

TEMA 3

JAVASCRIPT

APLICACIONES WEB - GIS - CURSO 2019/20

Marina de la Cruz [marina.cruz@ucm.es]
Dpto de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid



Esta obra está bajo una
Licencia CC BY-NC-SA 4.0 Internacional.

Este documento está basado en <https://manuelmontenegro.github.io/AW-2017-18/02.html#/> de Manuel Montenegro [montenegro@fdi.ucm.es] bajo
una Licencia CC BY-NC-SA 4.0 Internacional.

1. INTRODUCCIÓN
2. CONCEPTOS BÁSICOS
3. LOS TIPOS DEL LENGUAJE
4. OBJETOS
5. FUNCIONES
6. ARRAYS
7. EXPRESIONES REGULARES
8. ORIENTACIÓN A OBJETOS
BASADA EN PROTOTIPOS
9. CLASES ESTÁNDAR
10. HERRAMIENTAS
11. BIBLIOGRAFÍA



INTRODUCCIÓN

EL LENGUAJE JAVASCRIPT

Javascript fue creado por Brendan Eich en 1995, para ser incluido en el navegador *Netscape*.

Netscape colaboraba en aquel momento con la empresa *Sun Microsystems*, propietaria por entonces del lenguaje Java.

Concebido inicialmente como un **lenguaje «pegamento»**, destinado a integrar los distintos componentes de las páginas web: applets, plugins, etc.

Pero su destino fue bien distinto...

ALGUNOS HITOS EN LA HISTORIA DE JAVASCRIPT

- 1997 - **HTML Dinámico**

Los programas modifican dinámicamente la estructura de un documento HTML mediante la manipulación de su DOM.

- 2005 - **AJAX**

Los programas pueden realizar peticiones al servidor desde Javascript, lo que impulsó el paradigma de aplicaciones web de una sola página (SPA).

- 2009 - **Node.js**

Permite utilizar Javascript en el lado del servidor.

JAVASCRIPT Y ECMASCIPT

En el año 1996 Netscape decidió estandarizar Javascript.

El estándar fue publicado por la organización *Ecma International*. El nombre del estándar era **ECMAScript**.

La versión actual del estándar (10^a edición) es **ECMAScript 2019**.

JAVASCRIPT EN EL NAVEGADOR

Los principales navegadores contienen un **intérprete** que permite ejecutar los programas Javascript incluidos en las páginas web.

El componente del navegador encargado de esto recibe el nombre de **motor Javascript**.

Motores Javascript más conocidos:

- **SpiderMonkey**, utilizado en Firefox.
- **V8**, utilizado en Chrome.
- **Chakra**, utilizado en Edge.

¿Y NODE.JS?

Es un intérprete del lenguaje Javascript, pensado para ejecutarse **fueras de un navegador**.

Su implementación está basada en el motor **V8** de Chrome.

Se utiliza principalmente para implementar las funcionalidades del lado del servidor en aplicaciones web.



EJECUCIÓN/DEPURACIÓN DE JAVASCRIPT EN EL NAVEGADOR

Fichero bucle.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <script src="bucle.js"></script>
  </head>
  <body>
    Carga del fichero bucle.js.
  </body>
</html>
```

Fichero bucle.js

```
"use strict";

for (let k=1; k<=5; k++) {
  console.log("k = " + k);
}
```

The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. On the left, the file tree shows 'top' and a local file system entry for 'H:/UCM/FDI/AW/bucle.html' containing 'bucle.js'. The 'bucle.js' file is open, displaying the following code:

```
1 "use strict";
2
3 for (let k=1; k<=5; k++) {
4     console.log("k = " + k);
5 }
6
7
8
```

The 'Console' tab is active, showing the output of the code execution:

- 5 messages
- 5 user messages
- No errors
- No warnings
- 5 info
- No verbose

Message	Source
k = 1	bucle.js:4
k = 2	bucle.js:4
k = 3	bucle.js:4
k = 4	bucle.js:4
k = 5	bucle.js:4

On the right side of the DevTools, there is a sidebar with various developer tools options:

- Pause on caught exceptions
- Watch
- Call Stack
- Scope
- Breakpoints
- XHR/fetch Breakpoints
- DOM Breakpoints

CONCEPTOS BÁSICOS

¿"use strict"?

JavaScript es un lenguaje muy flexible.

JavaScript tiene algunas características que hacen que el lenguaje sea, a veces, **demasiado permisivo**.

Introduciendo la cadena "`use strict`" o '`'use strict'`' al principio del programa hace que éste se evalúe en **modo estricto**.

En este modo no está permitido el uso de características demasiado permisivas del lenguaje:

- Utilizar variables sin declararlas.
- Borrar una variable mediante `delete`.
- Duplicidades en nombres de parámetros de funciones.

ASPECTOS GENERALES DEL LENGUAJE

- Sensible a mayúsculas y minúsculas.
- Terminar cada sentencia con punto y coma (no es obligatorio).
- Nombre de identificadores (variables, funciones, ...)
 - Pueden contener letras, dígitos, _ y \$
 - No pueden empezar por dígito
- Comentarios:

```
// Comentario de una línea
/* Comentario de varias
línneas */
```

DECLARACIÓN DE VARIABLES

Ámbitos de las variables:

- **Global**
 - navegador: objeto window
 - fuera del navegador: depende del entorno
- **Local**: función
- **Bloque**: {} a partir de ES2015/ES6

Declaración con las palabras reservadas:

- **var**
- **let** y **const** (desde ES2015)

Declaración de variables **globales** con **var** y **let**

```
// var permite redeclaración, let no
var x;
var x,y;
var x=8, y=9;
var x, y=9;
let z=20;
...
```

Declaración de variables **locales** con **var** y **let**

```
function foo() {
  var k=100;
  let j=89;
  console.log(k);
  console.log(j);
}
...
```

Declaración de variables **de bloque** con **var** y **let**

```
...
{
  var x=9;
  let z=20;
}
...
```

Declaración de costantes con **const**

```
...
const MAXIMO=1000;
...
```

Se permite el uso de una variable declarada con **var** antes de su declaración.

```
x=8;           // Uso de antes de la declaración  
console.log(x); // Imprime 8  
var x;         // Declaración
```

Este comportamiento es posible por el proceso de **"hoisting"**: mover las declaraciones de las variables al principio de su ámbito.

```
var x;          // Declaración  
x=8;           // Uso de antes de la declaración  
console.log(x); // Imprime 8
```

El hoisting sólo se realiza sobre las variables, no sobre los valores.

```
console.log(x);      // Imprime undefined
var x=8;            // Declaración
console.log(x);      // Imprime 8
```

Declarar siempre las variables al principio del ámbito.

COMPARACIÓN ENTRE var Y let

- El **hoisting** sólo se hace sobre variables **var** en cualquier ámbito.

```
console.log(x);      // Imprime undefined
var x=8;            // Declaración
console.log(y);      // ReferenceError: y is not defined
let y=9;             // Declaración
```

- Ámbito **global**: sólo las **var** pertenecen al objeto global.

```
var x=8;
let y=9;
console.log(window.x);    // Imprime 8
console.log(window.y);    // Imprime undefined
```

- **Redeclaraciones**
 - Con **var** se permite en cuquier ámbito (cuidado con los bloques).

