemplo de instancia

Js

```
function Factura(idFactura, idCliente) {  
    this.idFactura = idFactura;  
    this.idCliente = idCliente;  
}
```

```
var fact1 = new Factura(123, "A55");  
console.dir(fact1)
```

```
▼ Factura [Object]  
  idCliente: "A55"  
  idFactura: 123  
  ► __proto__: Factura
```

```
▼ Factura [Object]  
  idCliente: "A55"  
  idFactura: 123  
  ► __proto__: Factura  
    ► constructor: function Factura(idFactura, idCliente) { [native code]  
    ► __proto__: Object  
      ► _defineGetter_: function _defineGetter_() { [native code]  
      ► _defineSetter_: function _defineSetter_() { [native code]  
      ► _lookupGetter_: function _lookupGetter_() { [native code]  
      ► _lookupSetter_: function _lookupSetter_() { [native code]  
      ► constructor: function Object() { [native code] }  
      ► hasOwnProperty: function hasOwnProperty() { [native code]  
      ► isPrototypeOf: function isPrototypeOf() { [native code]  
      ► propertyIsEnumerable: function propertyIsEnumerable()  
      ► toLocaleString: function toLocaleString() { [native code]  
      ► toString: function toString() { [native code] }  
      ► valueOf: function valueOf() { [native code] }  
      ► get __proto__: function __proto__() { [native code] }  
      ► set __proto__: function __proto__() { [native code] }
```

El prototipo del objeto es la función constructora Factura(), que como objeto de tipo función tiene su propio prototipo, que es la función constructora genérica, Object(). Es lo que se conoce como **cadena de prototipos (prototype chain)**

Ejemplo de instanciaión: propiedad "constructor"

Js

```
var fact1 = new Factura(123, "A55");
console.dir(fact1)
```

```
▼ Factura ⓘ
  idCliente: "A55"
  idFactura: 123
  ► __proto__: Factura
```

```
▼ Factura ⓘ
  idCliente: "A55"
  idFactura: 123
  ► __proto__: Factura
    ► constructor: function Factura(idFactura, idCliente) {
    ► __proto__: Object
```

La propiedad "constructor" del objeto toma como valor la función constructora, en este caso Factura().

Eso hace que se pueda decir que el objeto pertenece a la "clase" Factura

La propiedad constructor puede usarse en la instanciaión de un nuevo objeto: fact2 = new fact1.constructor(124, "B64")

operador instanceof

Js

Es un operador (escrito todo con minúsculas) que compara un objeto con una posible función constructora y devuelve verdadero o falso

```
var oObj = new Object();
var fact = new Factura(123, "A55");
```

```
oObj instanceof Object; → true
```

```
oObj instanceof Factura; → false
```

```
fact instanceof Object; → true
```

```
fact instanceof Factura; → true
```

Funciones constructoras: incorporación de métodos

Js

Se pueden incorporar métodos en las funciones constructoras propias que creamos.

```
function Factura(idFactura, idCliente) {  
    this.idFactura = idFactura;  
    this.idCliente = idCliente;  
  
    this.muestraCliente = function() {  
        alert(this.idCliente);  
    }  
    this.muestraId = function() {  
        alert(this.idFactura);  
    }  
}
```

El inconveniente es que al crear objetos con este constructor, los métodos se crearan de nuevo por cada objeto creado..

Ejercicio (2o)

Catálogo de productos (varios objetos).

Crear un objeto usado para representar un artículo de una tienda. El artículo se va a caracterizar por una descripción, un código y un precio, y debe permitir el cálculo de su correspondiente IVA y ser capaz de mostrar los datos por pantalla (2 métodos).

Detalles: en lugar de incorporar las funciones directamente en el objeto, se incorporaran en la **función constructora**.

Objetivo: Conocer la técnica para declarar objetos e inicializar una serie de propiedades y métodos

Constructores y redundancia

Js

Cada instancia en la que invocamos un constructor copia en el nuevo objeto los métodos que contiene dicho constructor.

```
function Factura(idFactura, idCliente) {  
    this.idFactura = idFactura;  
    this.idCliente = idCliente;  
    this.muestraCliente = ...  
    this.muestraId = ...  
  
    var fa_1 new Factura(idFactura001, idCliente001)
```

El objeto fa_1 incluye el código de los métodos muestraCliente y muestralId

Prototipo

Js

Todos los objetos tienen una propiedad denominada prototipo que apunta al **objeto (función) a partir del que han sido creados:**

- el objeto genérico *Object*.
- alguno de los objetos predefinidos, *Array*, *Date* ...
- un objeto de igual nombre, en caso de una función
- un objeto definido por el usuario

Recordamos: al
crear un objeto

- se crea un objeto genérico del tipo *Object*
- se asigna al prototipo de ese objeto genérico la función constructora invocada (mediante *new* o literalmente)

Como cualquier objeto, el prototipo puede modificarse dinámicamente

Constructores y prototipos.

JS

Para evitar ese problema, podemos reescribir la función constructora utilizando el objeto prototype:

```
function Nombre() {  
    this.propiedad = <valor>;  
    this.metodo = function() {...;}  
  
    Nombre.prototype.propiedad = <valor>;  
    Nombre.prototype.metodo = function() {...}  
}
```

La propiedad prototype del objeto permite incluir propiedades y métodos en el prototipo de un objeto

La diferencia entre propiedades / métodos del prototipo y los que no lo son es que los primeros se transfieren por referencia y los segundos por valor.

Constructores y prototipos.

Ejemplo (1)

Js

Podemos reescribir la función constructora del ejemplo anterior, Factura(), utilizando el objeto prototype:

```
function Factura(idFactura, idCliente) {  
    this.idFactura = idFactura;  
    this.idCliente = idCliente;  
}  
Factura.prototype.muestraCliente = function() {  
    alert(this.idCliente);  
}  
Factura.prototype.muestraId = function() {  
    alert(this.idFactura);  
}
```

La propiedad prototype del objeto permite incluir los métodos en el prototipo de un objeto para que no se repitan en cada instancia

Constructores y prototipos.

Ejemplo (2)

Js

En lugar de modificar el prototipo, podemos crear uno completamente nuevo:

```
function Factura(idFactura, idCliente) {  
    this.idFactura = idFactura;  
    this.idCliente = idCliente;  
}  
Factura.prototype = {  
    muestraCliente : function() {alert(this.idCliente)},  
    muestraId : function() {alert(this.idFactura)}  
}
```

Se crea un objeto *prototype* completamente nuevo, que sustituye al anterior

¿Qué incluimos como prototipo?

JS

en el prototype de un objeto sólo se deben añadir aquellos elementos comunes para todos los objetos

los métodos

las constantes (propiedades cuyo valor no varía durante la ejecución de la aplicación)

permanecen en la función constructora sin vincular al prototipo

las propiedades del objeto para que cada objeto diferente pueda tener un valor distinto en esas propiedades.

Acceso al prototipos

JS

```
fact = new Factura(idF, idC)  
  
fac.idFactura;  
fac.idCliente;  
fac.muestraCliente();  
fac.muestraId();
```

Todas las propiedades y métodos del objeto se acceden de igual manera, independientemente de que se encuentren en el prototipo
Además, todas son públicas, ya que inicialmente no existe ningún ámbito de visibilidad privado.

Si se hace referencia a una propiedad que no existe como propia, se busca sucesivamente en la **cadena de prototipos** del objeto.

Una propiedad propia con igual nombre que una del prototipo **oculta** completamente esta última

Cambios en el prototipo

Js

Las propiedades incluidas en el prototipo se pueden modificar **dinámicamente**, como las de cualquier objeto.

El prototipo es un elemento **vivo**

Cualquier cambio en un prototipo se refleja en todos los elementos que lo usan, es decir en **todas sus instancias** (objetos creados a partir de él) incluso en las realizadas previamente

Esto se debe a que la propiedad prototipo contiene una referencia al objeto correspondiente y por tanto refleja siempre cualquier cambio en el objeto

Ejercicio (21)

Catálogo de productos (varios objetos).

Recuperamos el ejercicio en el que creábamos objetos para representar los artículos de una tienda. (El artículo se va a caracterizar por una descripción, un código y un precio, y debe permitir el cálculo de su correspondiente IVA.)