```
var x=8;  
// aquí x vale 8  
var x=1;  
// aquí x vale 1
```

```
var x=8;  
// aquí x vale 8  
{  
var x=1;  
// aquí x vale 1  
}  
// aquí x vale 1
```

- Con **let** NO se permite en ningún ámbito.

```
let y=9;  
  
let y=2;      // Identifier 'y' has already been declared
```

Siempre que sea posible utilizar **let**.

SENTENCIAS CONDICIONALES

- **if-then-else**

```
if(x < y) {  
    console.log("x es menor que y");  
}  
else {  
    console.log("x NO es menor que y");  
}
```

- **switch**

```
switch(day) {  
    case 6:  
    case 7:  
        console.log("Es fin de semana :-)");  
        break;  
    default:  
        console.log("Es día laborable :-(");  
}
```

BUCLES

- **while**

```
let resultado = 0;
let numero = 100;
let i = 0;

while(i ≤ numero) {
    resultado += i;
    i++;
}
```

- **do-while**

```
let resultado = 0;
let numero = 100;
let i = 0;

do {
    resultado += i;
    i++;
} while(i <= numero)
```

- **for**

```
for ( let k=0; k<10 ; k++) {  
    console.log( "El valor del k es: " + k );  
}
```

- **for...of** para recorrer arrays

```
let arr = [4, 6, 10];  
let sum = 0;  
for (let x of arr) {  
    sum += x;  
}
```

- break y continue

```
let i = 0;
let x = "fichero.pdf"

while (i < x.length) {
    if (x[i] === ".") break;
    console.log(x[i]);
    i++;
}
```

```
let x = [1,2,3,4,5];
let z = 1;

for (let i = 0; i < x.length; i++) {
    if (x[i] % 2 === 0)
        continue;
    console.log(x[i]);
    z *= x[i];
}

console.log(z);
```

FUNCIONES

- Definición de funciones

```
function abs(x) {  
    if (x < 0) {  
        return -x;  
    }  
    return x;  
}
```

- Llamadas a funciones y métodos

```
let x = abs(-3);  
console.log("El valor absoluto de x es ", x);
```

MANEJO DE EXCEPCIONES

- try-catch o try-catch-finally

```
try {
    funcion_no_existe();
} catch (e) {           No se especifica el tipo de excepción
    console.log(e.message);
} finally {
    console.log("finally se ejecuta siempre");
}
```

- Lanzamiento de excepciones

```
throw new Error("Inconsistencia en los parámetros de la función");
```

- Atributos de Error

- **Error.message**: mensaje de error.
- **Error.stack**: pila de ejecución.
- **Error.name**: nombre de la clase del error.

CADENAS DE TEXTO

- Inicialización:

```
var str = "Esto es una cadena";
```

o bien

```
let str = 'Esto es una cadena';
```

- Acceso al carácter i-ésimo:

```
str[2] // → "t"
```

- Algunos métodos:

```
str.slice(2, 5);  
// → "to es"  
" vale ".trim();  
// → "vale"  
str.split(" ");  
// → ["Esto", "es",  
//      "una", "cadena"]  
str.toUpperCase();  
// → "ESTO ES UNA CADENA"  
str.startsWith("Est");  
// → true  
"ab".repeat(5);  
// → "abababab"
```

CADERAS PLANTILLA

Si delimitamos una cadena entre comillas invertidas (`), podemos utilizar la sintaxis `${...}` para introducir expresiones Javascript en su contenido.

```
let nombre = "Juan";
let edad = 27;
let cadena = `Me llamo ${nombre} y tengo ${edad} años`;
console.log(cadena);
// → Me llamo Juan y tengo 27 años

console.log(`Pero el año que viene tendré ${edad + 1} años`);
// → Pero el año que viene tendré 28 años
```

OPERADORES

- Relacionales: `==`, `==`, `!=`, `!==`, `<`, `<=`, `>`, `>=`
- Aritméticos: `+`, `-`, `*`, `/`, `%`
- Lógicos: `&&`, `||`, `!`
- A nivel de bit: `&`, `|`, `^`, `>>`, `<<`, `>>>`

ARRAYS

- Inicialización:

```
let x = [4, 6, "pepe", 1, 3];  
  
let z = [] ; // array vacío  
  
let m = new Array(3);
```

- Acceso:

```
console.log(x[3]); // → 1  
m[2] = "Elemento nuevo";
```

- Longitud:

```
x.length // → 5  
z.length // → 0  
m.length // → 3
```

OBJETOS LITERALES

- Inicialización:

```
let x = {  
    nombre: "Nicolás",  
    apellidos: "Ortega",  
    edad: 14  
};  
  
let y = {};  
      // objeto vacío
```

- Acceso:

```
console.log(x.nombre); // → Nicolás  
console.log(x["nombre"]); // → Nicolás
```

- Modificación:

```
x.edad++ ;
```

LOS TIPOS DEL LENGUAJE

TIPOS DE LENGUAJES (SEGÚN LA GESTIÓN DE LOS TIPOS)

- Lenguajes **estáticamente tipados**

Se detecta **en tiempo de compilación** que las operaciones se realizan sobre argumentos del tipo correcto.

- Lenguajes **dinámicamente tipados**

Se comprueba **durante la ejecución del programa** que las operaciones se realizan sobre argumentos de tipo correcto.

Ejemplo: detección del error $3 * "foo"$

```
if (...) {  
    y = 3 * "foo"; }
```

- Estáticamente: en compilación.
- Dinámicamente: sólo si se cumple el if.

JavaScript es un lenguaje **dinámicamente tipado** ...
... y con ciertas "peculiaridades".

TIPOS DISPONIBLES EN JAVASCRIPT

- **number**: tipo numérico (no hay distinción entre enteros y coma flotante).
- **boolean**: tipo booleano (incluye los valores **true** y **false**).
- **string**: tipo cadena.
- **undefined**: valor no definido.
- **function**: funciones.
- **object**: tipo objeto (incluye también arrays, expresiones regulares, etc).

EL VALOR INDEFINIDO (`undefined`)

Se utiliza para las variables no inicializadas y para atributos no existentes dentro de objetos.

```
let coordenadas = { x: 5, y: 6 };
let v;
console.log(v); // → undefined
console.log(coordenadas.z); // → undefined
```

EL VALOR NULO (`null`)

Se utiliza para denotar una referencia a objeto nula.

```
let x = null; // La variable 'x' esta inicializada, pero a una
              // referencia nula.
console.log(x); // → null
```

COMPROBACIÓN DE TIPOS

La función **typeof** permite obtener el tipo de un elemento.
Devuelve una cadena con el nombre del tipo.

```
"use strict";  
  
var x=10;  
console.log(typeof(x));      // number  
x = 1.23;  
console.log(typeof(x));      // number  
  
x = true;  
console.log(typeof(x));      // boolean  
x = false;  
console.log(typeof(x));      // boolean  
console.log(typeof(3<4));   // boolean  
  
x = "cadena";  
console.log(typeof(x));      // string  
  
console.log(typeof(z));      // undefined  
  
// continúa ...
```

```
function suma(x,y) {  
    return x+y;  
}  
console.log(typeof(suma)); // function  
  
x = {alto:2, ancho:10};  
console.log(typeof(x)); // object  
x = [1,2,3]  
console.log(typeof(x)); // object  
x = new Date();  
console.log(typeof(x)); // object  
  
console.log(typeof(null)); // object
```

La función **typeof** no permite distinguir entre un objeto, un array, el valor **null**, etc.

La función **instanceof** si permite esa distinción.

CONVERSIONES JAVASCRIPT

¿A qué valor se evalúan las siguientes expresiones?

```
"3" * 4          // → 12
3 * 4          // → 12
"3" * "4"        // → 12
"3" * "pepe"      // → NaN
"12" + "20"        // → 1220
"12" + 20          // → 1220
12 + "20"          // → 1220
Math.log10("1000") // → 3
"10" < "2"          // → true
"10" < 2            // → false
```

EL VALOR NOT-A-NUMBER (NaN)

Se devuelve como resultado de operaciones aritméticas incorrectas:

```
Math.log(-2)      // → NaN  
parseInt("x2d") // → NaN
```

¡Cuidado con las comparaciones con **NaN**!

```
Math.log(-3) === NaN // → false  
NaN === NaN        // → false
```

Si se quiere determinar si una operación ha dado **NaN** como resultado, debe utilizarse la función **isNaN**

```
isNaN(NaN)          // → true  
isNaN(Math.log(-3)) // → true
```

MÁS CONVERSIONES JAVASCRIPT

¿En qué casos se cumple la condición del **if**?

```
if (23) { .... }           // → se cumple
if (-1) { .... }           // → se cumple
if (0) { .... }            // → no se cumple
if ("Pepe") { .... }       // → se cumple
if ("") { .... }           // → no se cumple
if ([1, 3]) { .... }       // → se cumple
if ([]) { .... }           // → se cumple
if (null) { .... }          // → no se cumple
if (undefined) { .... }      // → no se cumple
```

CÓMO EVITAR CONFUSIONES

Con este panorama, hay dos alternativas:

1. Aprenderse concienzudamente las reglas de conversión de JavaScript:

Información:

<http://webreflection.blogspot.com.es/2010/10/javascript-coercion-demystified.html>

2. **[Recomendado]** Hacer las conversiones explícitamente (si se duda del tipo de una expresión).

Funciones `Number(...)`, `String(...)`, `Boolean(...)`

FUNCIONES DE CONVERSIÓN

La función Number()

```
Number("32")           // → 32
Number("2f3")          // → NaN
Number(true)           // → 1
Number(false)          // → 0
Number(undefined)      // → NaN
Number(null)           // → 0
Number(new Date())     // → 1476191814528 (depende de fecha y hora)
```

Cuando la función **Number** se llama sobre un objeto **x**, la conversión se aplica sobre **x.valueOf()**.

Ver también: **parseInt** [+]

La función **String()**

```
String(true)      // → "true"  
String(undefined) // → "undefined"  
String(32)        // → "32"  
String(new Date()) // → "Tue Oct 11 2016 15:23:02 GMT+0200 (CEST)"
```

La función **String** aplicada sobre un objeto **x** llama al
método **x.toString()**

La función Boolean()

- Valores falsos: `undefined`, `null`, `false`, `0`, `NaN`, `""`.
- Valores ciertos: el resto.

```
Boolean("")          // → false
Boolean(undefined)  // → false
Boolean(null)       // → false
Boolean(34)         // → true
```

LOS OPERADORES DE COMPARACIÓN == Y ===

- == compara sólo valores.
- === compara valores y tipos.

```
"25" == 25          // → true
"25" === 25        // → false
false == 0          // → true
"" == 0             // → true
2.0 === 2           // → true (recuerda: no se distingue entre tipo
                     //       de enteros y de coma flotante)
```

Cuando se compara un string con un número, se convierte el string a número. Si el string es vacío se convierte a 0. Si el string no es numérico se convierte a NaN

Los operadores de comparación != y !== trabajan de manera similar a == y ===

TIPOS PRIMITIVOS VS. OBJETOS

JavaScript es un lenguaje basado en objetos. Cada variable será de tipo **primitivo** o de tipo **Object**.

- **Tipos primitivos**
 - Numérico
 - Booleano
 - Cadena
 - Indefinido
 - Nulo

- **Tipo Object**
 - Objetos
 - Funciones

La función `instanceof` permite conocer si un objeto pertenece a una clase. Devuelve `true/false`.

```
x = [1,2,3]
console.log(x instanceof Array);      // true

x = new Date();
console.log(x instanceof Date);      // true

x = 14;
console.log(x instanceof Array);      // false
console.log(x instanceof Date);      // false
```

Todos los objetos heredan de `Object`.

```
x = [1,2,3]
console.log(x instanceof Object);    // true

x = new Date();
console.log(x instanceof Object);    // true

x = 14;
console.log(x instanceof Object);    // false
```



elemento	typeof	primitivo/Object
8	"number"	primitivo
true	"boolean"	primitivo
"cadena"	"string"	primitivo
var x	"undefined"	primitivo
null	"object"	primitivo
NaN	"number"	primitivo
function fun(...) {...}	"function"	Object
[1,2,3]	"object"	Object (Array)
{x:1, y:2}	"object"	Object

OBJETOS

OBJETOS EN JAVASCRIPT

Un **objeto** en Javascript no es más que una colección de **atributos**, cada uno de ellos asociado a un **valor**.

```
let x = {  
    nombre: "Ana María",  
    apellidos: "Gamboa Esteban",  
    edad: 54  
};
```

Los atributos se definen **al crear el objeto**.

El literal **{}** representa un objeto vacío (sin atributos)

```
let y = {};
```

El acceso a los atributos de un objeto se realiza mediante:

- El operador punto (.), igual que en Java.

```
x.apellidos // → "Gamboa Esteban"
```

- o bien, mediante el operador corchete

```
x["apellidos"] // → "Gamboa Esteban"
```

```
let atrib = "nombre";
x[atrib] // → "Ana María"
```

El acceso a una propiedad inexistente devuelve **undefined**

```
x.noexiste // → undefined
```

Modificación de atributos:

```
x.edad = x.edad + 1; // o bien: x.edad++  
x["nombre"] = "Ana Josefa";
```

Es posible añadir atributos sobre la marcha:

```
x.direccion = "Calle Bautista, 25";  
y.nombre = "Javier";  
  
console.log(x);  
// { nombre: 'Ana Josefa', apellidos: 'Gamboa Esteban', edad: 55,  
//   direccion: "Calle Bautista, 25" }  
console.log(y);  
// { nombre: 'Javier' }
```

...y también borrarlos:

```
delete x.edad;  
console.log(x);  
// { nombre: 'Ana Josefa', apellidos: 'Gamboa Esteban' }
```

Los nombres de atributos no han de ser necesariamente identificadores válidos de Javascript. En caso de no serlo, han de aparecer entre comillas en la declaración:

```
let z = {  
    "Atributo con espacios": 21,  
    "14": "foo",  
    "false": "ok"  
};
```

Para acceder a estos atributos solo se puede utilizar la notación corchete

```
z["Atributo con espacios"] = 22;
```

La función `Object.keys()` devuelve un array con los nombres de propiedades de un objeto:

```
let x = {
    nombre: "Ana María",
    apellidos: "Gamboa Esteban",
    edad: 54
};

console.log(Object.keys(x));
// [ 'nombre', 'apellidos', 'edad' ]
```

El operador `in` permite determinar la existencia de un atributo dentro de un objeto:

```
if ("edad" in x) {
    console.log("x tiene un atributo llamado 'edad'");
}
```

IGUALDAD DE OBJETOS

Cuando se aplica el operador `==` o `===` sobre objetos, se comprueba que las referencias a ambos lados del operador apuntan al mismo objeto.