En lugar de incorporar las funciones directamente en la función constructora los incorporamos en su **prototipo**.

Objetivo: Conocer la técnica para declarar objetos e inicializar una serie de propiedades y métodos

Prototipos y "clases"

Js

Al crear una función constructora en la que se definen una serie de métodos con carácter de prototipos, podemos considerar que tenemos una **pseudoclase** por su similitud al uso del término en el modelo "clásico" de OOP

los objetos creados con esta función constructora incluyen por defecto los métodos definidos en la función

no se crean nuevas funciones por cada objeto, sino que los métodos se definen únicamente una vez y se pasan por [referencia](#) a todos los objetos creados

en consecuencia, cualquier modificación en un método en la función constructora definido como prototipo se traslada automáticamente a los objetos creados a partir de ella

En resumen: Objetos y prototipos

JS

Todos los objetos de JavaScript incluyen una referencia interna a otro objeto llamado prototype o "prototipo".

Cualquier propiedad o método que contenga el objeto prototipo, está presente de forma automática en el objeto original.

- Es como si cualquier objeto heredara de forma automática todas las propiedades y métodos de otro objeto llamado prototype.
- Cada tipo de objeto diferente hereda de un objeto prototype diferente
- Los métodos comunes a los objetos de un mismo tipo, se añaden directamente al prototipo a partir del cual se crean los objetos.

La capacidad de enlazar prototipos permite que un objeto herede propiedades de otro

Cambios en prototipos de objetos predefinidos

Js

Como sabemos, las propiedades incluidas en el prototipo se pueden modificar **dinámicamente**, reflejándose los cambios en todos los objetos que lo utilizan como tal (**instancias** a partir de él)

Esto es igual de válido para los **objetos predefinidos por JavaScript**.

La propiedad *prototype* permite también añadir y/o modificar las propiedades y métodos de estos objetos

- Es posible **redefinir** el comportamiento habitual de algunos métodos de los objetos nativos de JavaScript.
- Se pueden **añadir** propiedades o métodos completamente nuevos.

Ejemplo: cambios en el prototipo de objetos Array

Js

Clase Array, esta no dispone de métodos que

- indique la posición de un elemento dentro de un array - como la función indexOf() de Java)
- compruebe que un valor existe en el array – como la función in_array() de PHP

```
Array.prototype.inArray = function(needle) {  
    for (var i = 0, len = this.length; i < len; i++) {  
        if (this[i] === needle) {  
            return true;  
        }  
    }  
    return false;  
}
```

Modificando el prototipo con el que se construyen los objetos de tipo Array, es posible añadir estas funcionalidades.

Uso de la técnica en librerías

JS

Una de las características habituales de las librerías que complementan JavaScript que luego veremos, es el uso de la propiedad `prototype` para mejorar las funcionalidades básicas del lenguaje.

La librería **Prototype** emplea precisamente esta técnica

<http://prototypejs.org/>

prototype

```
Object.extend(Array.prototype,
{
  _each: function(iterator) {
    for (var i = 0; i < this.length; i++)
      iterator(this[i]);
  },
  clear: function() {
    this.length = 0;
    return this;
  },
  first: function() {
    return this[0];
  },
  last: function() {
    return this[this.length - 1];
  },
  ...
}
```

7.Prototype

martes, 17 de julio de 2018 22:06

```
var Empleado = function(nombre, sueldo) {
  this.nombre = nombre;
  this.sueldo = sueldo;
};
```

Al crear una función sus propiedades incluyen
- `prototype` que apunta a un determinado objeto

Al usar la función como constructor creamos un objeto
que a su vez incluye las propiedades:

- `constructor`, que apunta a la función constructora
- `__proto__` que hace referencia al prototipo del
objeto, proporcionado a su vez por la función
constructora

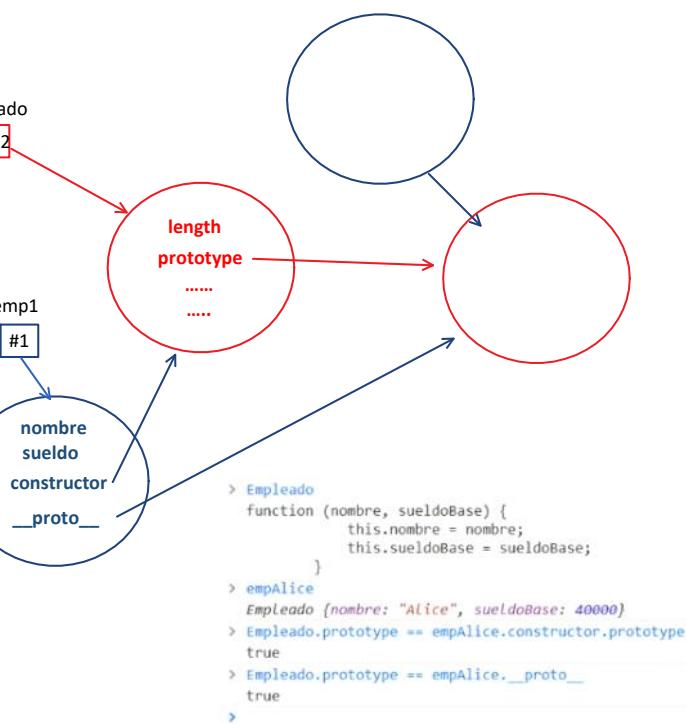
Al prototipo de la función constructora puedo
añadirle propiedades

```
Empleado.prototype.empresa = "Boracay"
```

Si se consulta la propiedad en el objeto

```
emp1.empresa // "Boracay"
```

se recupera el valor existente en su propiedad
`__proto__` que corresponde al prototype de la
función constructora.

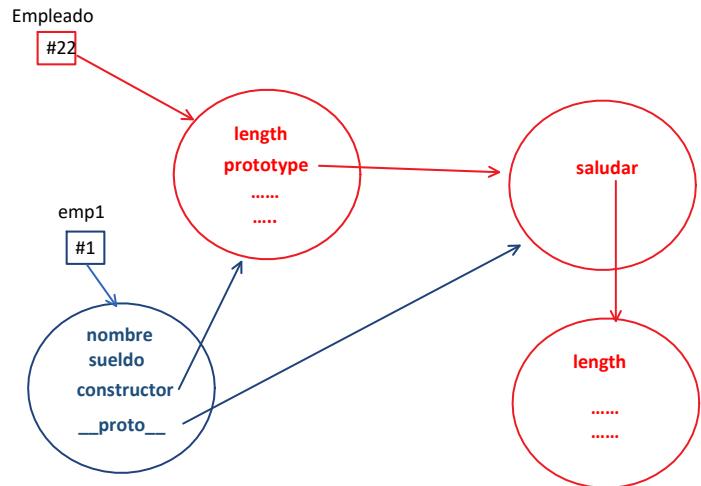


En caso de no encontrarla, se seguiría toda la cadena
de prototipos, de acuerdo con el principio de
delegación, mal llamado "herencia prototípica"

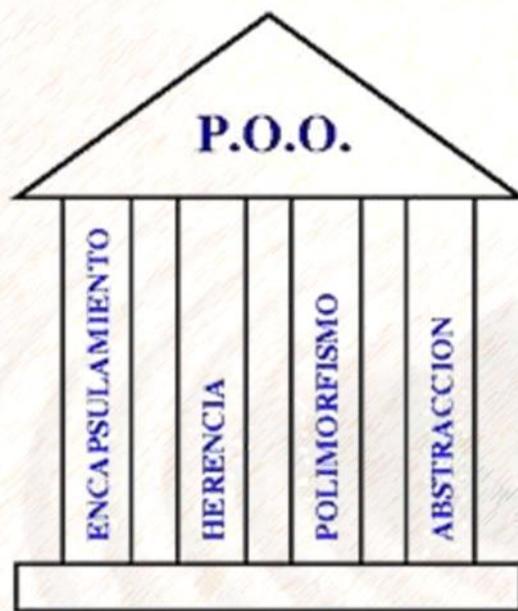
Esto es útil si se hace con los métodos

```
Empleado.prototype.saludar = function() {
  console.log('Hola, soy ' + this.nombre);
}
```

El prototipo se convierte en el responsable
del comportamiento básico de los objetos
vinculados a él. EN este caso el
comportamiento de todos los empleados.