```
let coords1 = { x: 20, y: 30 };  
let coords2 = { x: 20, y: 30 };  
let coords3 = coords1;  
  
console.log(coords1 === coords2); // → false  
  
console.log(coords1 === coords3); // → true
```

FUNCIONES

DEFINICIÓN DE UNA FUNCIÓN

```
function imprime_args(p1, p2, p3) {  
    console.log(`p1: ${p1}`);  
    console.log(`p2: ${p2}`);  
    console.log(`p3: ${p3}`);  
}
```

Observar en la cabecera de la función que no se especifica nada acerca de los tipos de los parámetros ni del tipo de retorno de la función.

LLAMADA A UNA FUNCIÓN

```
imprime_args(1, "bar", true);
```

Resultado:

```
p1: 1  
p2: bar  
p3: true
```

Algunas características de las funciones:

- Se puede asignar una función a una variable.
- Pueden pasarse funciones como parámetros.
- Pueden devolverse funciones (como retorno de otras funciones).

ASIGNACIÓN DE UNA FUNCIÓN A UNA VARIABLE

Partimos de las siguientes definiciones:

```
function incrementar(x) {  
    return x + 1;  
}  
  
function duplicar(x) {  
    return 2 * x;  
}  
  
function cuadrado(y) {  
    return y * y;  
}  
  
function factorial(n) {  
    if (n <= 0) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

Asignamos algunas de estas funciones a variables:

```
let i = incrementar;  
console.log(i(5));  
// Imprime: 6
```

```
let f = factorial;  
console.log(f(10));  
// Imprime: 3628800
```

¡Cuidado con los paréntesis!

```
let i = incrementar(); // INCORRECTO  
// Esto realiza la llamada incrementar(undefined), y asigna el  
// valor resultante (que también es undefined) a i.  
  
console.log(i(5));  
// ERROR: i no es una función
```

FUNCIONES COMO PARÁMETROS DE FUNCIONES

La siguiente función recibe un array de funciones y un valor. Aplica cada una de las funciones del array al valor dado y muestra los resultados por consola:

```
function aplicar_funciones(funss, z) {  
    for (let i = 0; i < funss.length; i++) {  
        console.log(`Aplicar función ${i} pasando ${z}: ${funss[i](z)} `)  
    }  
}
```

Ejemplo:

```
aplicar_funciones([incrementar,duplicar,cuadrado,factorial], 5);
```

```
Aplicar función 0 pasando 5: 6  
Aplicar función 1 pasando 5: 10  
Aplicar función 2 pasando 5: 25  
Aplicar función 3 pasando 5: 120
```

FUNCIONES COMO RETORNO DE FUNCIONES

De igual modo, se puede devolver una función como resultado:

```
function buscar_por_nombre(nombre) {  
    switch(nombre) {  
        case "INC": return incrementar;  
        case "DUP": return dup;  
        case "SQR": return cuadrado;  
        case "FCT": return factorial;  
    }  
    // Si la función termina sin alcanzar un return,  
    // se considera que devuelve undefined  
}
```

Ejemplo:

```
var g = buscar_por_nombre("INC");  
console.log(g(10));
```

FUNCIONES COMO EXPRESIONES

Se puede declarar una función utilizando una expresión.

En estos casos es posible omitir el nombre de la función
(función anónima)

```
let f = function() { console.log("Hola"); };
f();

let g = function(x, y) { return x + y; };
console.log(g(3, 5));
```

Puede no ser una función anónima, pero no está disponible fuera de la función.

```
let g = function suma(x, y) { return x + y; };
console.log(suma(3, 5)); // ReferenceError: suma is not defined
```

Dar un nombre a una función definida como expresión permite la escritura de funciones recursivas.

```
var factorial = function fac(n) {return n<2 ? 1 : n*fac(n-1)};
```

En el ejemplo anterior:

```
aplicar_funciones(  
  [ function(x) { return x - 3; },  
   function(x) { return Math.sqrt(x); },  
   factorial,  
   function(z) { return Math.log(z); } ], 2);
```

Aplicar función 0 pasando 2: -1

Aplicar función 1 pasando 2: 1.4142135623730951

Aplicar función 2 pasando 2: 2

Aplicar función 3 pasando 2: 0.6931471805599453

¿Puede reemplazarse la referencia a **factorial** por otra función anónima?

HOISTING DE FUNCIONES

El hoisting se aplica a las declaraciones de las variables (**var**) y de las funciones.

NO se aplica a las funciones declaradas como expresiones.

```
"use strict";

let s;
let y;
let inc;

s = suma(6,7); // Se ejecuta sin problemas
function suma (a,b) {
    return a+b;
}

y = inc(8);      // TypeError: inc is not a function
inc = function (x) {return ++x;};
```

NOTACIÓN LAMBDA

Existe una sintaxis más sencilla para denotar funciones anónimas.

Las funciones escritas con esta notación se denominan **funciones flecha**

En lugar de:

```
function (x, y, z) { /* ... */ }
```

Puede escribirse:

```
(x, y, z) => { /* ... */ }
```

Si la función anónima solo tiene un parámetro pueden omitirse los paréntesis iniciales.

En lugar de:

```
function (x) { console.log(`Valor recibido: ${x}`); }
```

Puede escribirse:

```
x => { console.log(`Valor recibido: ${x}`); }
```

Además, si el cuerpo de la función es de la forma `return` `exp`, pueden omitirse las llaves y el `return`.

En lugar de:

```
function (x) { return x + 1; }
```

Puede escribirse:

```
x => x + 1
```

En el ejemplo anterior:

```
aplicar_funciones(  
  [ x => x - 3,  
    x => Math.sqrt(x),  
    factorial,  
    x => Math.log(x) ], 2);
```

NÚMERO DE ARGUMENTOS EN LA LLAMADA

El número de argumentos en la llamada a la función no tiene que coincidir con el número de parámetros en la definición.

```
function imprimirArgumentos(a1,a2,a3) {  
    console.log(`a1: ${a1}`);  
    console.log(`a2: ${a2}`);  
    console.log(`a3: ${a3}`);  
}  
  
imprimirArgumentos(1, true, "foo");
```

```
a1: 1  
a2: true  
a3: foo
```

- Si se proporcionan argumentos «de más» se ignoran los sobrantes:

```
imprimeArgumentos ("uno", "dos", "tres", "cuatro");
```

```
a1: uno  
a2: dos  
a3: tres
```

- Si faltan argumentos, los parámetros correspondientes tomarán el valor **undefined**.

```
imprime_args ("uno", "dos");
```

```
a1: uno  
a2: dos  
a3: undefined
```

PARÁMETROS POR DEFECTO

Desde ES6 es posible asignar valores por defecto a los parámetros de las funciones.

El valor por defecto se aplica si el parámetro no se pasa en la llamada o bien si se pasa el valor **undefined**.

```
function multiplicar(a, b = 1) {  
    return a*b;  
}  
  
multiplicar(5); // 5  
multiplicar(5,undefined); // 5
```

PARÁMETROS NOMINALES

Utilizando *objetos* podemos *simular* el paso de *parámetros nominales*.