Orientación a objetos



Abstracción: clases y objetos



CLASE



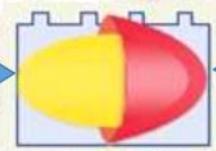
Entidad teórica (abstracta) que describe el comportamiento de los objetos

Atributos -> posibles datos
Métodos ---> servicios que proporciona

OBJETOS

Atributos:
datos con un valor determinado

Fragmento de código



Métodos:
servicios que proporciona

INSTANCIACION
(en tiempo de ejecución)

Objetos y encapsulación



La idea fundamental de la Programación Orientada a Objetos es **encapsular** en una misma entidad (un objeto) una serie de datos que definan su estado (variables de instancia) y las funciones encargadas de acceder a esos datos (métodos.). De esta forma se reúne al **mismo nivel de abstracción**, a todos los elementos que puedan considerarse pertenecientes a una misma entidad.

Cada objeto de una determinada clase funciona como una **caja negra** de la que poco nos importa su implementación interna. Cuando trabajamos con un objeto de la clase “coche” nos da igual cómo funcione internamente el método “acelerar”. Tan sólo sabemos que nuestro coche puede acelerar, y sabemos cuál será el resultado, sin preocuparnos del proceso interno que se siga para conseguir este fin.

De esta forma **se independiza la implementación frente al uso de sus objetos**, y el flujo del programa pasa de ser procedural a convertirse en un diálogo entre objetos que interactúan entre sí.

Ocultación y ámbitos de visibilidad

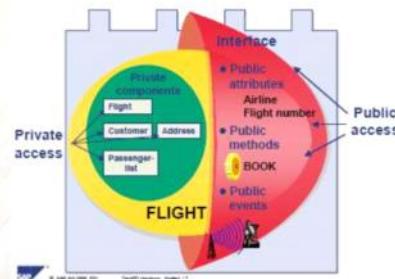


Ámbitos:

- Público
- Privado

Atributos privados:
visibles solo para los
métodos

Métodos públicos:
interrelacionan atributos
privados
con otros objetos



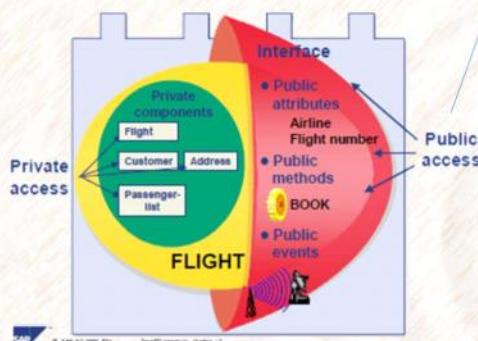
Interfaces

Clases: modificadores de acceso (1)



Ámbitos:

- Público
- Privado



Público

Es el nivel de acceso más permisivo. Es el modificador que se aplica si no se indica otra cosa.

Un método o atributo público puede ser accedido para visualizarlo, editarlo o invocarlo, por cualquier otro elemento de nuestro programa

Para una correcta ocultación en los objetos encapsulados, sólamente serán **públicos** los **métodos** que manipulen los atributos, constituyendo la interfaz del objeto y en menor medida los atributos que sea imprescindible situar en este ámbito.

Clases: modificadores de acceso (2)



Privado

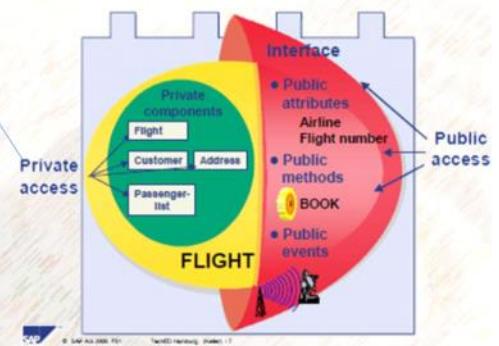
Es el nivel de acceso más restrictivo.

Un método o atributo privado NO puede ser accedido para visualizarlo, editarlo o invocarlo, por cualquier otro elemento de nuestro programa

Para una correcta ocultación en los objetos encapsulados, los **atributos** serán **privados** a no ser que sea imprescindible por algún motivo situarlos fuera de este ámbito, junto con los métodos que no constituyen la interfaz del objeto

Ámbitos:

- Público
- Privado



Ámbitos de existencia



En base a la existencia de las clases, se distinguen diferentes ámbitos para sus distintos componentes de la clase (atributos, métodos...)

- dependientes de la instancia → existen separadamente para cada objeto de una clase
- independientes de la instancia (STATIC) → existen sólo una vez para la clase entera, independientemente del número de instancias.

Enfoque clásico de la OOP. Propiedades y métodos

Js

Instanciación → siempre a partir de la función constructora y el operador new: asimilado a la existencia de clases

Propiedades → Estáticas
 Públicas
 Privadas

Métodos → Estáticos
 Públicos
 Privados
 Privilegiados

Ámbitos de visibilidad

Js

En JS todas las propiedades y métodos de un objeto son inicialmente **públicos**

- **No existe ámbito privado**
- No existen palabras reservadas como
 - *public*,
 - *private* o
 - *protected*

Existen **patrones** de programación que producen un comportamiento de las propiedades y los métodos **muy similar** al que se derriba de un ámbito de visibilidad **privado**



Ejercicio (22a)

Función constructora de objetos Triángulo:

Crear la función Triangulo, con tres propiedades propias, lado1, lado2 y lado3, y tres métodos, incluidos en el prototipo, getLados, getPerímetro y getTipo, que devuelve equilátero, isósceles o escaleno.

Crear 3 objetos, un triangulo de cada tipo, y presentar en pantalla la información proporcionada por cada método.

Nota: Los "atributos se denominaran empezando por "_" y se tratarán como si fueran privados

Objetivo: Conocer la técnica para crear funciones constructoras empleando los prototipos de los objetos para incluir una serie de propiedades y métodos generales.



Ámbito público

Js

```
function NuevoTipo () {
```

```
    this.propiedad = <valor>;  
    this.método = function () {  
        < código >};  
}
```

```
NuevoTipo.prototype.propiedad = <valor>;  
NuevoTipo.prototype.método = function () {  
    < código >};
```

Elementos de
instancia

principalmente
propiedades

Elementos del
prototipo

principalmente
métodos

Al aplicar **patrones** de programación que generen un ámbito de visibilidad **privado**, el comportamiento de los métodos será diferente según sean de instancia o del prototipo

Patrones: propiedades privadas (1)



Las variables locales a la función constructora actúan como se fueran propiedades privadas de cada una de las instancias de dicha función

```
function NuevoTipo () {  
    var propiedad = <valor>;  
    this.propiedad = <valor>;  
}  
  
NuevoTipo.prototype.método = function () {  
    < código >;  
}
```

"Propiedades
privadas"

Patrones: propiedades privadas (2)



Las variables locales a la función constructora actúan como se fueran propiedades privadas de cada una de las instancias de dicha función

```
function NuevoTipo () {  
    var propiedad = <valor>;  
    this.propiedad = <valor>;  
}  
  
NuevoTipo.prototype.método = function () {  
    < código >;  
}
```

Los **métodos del prototipo** ("públicos") no tienen acceso a los elementos privados (realmente locales al interior de la función constructora)

Usar en su lugar **métodos de instancia** lo solucionaría pero implicaría la redundancia de código ya conocida

Patrones: métodos privilegiados (1)



Las métodos públicos de instancia se denominan **privilegiados** porque SI tienen acceso a las propiedades privadas.

```
function NuevoTipo () {  
    var propiedad = <valor>;  
  
    this.propiedad = <valor>;  
    this.método = function () {  
        < código >;  
    }  
  
    NuevoTipo.prototype.método = function () {  
        < código >;  
    };  
}
```

Los **métodos de la instancia** ("privilegiados") actúan de interface entre los elementos privados y el prototipo

Suelen ser métodos simple **getters** y **setters**, haciendo mínima la inevitable redundancia de código

Patrones: *getters y setters*



Cuando se definen variables privadas, los métodos privilegiados suelen ser *getters* y *setters*

- son muy simples, por lo que no penaliza apenas el que vayan en el objeto en lugar de en el prototipo
- forman el interfaz entre las variables privadas y el exterior
- permiten distinguir distintos accesos a estas: lectura, escritura, ambos o ninguno

```
var propiedad
```

```
this.setPropiedad(propiedad) {this.propiedad = propiedad}  
this.getPrpiedad() {return this.propiedad}
```

Resumen: Patrones de visibilidad



Privado: su definición como variables locales permite que las propiedades y los métodos tengan un comportamiento muy similar al que se derriba de un ámbito de visibilidad privado

```
function Clase() {  
    var _propiedad_privada = <valor>  
    var _método_privado = function () {...}  
  
    this.propiedad_publica = <valor>  
    this.método_publico = function () {...}  
}  
Clase.prototype.propiedad_publica = <valor>  
Clase.prototype.método_publico = function () {...}
```

Privilegiado: Son todos los que se definen dentro de la función constructora, gracias a lo cual pueden acceder a los elementos privados (variables locales)

Público: El último sería un método público pero no privilegiado: sin acceso a las variables locales

Ejercicio (22b)

Función constructora de objetos Triángulo:

Recuperamos la función Triangulo, con tres propiedades propias, lado1, lado2 y lado3, que en este caso definimos en el ámbito **privado**, con sus correspondientes métodos get **privilegiados**.

En el prototipo, incluimos los métodos **públicos** showLados, calcPerímetro y calcTipo, que devuelve "equilátero", "isósceles" o "escaleno".

Creamos 3 objetos, un triangulo de cada tipo, y presentamos en pantalla la información proporcionada por cada método.

Objetivo: Conocer la patrones de desarrollo empleados para generar elementos privados, privilegiados y públicos dentro de los objetos de JavaScript.



Ejercicio (22c)

Catálogo de productos (varios objetos).

Recuperamos el ejercicio en el que creábamos objetos para representar los artículos de una tienda incorporando las funciones en el **prototipo**.

(El artículo se va a caracterizar por una descripción, un código y un precio, y debe permitir el cálculo de su correspondiente IVA.)

Lo reescribimos incorporando los elementos adecuados a los distintos ámbitos de visibilidad

Objetivo: Conocer la patrones de desarrollo empleados para generar elementos privados, privilegiados y públicos dentro de los objetos de JavaScript.



Sesión 34

- ▶ Prototipos.
 - ▶ Constructores y redundancia. Uso de prototipos
 - ▶ Accesos y cambios en el prototipo
 - ▶ Modificaciones en los tipos de objetos predefinidos
- ▶ Conceptos básicos en OOP clásica.
 - ▶ Ámbitos de visibilidad y existencia
- ▶ Patrones para implementarlo en JS
 - ▶ Ámbitos: público v. privado
 - ▶ Ámbito privilegiado: *getters* y *setters*
- ▶ Patrones y ámbitos en objetos literales: *closures*

Patrones y literales: ámbito privado (1)



En el marco de los objetos literales, el ámbito viene dado por el uso de cierres (*closures*)

```
var myobj; // el objeto
(function () {
    // var propiedad = <valor>;
    myobj = {
        // métodos privilegiados
        metodo: function () {...}
    }; // Fin del objeto literal
}());
```

propiedades privadas

implementación de la parte
publica/privilegiada asignada a
una variable externa a la función

Recuperamos la idea de *closure*, pero su interfaz no es una función, sino un objeto

Patrones y literales: ámbito privado (2)



Como vimos al hablar de funciones y cierres (*closures*), existe una segunda forma de implementación

```
// el objeto
var myobj = (function () {
    // var propiedad = <valor>;
    return {
        // métodos privilegiados
        metodo: function () {...}
    }; // Fin del objeto literal
}());
```

propiedades privadas

implementación de la parte
publica/privilegiada devuelta
(return) para asignarsela a una
variable externa a la función

Ejercicio (23)

Objeto literal facturas (*closure*):

Recuperamos el ejercicio que definir literalmente un objeto que almacena una factura (19b). Todas las **propiedades** se redefinen como **privadas**.

Las facturas están formadas por la información de la propia empresa (nombre de la empresa, dirección, teléfono, NIF), la información del cliente (similar a la de la empresa), una lista de elementos (cada uno de los cuales dispone de descripción, precio, cantidad) y otra información básica de la factura (importe total, tipo de IVA, forma de pago).

Se incluyen 2 **métodos públicos** 1) calcula el importe total de la factura y actualice el valor de la propiedad correspondiente. 2) muestre por pantalla el importe total de la factura en un formato HTML adecuado.

Opción: el primero de los métodos puede redefinirse como privado.

Objetivo: Conocer la técnica para declarar literalmente objetos e inicializar una serie de propiedades y métodos distribuyéndolos en ámbitos privados y públicos

Enfoque clásico de la OOP. Propiedades y métodos

Js

Instanciación → siempre a partir de la función constructora y el operador new: asimilado a la existencia de clases

Propiedades →

- Estáticas
- Públicas
- Privadas

Métodos →

- Estáticos
- Públicos
- Privados
- Privilegiados

Elementos estáticos



Elementos estáticos

- existen a nivel de la Función constructora
- no se reflejan en sus instancias
- sólo son accesibles a través de la función constructora

```
function NuevoTipo () {  
    ...  
}  
  
NuevoTipo.propiedad = <valor>;  
NuevoTipo.método = function () {  
    < código >};
```

Se definen directamente
a nivel de la función
constructora, no de su
prototipo

Se utilizan invocando directamente la función constructora,
ya que no existen a nivel de sus instancias



Ejercicio (24)

Función constructora de objetos Triángulo:

Recuperamos la función Triangulo, con tres propiedades propias, lado1, lado2 y lado3, definidas en el ámbito privado, con sus correspondientes métodos *get* privilegiados, y los métodos públicos del prototipo showLados, calcPerímetro y calcTipo, que devuelve el tipo de triangulo. Los valores para esto último, "equilátero", "isósceles" o "escaleno", lo definimos como un **array estático** a la función constructora Triangulo. Creamos 3 objetos, un triangulo de cada tipo, y presentamos en pantalla la información proporcionada por cada método.

Objetivo: Conocer la patrones de desarrollo empleados para generar **elementos estáticos** dentro de los objetos de JavaScript.



Relaciones entre clases



- Relación de Dependencia
- Relación de Asociación
 - Relación de Agregación
 - Relación de Composición
- Relación de Herencia

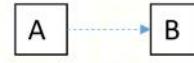


Hacen posible la reutilización de código

Relaciones entre clases: Dependencia y Asociación



- Relación de Dependencia
- Relación de Asociación
 - Relación de Agregación
 - Relación de Composición



En ambos
casos

- La clase A depende de la clase B
- La clase A usa / está asociada a la clase B
- La clase A conoce la existencia de la clase B, pero la clase B no conoce la existencia de la clase A (sentido de la flecha)
- Todo cambio que se haga en la clase B, por la relación que hay con la clase A, podrá afectar a la clase A.



Relación de dependencia

- es una relación de uso entre dos entidades” (una usa a la otra)
- una clase depende de la funcionalidad que ofrece otra clase.
- desde un punto de vista “Cliente/Servidor”, una clase es “cliente”, necesita de un “servicio”, que es proporcionado por otra clase



Codificación

1. En un método de la clase A instancio un objeto de tipo B y posteriormente lo uso.
2. En un método de la clase A recibo por parámetro un objeto de tipo B y posteriormente lo uso.