Por ejemplo, supongamos una función `abrir_fichero` que espera un nombre de fichero y, opcionalmente:

- Un parámetro `solo_lectura` que indica si el fichero se abre en modo lectura o en modo lectura/escritura.

Valor por defecto: `true`

- Un parámetro `binario` que indica si el fichero es binario o no.

Valor por defecto: `false`

Ejemplos de llamadas

```
abrir_fichero("mio.txt");
// Abriendo fichero mio.txt en modo lectura

abrir_fichero("mio.txt", { solo_lectura: false });
// Abriendo fichero mio.txt en modo lectura/escritura

abrir_fichero("mio.txt", { binario: true });
// Abriendo fichero binario mio.txt en modo lectura

abrir_fichero("mio.txt", { binario: true, solo_lectura: false });
// Abriendo fichero binario mio.txt en modo lectura/escritura

abrir_fichero("mio.txt", { solo_lectura: true, binario: false });
// Abriendo fichero mio.txt en modo lectura
```

Implementación

```
// El objeto 'ops' tiene como atributos los parámetros nominales.

function abrir_fichero(nombre, ops = {}) {
    // Inicialización de los parámetros nominales no pasados
    if (ops.solo_lectura === undefined) ops.solo_lectura = true;
    if (ops.binario      === undefined) ops.binario      = false;

    // Cuerpo de la función
    console.log(`Abriendo fichero ${ops.binario ? "binario" : ""} ` +
               `${nombre} en modo ` +
               `${ops.solo_lectura ? "lectura" : "lectura/escritura"}`);
}
```

Sintaxis alternativa

```
function abrir_fichero(nombre, ops={solo_lectura:true, binario:false}

console.log(`Abriendo fichero ${ops.binario ? "binario" : ""} ` +
           `${nombre} en modo ` +
           `${ops.solo_lectura ? "lectura" : "lectura/escritura"}`);
}
```

EL OBJETO arguments

Es una variable local de las funciones (no flecha).

- Es un objeto "parecido" al objeto **Array**.
- Contiene los argumentos de la llamada a la función.
- Se puede acceder al número de argumentos de la llamada con su propiedad **length**.

```
function imprimeArgumentos(a,b,c) {  
    for (let i=0; i<arguments.length; i++) {  
        console.log(arguments[i]);  
    }  
}  
imprimeArgumentos(1,2,3); // Imprime 1 2 3
```

Ejemplo de uso del objeto arguments.

```
//  
// Función que busca el mínimo de un conjunto de valores  
  
function minimo() {  
    let min = arguments[0];  
    for (let i=0; i<arguments.length; i++) {  
        if (arguments[i]<min) {  
            min = arguments[i];  
        }  
    }  
    return min;  
}  
  
console.log(minimo()); // undefined  
console.log(minimo(1)); // 1  
console.log(minimo(3,4,5)); // 3  
console.log(minimo(9,8,7,6,5,4,3,2,1,0)); // 0
```

FUNCIONES DENTRO DE OBJETOS

Las funciones pueden ser asignadas a los atributos de un objeto.

```
var empleado = {  
    nombre: "Manuel",  
    saludar: function() {  
        console.log("¡Hola!"); } } ;  
  
empleado.saludar(); // → ¡Hola!
```

¡No utilizar expresiones lambda aquí!

Este tipo de funciones reciben el nombre de **métodos**.

Se puede añadir métodos a un objeto ya construido.

```
empleado.despedir = function() { console.log("¡Adios!"); } ;  
empleado.despedir(); // → ¡Adios!
```

this

empleado.saludar();

En toda llamada a un método se distinguen tres componentes:

- Método llamado: saludar
- Argumentos (ninguno, en este caso)
- Objeto sobre el que se realiza la llamada: empleado

Cuando se llama a un método, éste recibe, además de los correspondientes argumentos, una variable especial (**this**) que contiene una **referencia al objeto sobre el que se realiza la llamada**.

Ejemplo

```
var empleado = {
    nombre: "Manuel",

    saludar: function() {
        console.log(`¡Hola, ${this.nombre}!`);
    },
    cambiarNombre: function(nuevoNombre) {
        this.nombre = nuevoNombre;
    }
};
```

```
empleado.saludar(); // → ¡Hola, Manuel!
```

```
empleado.cambiarNombre("Irene");
```

```
empleado.saludar(); // → ¡Hola, Irene!
```

Pueden transferirse métodos entre distintos objetos.

```
var otro_empleado = {  
    nombre: "David",  
    saludar: empleado.saludar  
};  
  
otro_empleado.saludar(); // → ¡Hola, David!
```

Se imprime el nombre de **otro_empleado**, porque es el objeto que recibe la llamada, aunque se llame a un método proveniente de otro objeto.

ARRAYS

INICIALIZACIÓN DE ARRAYS

Un array puede inicializarse enumerando sus elementos:

```
let a = [23, 12, 69, 11, 34, 45];
```

o bien mediante el constructor **Array**:

```
let b = new Array(10);
// Todos los elementos tienen el valor 'undefined'
```

LOS ARRAYS SON OBJETOS

Es posible asignar propiedades arbitrarias a un array.

```
let a = [23, 12, 69, 11, 34, 45];
a.estaOrdenado = false;

console.log(a);
// → [ 23, 12, 69, 11, 34, 45, esta_ordenado: false ]
```

Todos los arrays extienden a la clase **Array**, que contiene algunos métodos de utilidad sobre arrays. [\[+\]](#)

LOS ARRAYS SON FLEXIBLES Y PUEDEN TENER «HUECOS»

Puede variarse la longitud de un array en tiempo de ejecución. Basta con modificar la propiedad `length`:

```
let a = [23, 12, 69, 11, 34, 45];
a.length += 2; // Ampliamos el array

console.log(a); // → [ 23, 12, 69, 11, 34, 45, , ]
```



```
a.length = 3; // Reducimos el array

console.log(a); // → [ 23, 12, 69 ]
```

También se puede ampliar el array añadiendo elementos fuera de su rango:

```
a[5] = 32;
console.log(a); // → [ 23, 12, 69, , , 32 ]
```

Métodos que modifican el tamaño del array:

- **push(x)**

Inserta **x** al final del array.

- **pop()**

Elimina y devuelve el último elemento del array.

- **unshift(x)**

Añade **x** al principio del array, desplazando los restantes elementos.

- **shift()**

Elimina el primer elemento del array, desplazando los restantes elementos.

- **splice(ini, num)** Partiendo del elemento en la posición **ini**, elimina **num** elementos.

Ejemplo:

```
let a = [1, 2, 3, 4, 5];
// a = [1, 2, 3, 4, 5];

a.push(8);
// a = [1, 2, 3, 4, 5, 8];

a.unshift(-4);
// a = [-4, 1, 2, 3, 4, 5, 8];

a.pop(); // → 8
// a = [-4, 1, 2, 3, 4, 5];

a.shift(); // → -4
// a = [1, 2, 3, 4, 5];

a.splice(2, 2); // → [3, 4]
// a = [1, 2, 5];
```

Otras operaciones destructivas:

```
a = [4, 7, 4, 1, 3, 5];
a.sort();
    // a = [1, 3, 4, 4, 5, 7]

a.reverse();
    // a = [7, 5, 4, 4, 3, 1]
```

Operaciones no destructivas:

- **concat(arr_1, ..., arr_n)**

Añade los arrays pasados como argumento y devuelve el resultado.

```
[1, 2, 3].concat([4, 5], [6, 7, 8]);  
// → [1, 2, 3, 4, 5, 6, 7, 8]
```

- **slice(ini, fin)**

Devuelve el segmento **[ini, fin]** del array.

```
["a", "b", "c", "d", "e", "f", "g"].slice(2, 5);  
// → ["c", "d", "e"]
```

- **join(sep)**

Concatena los elementos del array intercalando **sep** como separador:

```
["Esto", "no", "me", "gusta"].join(" - ");
// → "Esto - no - me - gusta"
```

Búsqueda de valores:

- `indexOf(elem, [pos_inicial])`

Devuelve el índice de la primera aparición de `elem` en el array a partir de `pos_inicial`, o -1 si no se encuentra.

- `lastIndexOf(elem)`

Devuelve el índice de la última aparición de `elem` en el array, o -1 si no se encuentra.

FUNCIONES DE ORDEN SUPERIOR

Una función de **orden superior** es una función que recibe funciones como parámetro y/o devuelve funciones.

Javascript proporciona varios métodos de orden superior para arrays que son muy útiles en la práctica.

FUNCIONES DE ITERACIÓN

- **forEach(f)**

Aplica la función **f** sobre todos los elementos del array.

```
let personas = [ { nombre: "Ricardo", edad: 45 },
                 { nombre: "Julia", edad: 24 },
                 { nombre: "Ashley", edad: 28 } ];

personas.forEach(p => {
  console.log("Hola, me llamo " + p.nombre
              + " y tengo " + p.edad + " años");
})
```

La salida esperada es:

```
Hola, me llamo Ricardo y tengo 45 años
Hola, me llamo Julia y tengo 24 años
Hola, me llamo Ashley y tengo 28 años
```

En la función `forEach(f)`, `f` tiene tres parámetros.

- `v`: el valor de elemento en proceso de array.
- `i`: el índice del elemento en proceso dentro del array .
- `a`: el array total.

```
personas.forEach((v,i,a) => {
    console.log("Hola,me llamo " + v.nombre
                + " y tengo " + v.edad + " años"
                + " y soy el " + i + " de un array de "
                + a.length +" elementos");
})
```

La salida esperada es:

Hola,me llamo Ricardo y tengo 45 años, soy el 0 de un array de 3 elem
Hola,me llamo Julia y tengo 24 años, soy el 1 de un array de 3 elem
Hola,me llamo Ashley y tengo 28 años, soy el 2 de un array de 3 elem

FUNCIONES DE TRANSFORMACIÓN

- $\text{map}(f) \rightarrow f(v, i, a)$

Aplica la función f a cada elemento del array, devolviendo otro array con los resultados.

```
var a = [1, 3, 5, 2, 4];
let dobles = a.map(n => n * 2);
console.log(dobles); // → [2, 6, 10, 4, 8]
```

- $\text{filter}(f) \rightarrow f(v, i, a)$

Selecciona los elementos x del array tales que $f(x)$ devuelve true y devuelve un array con dichos elementos.

```
var a = [1, 3, 5, 2, 4];
let pares = a.filter(n => n % 2 === 0);
console.log(pares); // → [2, 4]
```

FUNCIONES DE REDUCCIÓN (I)

- $\text{every}(f) \rightarrow f(v, i, a)$

Devuelve **true** si para *todo* elemento **x** del array, **f(x)** devuelve **true**.

```
var a = [1, 3, 5, 2, 4];
var sonNumeros = a.every( n => typeof n === "number");
console.log(sonNumeros); // → true

var b=[1, 2,"foo"];
var sonNumeros = b.every( n => typeof n === "number");
console.log(sonNumeros); // → false
```

- $\text{some}(f) \rightarrow f(v, i, a)$

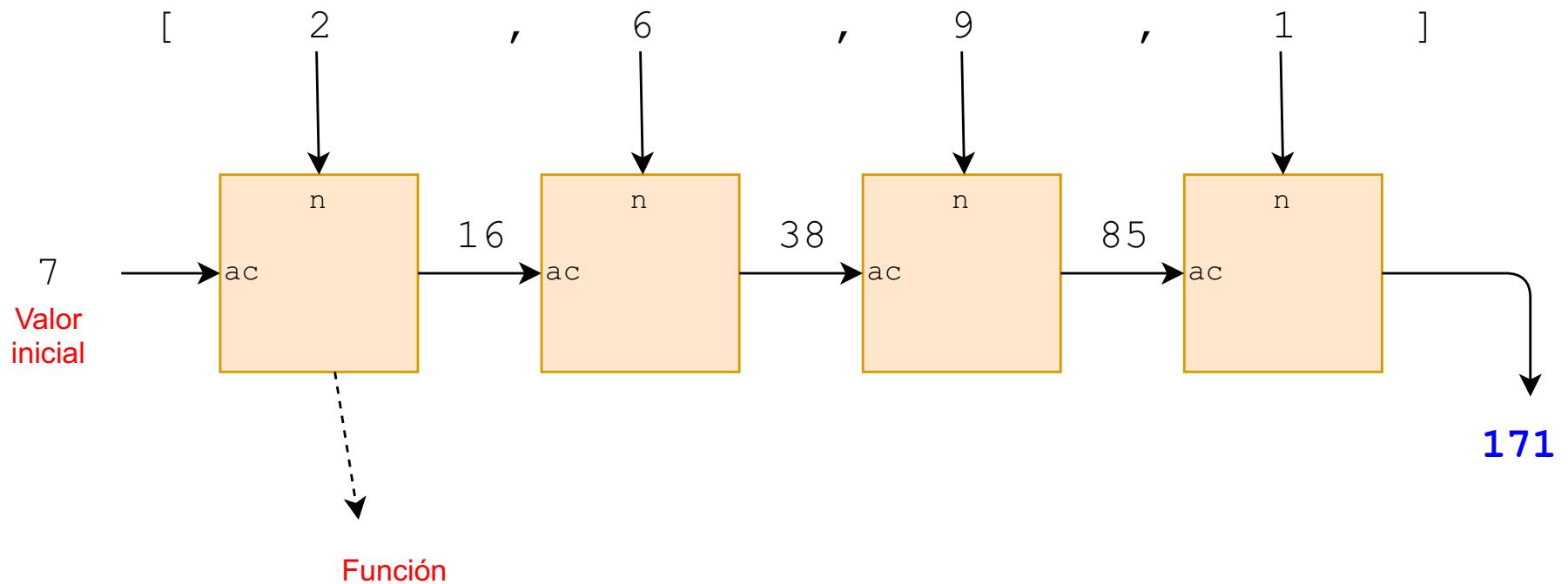
Devuelve **true** si existe un elemento **x** en el array tal que **f(x)** devuelva **true**.