Diapositiva 306

AC31 Programación Orientada a Objetos en PHP5
Enrique Place
OpenLibra, 2009
Alejandro Cerezo; 19/04/2014

Dependencia: ejemplo



Conductor ————— depende de ————— → Vehículo
utiliza

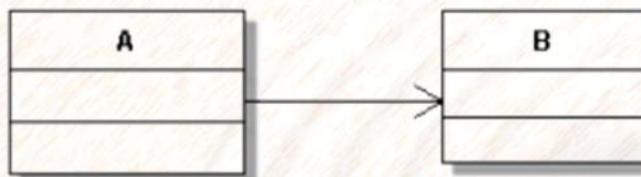
Depende de: no se puede conducir sin “tener” un vehículo

En los métodos
de conductor
(e.g. conducir) ————— Se instancia
Se recibe como
parámetro ————— → Vehículo



Relación de asociación

- es una relación estructural entre entidades" (una entidad se construye a partir de otras entidades)
- una clase tiene en su estructura a otra clase, o se puede decir también que se construye una clase a partir de otros elementos u objetos



Codificación

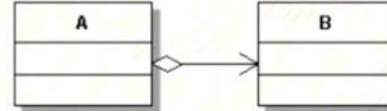
la clase A tiene como atributo un objeto de tipo clase B.
(como el atributo "b" se puede inferir de la relación no se representa en la clase A)

Asociación: variaciones



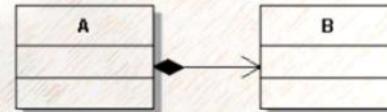
Relación de Agregación

Es una relación de asociación pero en vez de ser “1 a 1” es de “1 a muchos”, la clase A agrega muchos elementos de tipo clase B



Relación de Composición

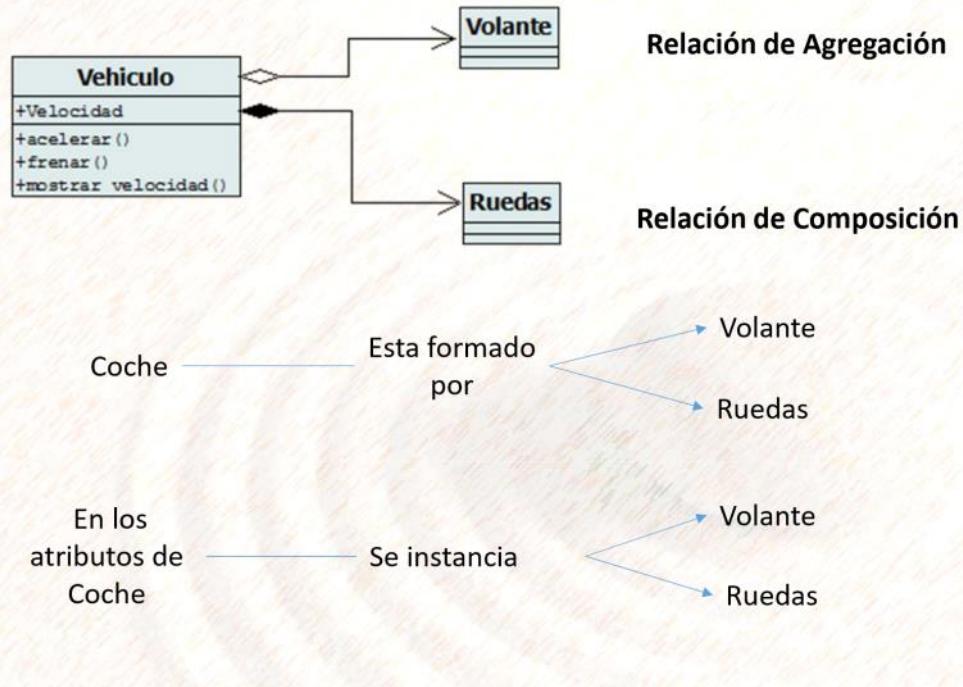
solo aporta semántica a la relación al decir que además de “agregar”, existe una relación de vida, donde elementos de B no pueden existir sin la relación con A.



Suponen la aparición de un array de objetos B en la clase A

AC33 Programación Orientada a Objetos en PHP5
Enrique Place
OpenLibra, 2009
Alejandro Cerezo; 19/04/2014

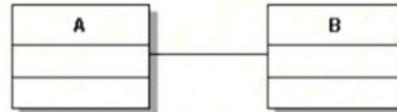
Asociación: ejemplo





Relaciones a evitar

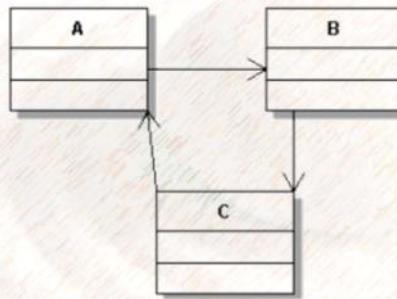
Relaciones bidireccionales



la visibilidad es en ambos sentidos
y un cambio en una clase genera
un impacto en la otra y viceversa,

Relaciones cíclicas

un cambio en A afecta a B, como
afecta a B, también afecta a C, que
a su vez afecta a A...



Diapositiva 311

AC34 Programación Orientada a Objetos en PHP5
Enrique Place
OpenLibra, 2009
Alejandro Cerezo; 19/04/2014

Asociación y dependencia de objetos en JS

Js

Asociación

(Agregación /
Composición)

- una **propiedad** de un objeto A tiene como valor un **objeto de tipo B**.

{... propiedad: {ObjetoB}}

Dependencia

- un **método** de un objeto A instancia un **objeto de tipo B** y posteriormente lo uso.
- En un **método** de un objeto A recibo por parámetro un **objeto de tipo B** y posteriormente lo uso.

{... método : function() {new ObjetoB}}

Asociación de objetos en JS

Asociación
(Agregación /
Composición)

- una **propiedad** de un objeto A tiene como valor un **objeto de tipo B**.
- {... propiedad: {ObjetoB}}

Relación muy habitual, incluso en objetos literales, dado que por definición una propiedad de un objeto puede tener como valor otro objeto cualquiera

Relación de Agregación

el objeto A agrega muchos objetos de tipo B, pero estos pueden existir también al margen de la agregación

Relación de Composición

solo aporta semántica a la relación: existe una relación de vida, donde elementos de B no pueden existir sin la relación con A.

Ejercicio (25)

Catálogo de productos (varios objetos).

Recuperamos el ejercicio en el que creábamos objetos para representar los artículos de una tienda incorporando los métodos directamente en el prototipo (21a).

Añadimos una nueva propiedad, correspondiente a los **datos del fabricante**, que se implementara como un **objeto** dentro del producto, reflejando la relación de **asociación entre los objetos**.

- 25a: Fabricante se construye literalmente en los parámetros que recibe el constructor
- 25b: Fabricante tiene su propia función constructora
- 25c: La función Fabricante se incorpora como método a Artículo

Objetivo: Conocer la técnica para declarar objetos e inicializar una serie de propiedades y métodos, estableciendo entre los objetos relaciones de asociación A->B. Ver como un mismo objeto B puede estar asociado con varios objetos A

Dependencia de objetos en JS

Js

- un **método** de un objeto A instancia un **objeto de tipo B** y posteriormente lo uso.
- En un **método** de un objeto A recibo por parámetro un **objeto de tipo B** y posteriormente lo uso.

```
{... método : function() {new ObjetoB}}
```

Ejemplo: Después de instanciar un objeto A, le añadimos el objeto B, creado gracias a un métodos del objeto A con el que establecen por tanto una relación de dependencia

```
fucntion A () {}  
// contructora de A  
A.prototuye.B = function () {}  
// constructora de B  
  
var a = new A()  
a.b = new B()
```



Ejercicio (26)

Pseudoclase Facturas: Modificar el ejercicio anterior del objeto Factura (19b) para crear una **función constructora** llamada **Factura** y que permita crear objetos de ese tipo. Se deben utilizar la propiedad prototype siempre que sea posible.