FUNCIONES DE REDUCCIÓN (II)

- `reduce(f, [valorInicial]) → f(ac,v,i,a)`

Recorre el array de izquierda a derecha, acumulando un valor durante el recorrido.

```
let a = [2, 6, 9, 1];  
  
console.log(  
    "Valor final: " +  
    a.reduce((ac, n) => 2 * ac + n, 7)); // → 171
```

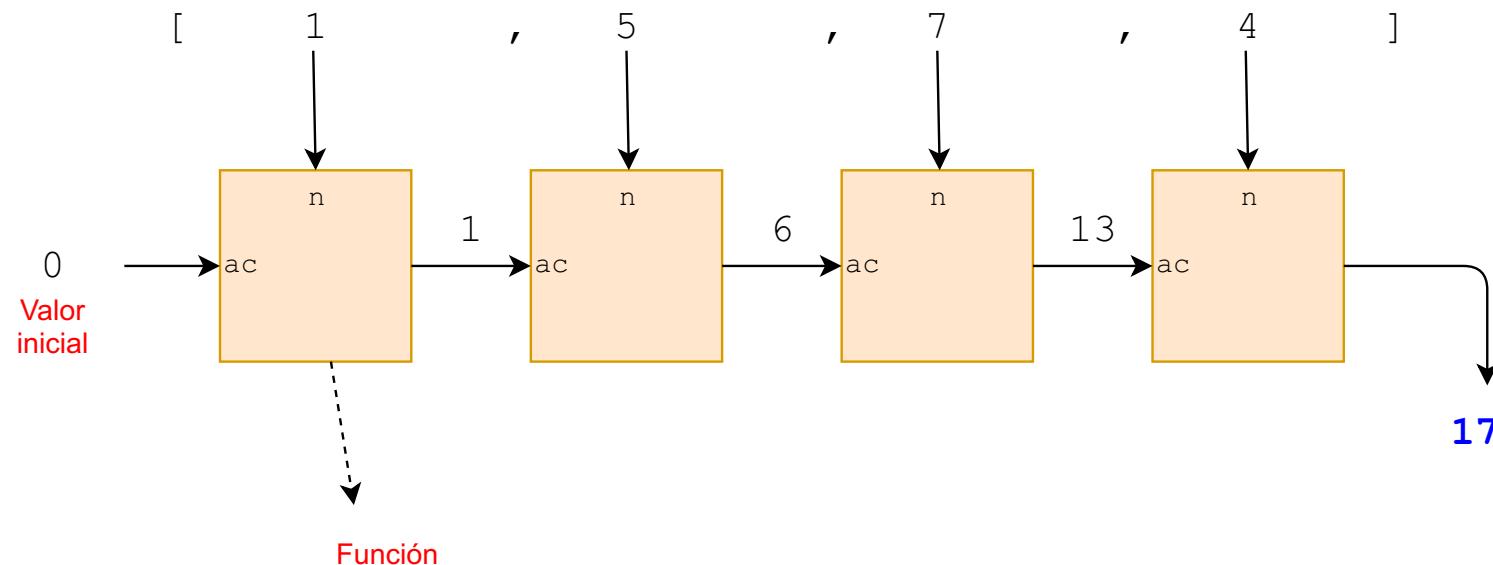


```

function(ac, n) {
    return 2 * ac + n;
}
    
```

Suma de los elementos de un array

```
[1, 5, 7, 4].reduce((ac, n) => ac + n, 0)
```



```
function(ac, n) {  
    return ac + n;  
}
```

Multiplicación de los elementos de un array

```
[1, 5, 7, 4].reduce((ac, n) => ac * n, 1)
```

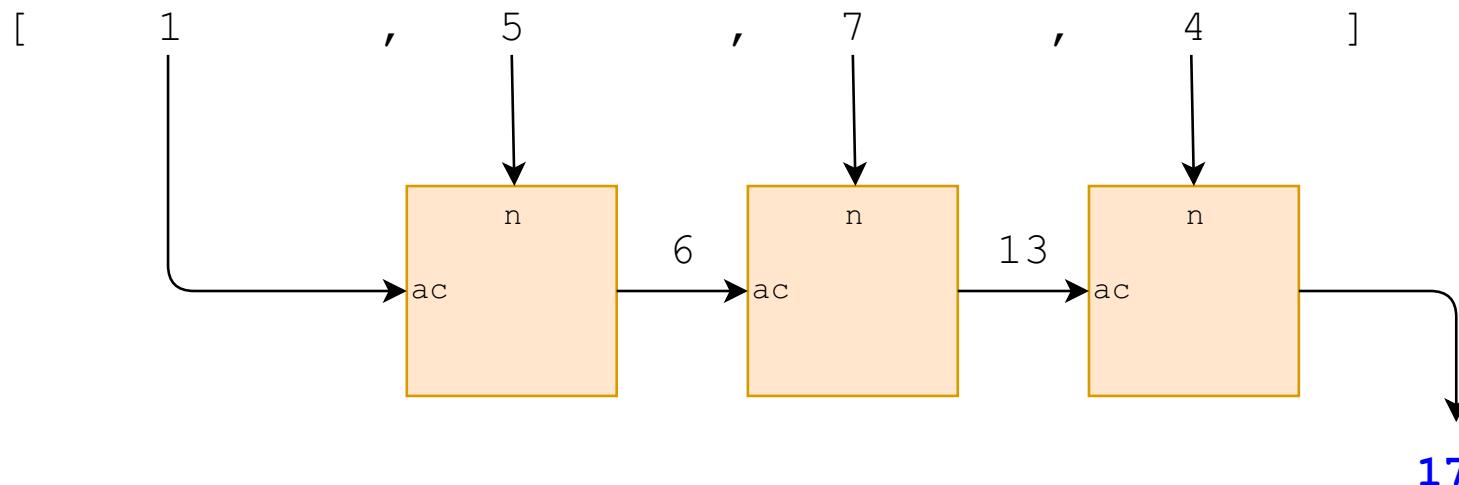
Máximo de los elementos de un array

```
[6, 1, 4, 3, 7].reduce((ac, x) => Math.max(ac, x), -Infinity);
```

reduce(f)

Si no se indica valor inicial, se supone que éste es el primer elemento del array.

```
[1, 5, 7, 4].reduce((ac, x) => ac + x)
```



VARIANTES

`reduceRight(f, [valorInicial])`

Realiza lo mismo que `reduce`, pero recorriendo el vector de derecha a izquierda.

EXPRESIONES REGULARES

DEFINICIÓN

Una expresión regular es un **patrón** que representa una o varias cadenas de texto.

Se utilizan para operaciones de búsqueda y sustitución de texto.

Por ejemplo, la expresión regular **[A-Z][0-9]{3}** denota el conjunto de cadenas que comienzan por una letra mayúscula y van seguidas por tres dígitos

Ejs: A324, F983, etc.

Ver: [Lenguaje de expresiones regulares](#)

ESCRITURA DE EXPRESIONES REGULARES

Una expresión regular puede contener dos tipos de caracteres:

- **Caracteres simples:** se representan a sí mismos.

Ej: la expresión regular */abc/* representa la cadena *abc*.

- **Metacaracteres:** tienen un significado especial.

Ej: la expresión regular */a+/* representa las cadenas *a, aa, aaa, aaaa*, etc.
El símbolo *+* es un metacaracter que significa *una o más ocurrencias*.

ALGUNOS METACARACTERES

- + Una o más ocurrencias de la expresión que lo precede.