Después de instanciar un objeto Factura, le añadimos el objeto cliente y el array de objetos elementos, creando estos nuevos objetos gracias a sus propias funciones constructoras métodos que son parte del objeto factura, con el que establecen por tanto una **relación de dependencia**.
(La relación entre los objetos se da a nivel de las funciones constructoras)

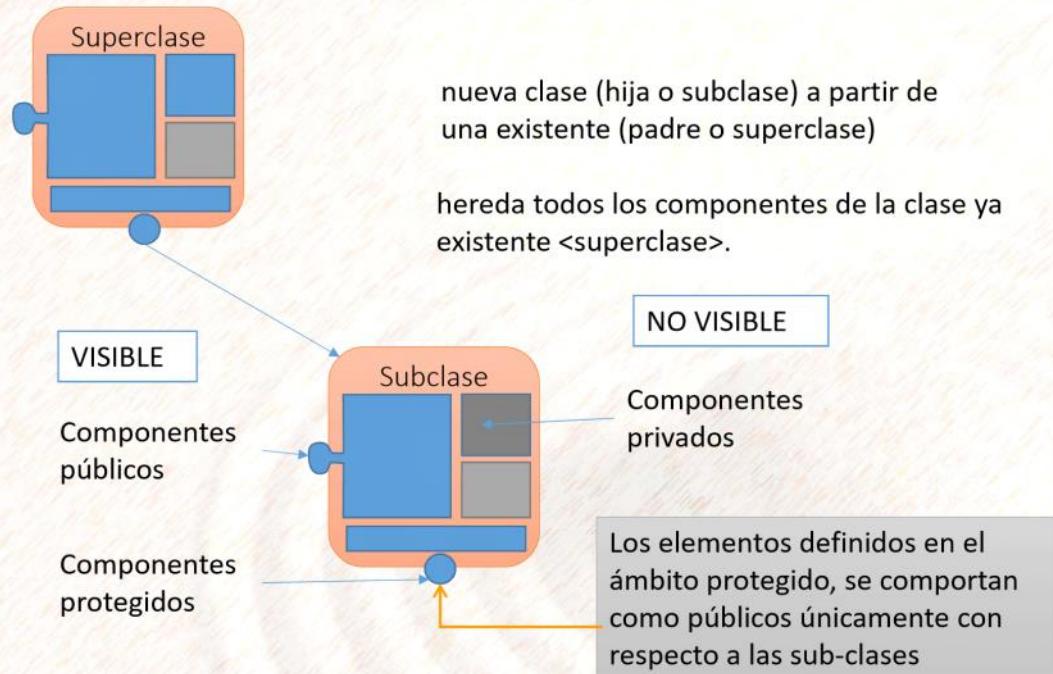
Objetivo: Conocer la técnica para crear pseudoclases empleando los prototipos de los objetos para incluir una serie de propiedades y métodos generales.



Sesión 35

- ▶ Patrones y ámbitos en objetos literales: *closures*
- ▶ Elementos tipo *Static*
- ▶ Relaciones entre clases
 - ▶ Dependencia
 - ▶ Asociación (Agregación y composición)
 - ▶ Ejemplos
 - ▶ Relaciones a nivel de objetos
 - ▶ Relaciones a nivel de funciones constructora

Herencia



Patrones de herencia en JavaScript



Patrones que trabajan con constructores ("clásicos")

- Encadenamiento de prototipos (*Prototype chaining*)
- Herencia de prototipos
- Encadenamiento con constructor temporal
- Copia de las propiedades del prototipo

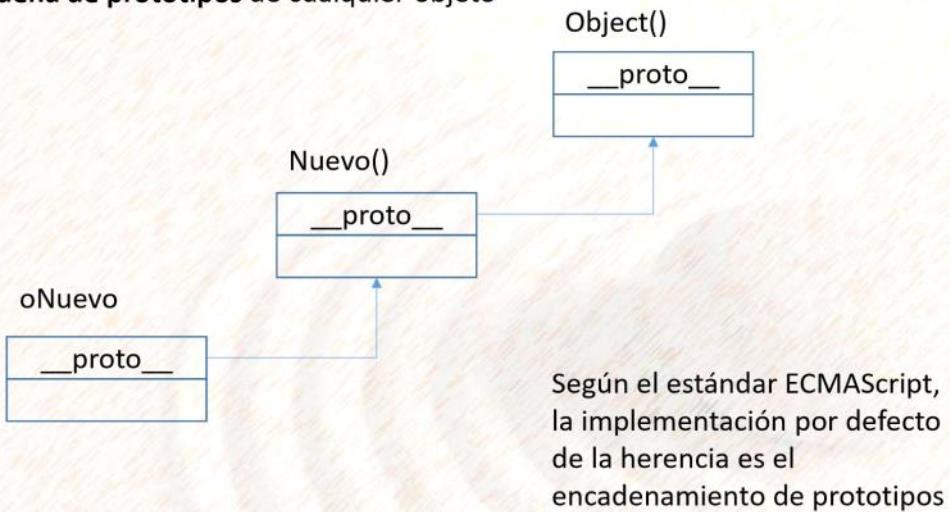
Patrones que trabajan directamente con objetos ("modernos")

- Herencia prototípica (prototypal inheritance)
- Copia de propiedades del objeto o copia superficial (*shallow copy*)
- Copia del objeto en profundidad (*deep copy*)
- Extensión y aumento
- Herencia múltiple
- Herencia parasitaria

Cadena de prototipos

Js

Al crear una función constructora nueva se produce automáticamente una relación con la función predefinida `Object()`, que es el final de la **cadena de prototipos** de cualquier objeto



Herencia: encadenamiento de prototipos (1)



Prototype Chaining

Si se definen dos funciones , Padre() e Hijo(), se establece una relación de herencia entre ambas **sobrescribiendo** completamente el prototipo de la segunda

```
Hijo.prototype = new Padre();
```

2 operaciones

- crea un objeto, una instancia de Padre()
- lo vincula al prototipo de la función hijo

Resultado;
Herencia
entre objetos

El primer paso determina la herencia desde otro objeto (instancia), no desde la función constructora

Cuando se instancie el objeto del tipo Hijo() estará heredando de un **objeto** del tipo Padre()

Herencia: encadenamiento de prototipos (2)



```
Hijo.prototype = new Padre();
```

Al sobrescribir el prototipo de la función Hijo(), se altera el valor implícito del **constructor**, que sería Padre() en lugar de hijo

Es necesario **reestablecer** explícitamente el correcto valor que indica cual debe ser tomada como **función constructora**

```
Hijo.prototype.constructor = Hijo;
```

La combinación de ambas instrucciones genera la relación de **herencia entre objetos basada en sus prototipos**

Ejemplo: Herencia (1)

```
function Figura() {...}
function Figura2D() {...}
function Triangulo(nLado, nAltura) {...}
```

```
Figura2D.prototype = new Figura();
Figura2D.prototype.constructor = Figura2D;
```

```
Triangulo.prototype = new Figura2D();
Triangulo.prototype.constructor = Triangulo;
```

```
▼ Triangulo
  ► getArea: function () { return this.side * this.height / 2; }
  height: 10
  name: "Triangle"
  side: 5
  ▼ __proto__: Triangulo
    ► constructor: function Triangulo(nLado, nAltura) {
        name: "figura"
      }
    ▼ __proto__: Figura2D
      ► constructor: function Figura2D() {
        name: "figura"
      }
      ► toString: function () { return this.name; }
    ▼ __proto__: Figura
      ► constructor: function Figura() {
      }
    ▼ __proto__: Object
```

En la consola vemos como la cadena de prototipos corresponde a la herencia que hemos implementado

Ejemplo: Herencia (2)

La herencia no implica que no se incorporen al prototipo todos los elementos convenientes, siempre que el orden sea el adecuado

```
function Figura() {}  
Figura.prototype.name = 'figura';  
Figura.prototype.toString =  
    function () {return this.name;};  
  
function Figura2D() {}  
  
// Implementacion de la herencia, necesariamente  
// antes de cualquier modificación del prototipo  
Figura2D.prototype = new Figura();  
Figura2D.prototype.constructor = Figura2D;  
  
// Agregamos elementos al prototipo  
Figura2D.prototype.name = 'figura';  
function Triangulo(nLado, nAltura) {...}
```

Encadenamiento de prototipos : problemas con el constructor



```
function Padre (parámetros ){
    this.propiedades = parámetros
};
```

```
function Hijo (parámetros ){
    this.propiedades = parámetros
};
```

Si las funciones **constructoras**, reciben parámetros, su primera labor es asignárselos a distintos propiedades del objeto que se va a crear

La función hijo recibe sus propios parámetros y los de la función padre, pero no hace nada con estos últimos: es necesario añadir una primera línea que invoque el constructor padre

```
Padre.prototype.constructor.call(this, parámetros);
```

```
Hijo.prototype = new Padre();
Hijo.prototype.constructor = Hijo();
```

Sólo así funcionara correctamente la relación de herencia finalmente creada entre ambas funciones.

Ejercicio (27a)

Objetos Persona y Estudiante (Herencia).

Crear mediante su propia función constructora un objeto **estudiante** con las propiedades nombre, edad, curso y nº de matricula (todas inicializadas por el constructor) y los método mostrar datos personales y mostrar datos académicos, ambos en el prototipo.

Las propiedades nombre y edad y el método mostrar_datos_personales se **heredaran** de un objeto más genérico, **persona**, que tendrá su propia función constructora.

Objetivo: Conocer el primer patrón de herencia descrito, “**encadenamiento de prototipos**” basado en la relación entre el prototipo de la función (clase) Hija y un objeto instanciado de la función (clase) padre. Ver los problemas específicos que surgen en relación con el constructor

Herencia entre prototipos



En una segunda opción para establecer una relación de herencia entre las funciones Padre() e Hijo(), podemos hacer que el prototipo de la segunda apunte directamente al prototipo de la primera

En lugar de → `Hijo.prototype = new Padre();`

Tendremos → `Hijo.prototype = Padre.prototype;`

Resultado:
**Herencia
entre funciones
(objetos)**

- se estará heredando sólo los elementos incluidos en el prototipo Padre()
- el prototipo de la función hijo se vincula al de la función padre: ambos prototipos serán punteros al mismo objeto, y los cambios se reflejaran en todos los objetos

Ejemplo: Herencia entre prototipos

La herencia puede vincular directamente los prototipo de los objetos padre e hijo, convirtiéndolos en referencias a un mismo objeto

```
function Figura() {}  
Figura.prototype.name = 'figura';  
Figura.prototype.toString =  
    function () {return this.name;};  
  
function Figura2D() {}  
  
// Implementacion de la herencia, necesariamente  
// antes de cualquier modificación del prototipo  
Figura2D.prototype = Figura.prototype  
Figura2D.prototype.constructor = Figura2D;  
  
// Agregamos elementos al prototipo  
Figura2D.prototype.name = 'figura';  
  
function Triangulo(nLado, nAltura) {...}...
```

Encadenamiento de prototipos con constructor temporal (1)



los prototipos de la función Hijo() y Padre() apuntando realmente a un mismo objeto suponen un problema que puede **evitarse interponiendo un constructor temporal**

```
function Figura() {...}

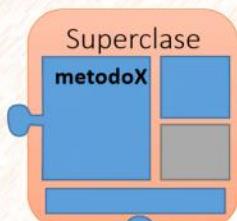
function Figura2D() { }

// Implementacion de la herencia, necesariamente
// antes de cualquier modificación del prototipo
var F = function () { }
F.prototype = Figura.prototype;
Figura2D.prototype = new F();
Figura2D.prototype.constructor = Figura2D;

// Agregamos elementos al prototipo
Figura2D.prototype.name = 'figura';
```

F() crea un objeto sin
propiedades propias y con
el prototipo del padre

Herencia y polimorfismo



La clase de la que parte la herencia se conoce como clase padre o superclase

La clase hija hereda todos los componentes de la clase anterior

La clase hija se **especializa**, añadiendo sus **nuevos métodos** y propiedades

Se pueden modificar los métodos heredados, lo que se conoce como **sobreescritura (overriding)** para dar lugar al **polimorfismo**.

palabra reservada "**super**" permite recuperar el método original, e.g. al principio del método sobreescrito con el mismo nombre

Encadenamiento de prototipos y superclass



En JS se utiliza el término alemán *uber* para referirse al acceso al método a nivel del Padre(), e.g. antes de añadirle nuevas funcionalidades

```
Figura.prototype.toString = function(){
    var result = [];
    if (this.constructor.uber) {
        result[result.length] = this.constructor.uber.toString();
    }
    result[result.length] = this.name;
    return result.join(', ');
};
```

```
Figura2D.uber = Figura.prototype
Triangulo.uber = Figura2D.prototype
```


Encadenamiento de prototipos: encapsulando en extend()



El **patrón completo** de encadenamiento de prototipos con un constructor interpuesto y la forma de acceso *uber* a los métodos de la clase padre se puede encapsular fácilmente en una función , que podemos denominar **extend()**, que es como suele denominarse en lenguajes OOP clásicos

```
function extend(Child, Parent) {  
    var F = function(){};  
    F.prototype = Parent.prototype;  
    Child.prototype = new F();  
    Child.prototype.constructor = Child;  
    Child.uber = Parent.prototype;  
}
```

La librería YUI implemente la herencia de forma muy similar

Ejercicio (27b)

Objetos Persona y Estudiante (Herencia).

Crear mediante su propia función constructora un objeto **estudiante** con las propiedades nombre, DNI, curso y nº de matricula (todas inicializadas por el constructor) y los método `mostrar_datos_personales` y `mostrar_datos_académicos`.

Las propiedades curso y nº de matricula y el método `mostrar_datos_académicos` se heredaran de un objeto más genérico, **persona**, que tendrá su propia función constructora. Crear una función `extend()` que se haga cargo del proceso de herencia clásico.

Objetivo: Conocer el primer patrón de herencia descrito basado en la relación entre el prototipo de la función (clase) Hija y un objeto instanciado de la función (clase) padre. Ver los problemas específicos que surgen en relación con el constructor

Patrones de herencia en JavaScript



Patrones que trabajan con constructores ("clásicos")

- Encadenamiento de prototipos (*Prototype chaining*)
 - Herencia de prototipos
 - Encadenamiento con constructor temporal
 - Copia de las propiedades del prototipo
-
- **Herencia prototípica (prototypal inheritance)**
 - Copia de propiedades del objeto o copia superficial (*shallow copy*)
 - Copia del objeto en profundidad (*deep copy*)
 - Extensión y aumento
 - Herencia múltiple
 - Herencia parasitaria

Patrones que trabajan directamente con objetos ("modernos")

Herencia prototípica



prototypal inheritance

Basándose en la idea de que los objetos heredan de objetos, **Douglas Crockford** sugirió una función que aceptara un objeto y devolviera uno nuevo que tuviera al primero como prototipo

```
function object(o) {  
    function F() {}  
    F.prototype = o;  
    return new F();  
}
```

Este patrón ha sido recogido por ECMAScript 5 como el nuevo método estático **Object.create()**

Si tengo un objeto oPadre (e.g. creado literalmente)
oHijo = Object.create(oPadre)

Método Object .create()

Js

Incorporado al estándar ECMAScript5

Su **segundo parámetro** son las nuevas propiedades extra que se añaden al objeto hijo

```
var child = Object.create(parent, {  
    age: { value: 2 } // ECMA5 descriptor  
});
```

Este segundo parámetro corresponde a un objeto, en el que cada propiedad se define mediante los nuevos descriptores, también incorporados en ECMA5

Alternativamente, una vez creado el objeto hijo, puede ser modificado dinámicamente como cualquier objeto JavaScript

Ejercicio (28a)

Objetos Persona y Estudiante (Herencia).

Crear literalmente un objeto **estudiante** con las propiedades nombre, DNI, curso y nº de matricula (todas inicializadas por el constructor) y los métodos `mostrar_datos_personales` y `mostrar_datos_académicos`.

Las propiedades curso y nº de matricula y el método `mostrar_datos_académicos` se heredaran de un objeto más genérico, persona, que ha sido creado previamente, también de forma literal.

Objetivo: Conocer el patrón de herencia directa entre objetos, sin la intervención de funciones constructoras.

Otros patrones de herencia directamente a nivel de objetos



- Copia de propiedades del objeto o copia superficial (*shallow copy*)
- Copia del objeto en profundidad (*deep copy*)

Ambos se basan en la posibilidad de recorrer un objeto padre con un bucle for-in copiando cada una de sus propiedades al objeto hijo

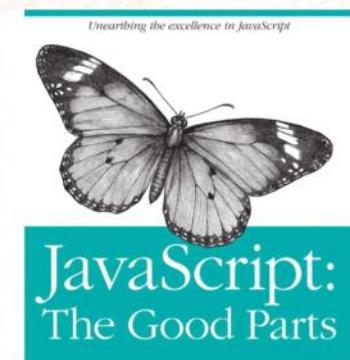
En el primer caso los elementos referenciados se copian como referencias

En el segundo, cada elemento referenciado se recorre en un nuevo for-in para copiar en el hijo cada una de sus propiedades

- Extensión y aumento
- Herencia múltiple
- Herencia parasitaria

Más Información: OOP y JS

Js



O'REILLY®

JavaScript: The Good Parts
Douglas Crockford
O'Reilly Media, 2008

Object-Oriented JavaScript.
Stoyan Stefanov
Packt Publishing, 2008

Object-Oriented JavaScript

Create scalable, reusable high-quality JavaScript
applications, and libraries

Stoyan Stefanov

[PACKT]

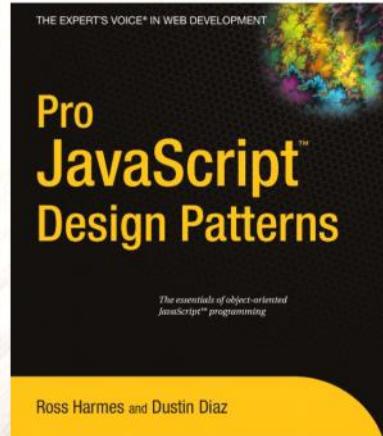
Más Información: Patrones y JS

Js

Build Better Applications with Coding and Design Patterns



O'REILLY® | Y
JavaScript Patterns.
Stoyan Stefanov
O'Reilly Media, 2010



Ross Harmes and Dustin Diaz
Pro JavaScript Design Patterns
Dustin Diaz & Ross Harmes
Apress, 2007



Sesión 36

- ▶ **Herencia**
- ▶ Patrones clásicos
 - ▶ Encadenamiento de prototipos (*Prototype chaining*)
 - ▶ Herencia de prototipos
 - ▶ Encadenamiento con constructor temporal
- ▶ Patrones de herencia entre objetos.
 - ▶ Herencia prototípica. `Object.create()`

AC49

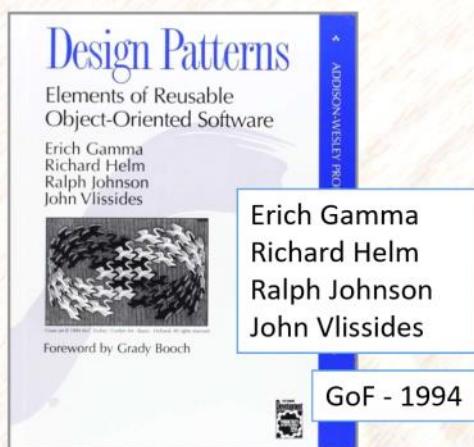
Patrones de diseño



Patrón
de diseño

Base para la búsqueda de soluciones a
problemas comunes en el desarrollo de
software

Reutilización del diseño



Para que una solución sea
considerada un patrón

- Efectivo resolviendo problemas similares en ocasiones anteriores.
- Reutilizable: aplicable a diferentes problemas de diseño en distintas circunstancias

Objetivos



Proporcionar catálogos de elementos reusables en el diseño de sistemas software

→ No reinventar la rueda

Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente

Formalizar un vocabulario común entre diseñadores

Estandarizar el modo en que se realiza el diseño

Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento existente.

Tipos de patrones de diseño



Patrones de creación

- *Abstract Factory* [Factoría Abstracta]
- *Factory Method* [**Factoría**]
- *Singleton Factory o Object Pool*
- *Prototype* [**Prototipo**]
- *Singleton*

Patrones estructurales

- Adapter [Adaptadores]
- Proxy [Proxies]
- Composite [Composición]
- Decorator [Decorador]
- Facade [Fachada]

Patrones de comportamiento

- Iterator
- Observer
- Model View Controller (MVC)
- Strategy [Estrategia]

Patrones de creación



Resuelven problemas relacionados con la creación clases y objetos.

Creación
de clases

- *Factory* [**Factoría**] o *Virtual Constructor*

Creación
de objetos

- *Abstract Factory* [Factoría Abstracta] o *Kit*
- *Builder* [Constructor]
 - *Constructor* [Constructor]
- *Prototype* [**Prototipo**]
- *Singleton* [instancia única]
 - *Singleton Factory* o *Object Pool*

Teóricamente independientes de lenguaje, pero creados en el marco de las clases y la herencia propios de C++/C# o Java.
Su aplicación en JavaScript es particular, a veces trivial

AC56

Patrón en JS



A JavaScript and jQuery Developer's Guide



Learning JavaScript Design Patterns

O'REILLY®

Addy Osmani

Learning JavaScript
Design Patterns

<http://addyosmani.com/resources/essentialjsdesignpatterns/book/>

Addy Osmani



Singleton



Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.

Este patrón puede ser utilizado cuando:

- Se requiere exactamente una instancia de una clase.
- Es necesario acceso controlado a un solo objeto.

*Singleton Factory
(Object Pool)*

Combina ambos:
singleton y *factory*

Singleton
-static uniqueInstance
-singletonData
+static instance()
+singletonOperation()

Patrón Singleton en JS



Algo tan simple como la creación de un objeto como literal sería un ejemplo de este patrón: una sola instancia de una clase sin la posibilidad de más objetos de la misma.

```
var persona1 = {  
    nombre : "Nicolas",  
    edad : 29,  
    empleo : "Programador",  
    decirNombre : function(){alert(this.nombre);}  
};
```

Al no existir clases, la propia definición de *singleton* puede considerarse técnicamente sin sentido.

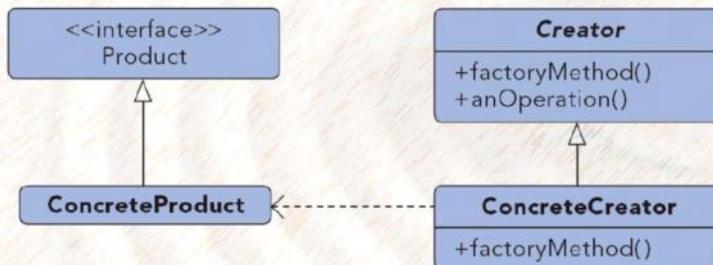


Factorías (Factory Method)

Define una interfaz para crear un objeto, pero **deja que sean las subclases quienes decidan qué clase instanciar**. Permite que una clase delegue en sus subclases la creación de objetos.

Este patrón puede ser utilizado cuando:

- Una clase no puede anticipar el tipo de objeto que debe crear.
- Subclases pueden especificar qué objetos deben ser creados.



Patrón Factory en JS



Abstacta el proceso de creación de objetos específicos, en lugar de limitarse a declararlo, e.j. mediante JSON.

En JS al no existir clases, se utiliza una función para encapsular la creación de un objeto con su interfaz específico

```
function crearPersona(nombre, edad, empleo){  
    var o = new Object();  
    o.nombre = nombre;  
    o.edad = edad;  
    o.empleo = empleo;  
    o.decirNombre = function(){alert(this.nombre);};  
    return o;}  
var persona1 = crearPersona("Nicolas", 29, "Programador");  
var persona2 = crearPersona("Gerardo", 27, "Diseñador");
```

AC54 Professional: JavaScript for Web Developers, 3th ed.
 Nicholas C. Zakas
 Wrox (Wiley), 2012
 Alejandro Cerezo, 21/12/2014

Patrón Constructor en JS



Aprovecha la posibilidad de definir constructores a medida, en lugar de funciones genéricas, para que sean ellos los que instancien los objetos

```
function Persona(nombre, edad, empleo){  
    this.nombre = nombre;  
    this.edad = edad;  
    this.empleo = empleo;  
    this.decirNombre = function(){alert(this.nombre);};  
}  
var persona1 = new Persona("Nicolas", 29, "Programador");  
var persona2 = new Persona("Gerardo", 27, "Diseñador");
```

Los elementos creados son instancias de Object, pero también lo son de Person

AC55 Professional: JavaScript for Web Developers, 3th ed.
 Nicholas C. Zakas
 Wrox (Wiley), 2012
 Alejandro Cerezo, 21/12/2014

Constructor Singleton en JS



Como ejercicio teórico, se podría plantear como mantener la característica de instancia única en un constructor invocado mediante new.

Existen al menos tres variaciones posibles de este patrón

- usar una variable global para almacenar la instancia (las variables globales no suelen ser la mejor opción)
- cachear la instancia en una propiedad estática del constructor
- envolver (*wrap*) la instancia en una envoltura (*closure*)



Sesión 37

- ▶ Patrones de diseño en JS
 - ▶ Concepto de patrones. Tipos
 - ▶ Patrones de creación en JS
 - ▶ Singleton
 - ▶ Factory
 - ▶ Constructor...



Índice

lunes, 22 de mayo de 2017 18:41