Ej: $/a^+ / \rightarrow a, aa, aaa, \dots$

Concuerda con a en "café", con todas las a s en "holaaaaaaaa" y con nada en "te".

- * Cero o más ocurrencias de la expresión que lo precede.

Ej: $/ab^*/ \rightarrow a, ab, abbbb, \dots$

Concuerda con a en "café", con ab en "cable" y con nada en "lejos".

?

Cero o una ocurrencias de la expresión que lo precede.

Ej: $/a?bc?/ \rightarrow b, ab, bc, abc$

Concuerda con b en "bar", con abc en "el diario abc" y con nada en "sol".

.

Cualquier caracter excepto salto de línea.

Ej: $.^* \rightarrow$ una línea entera.

{n}

n ocurrencias de la expresión que lo precede.

Ej: $/a\{2\}/ \rightarrow ee$

Concuerda con ee en "releer" y con las dos primeras ocurrencias de e en "beeeeeeee".

\ Si precede a un metacaracter hace que pierda su significado especial.

Ej: /*/ → repesenta el caracter "*"

Ej: /\./ → repesenta el caracter ":"

Ej: /\\" → repesenta el caracter "\\"

Si precede a ciertos caracteres simples, los transforma en caracteres especiales.

Ej: /\n/ → salto de línea

Ej: /\t/ → tabulador

Ej: /\s/ → espacio en blanco (incluye también tabulador, salto de línea, etc)

[xyz] Uno de los elementos de los indicados entre corchetes. Con el símbolo "-" se expresan rangos.

Ej: */[abc]/* → a, b o c

Ej: */[a-zA-Z]/* → cualquier letra minúscula o mayúscula

\w Caracter alfanumérico o guión bajo. Es equivalente a *[a-zA-Z0-9_]*.

Ej: */\w/* → Concuerda con a en "ab", con 5 en "5\$" y con _ en "%%"

Ej: */[a-zA-Z]/* → cualquier letra minúscula o mayúscula

\d Caracter numérico. Es equivalente a *[0-9]*.

CREACIÓN DE EXPRESIONES REGULARES

En JavaScript las expresiones regulares se pueden crear de dos maneras:

- De manera literal: patrón delimitado por el símbolo /

```
var expr = /[A-Z][0-9]{3}/;
```

Este método es adecuado cuando la expresión regular es constante.

- Utilizando la clase RegExp

```
var expr = new RegExp('[A-Z][0-9]{3}');
```

Este método es adecuado cuando la expresión regular no es constante, por ejemplo, cuando proviene de un usuario.

ALGUNOS MÉTODOS DE LA CLASE RegExp

- `test(str)`: devuelve `true` si en la cadena `str` contiene una subcadena que encaja con la expresión, o `false` en caso contrario.

```
/[A-Z][0-9]{3}/.test("A655"); // → true
/[A-Z][0-9]{3}/.test("__A655__"); // → true
/[A-Z][0-9]{3}/.test("Otra cosa"); // → false

let idValido = /\b[a-zA-Z_][a-zA-Z0-9_]*\b/;
idValido.test("8cte"); // → false
idValido.test("_cte"); // → true
```

`\b` significa separador de palabra.

- **exec(str)**: busca en la cadena una subcadena que concuerde con la expresión y devuelve el texto encontrado, o **null** en caso contrario.

```
/\d{3}/.exec("123 456 789"); // → ["123", index: 0, ...]  
  
/\w+@\w+/.exec("abc"); // → null  
  
/\w+@\w+/.exec("Esto es un correo nombre_apellidos@correo");  
// → ["nombre_apellido@correo", index: 18,...]
```

A veces se puede dividir la expresión regular en varios grupos con el fin de saber qué parte de la cadena capturada corresponde a cada grupo.

Cada grupo va delimitado entre paréntesis (,)

La función `exec` nos permite desglosar cualquier cadena que ajuste con el patrón en sus distintos grupos.

Por ejemplo, la siguiente expresión:

`\d{4}-[A-Z]{3}`

ajusta con cualquier secuencia de cuatro dígitos (`\d` = dígito) que vaya seguida de un guión (`\-`) y tres letras mayúsculas.

Ejs: `0249-GSW`, `1934-HHG`, etc.

Si queremos poder separar la secuencia de dígitos de la de letras utilizamos dos grupos:

`(\d{4})\-([A-Z]{3})`

```
var regexp = /(\d{4})\-([A-Z]{3})/;  
var result = regexp.exec("Mi matrícula de coche es 8367-AWD");
```

La subcadena **8367-AWD** ajusta con el patrón **regexp**, pero **exec** nos permite saber qué fragmento de ésta ajusta con cada grupo

```
result[0] // → "8367-AWD" (Cadena completa)  
result[1] // → "8367"      (Primer grupo de captura)  
result[2] // → "AWD"       (Segundo grupo de captura)  
  
result.index // → 25      (Posición del ajuste dentro de la cadena)
```

MODIFICADORES DE EXPRESIONES REGULARES

Se colocan tras el delimitador / final de la expresión.

- i** No distingue entre mayúsculas y minúsculas.
- g** Ajuste global.
Permite encontrar varias ocurrencias en la misma cadena.
- m** Buscar a lo largo de varias líneas.

EXPRESIONES REGULARES Y MÉTODOS DE CADENAS

Es habitual el uso de expresiones regulares con métodos de cadenas para realizar operaciones de búsqueda y sustitución de texto.

- `match(regexp)`: devuelve todas las subcadenas que ajustan con la expresión regular `regexp` (si ésta contiene el modificador `g`) o solamente la primera (en caso contrario).

```
var str = "a aa aaa";
str.match(/a+/); // → ["a", index: 0, input:"a aa aaa", groups:undefined]
str.match(/a+/g); // → ["a", "aa", "aaa"]
str = "HolA";
str.match(/hola/); // → null
str.match(/hola/i); // → ["HolA", index:0, input:"HolA", groups:undefined]
```

- **search(regexp)**: devuelve el índice de la primera subcadena que ajuste con **regexp**, o -1 si no hay ninguna.

```
var str = "123 456 789";
str.search(/\d{3}/);      // → 0
str.search(/3\d{2}/);    // → -1
```

- **replace(regexp, nuevaCadena)**: reemplaza por **nuevaCadena** las subcadenas que ajusten con **regexp**.

```
var str = "123 456 789";
str.replace(/\d{3}/, "AAA");      // → AAA 456 789
str.replace(/\d{3}/g, "AAA");    // → AAA AAA AAA
```

ORIENTACIÓN A OBJETOS BASADA EN PROTOTIPOS

LENGUAJES OO BASADOS EN CLASES VS. BASADOS EN PROTOTIPOS

Hay dos tipos lenguajes orientados a objetos:

- Lenguajes OO **basados en clases** (Java).
- Lenguajes OO **basados en prototipos** (JavaScript).

Ambos modelos difieren en aspectos muy importantes:

- Elementos presentes en el modelo.
- Mecanismo de creación de objetos.
- Herencia.
- Ampliación de las propiedades de los objetos
- ...

ELEMENTOS PRESENTES EN EL MODELO

OO-clases

Clases y
objetos (instancias).

OO-prototipos

Sólo objetos.
Todos tienen la misma
categoría.
Algunos objetos son **prototipos**
de otros.

MECANISMO DE CREACIÓN DE OBJETOS

OO-clases

OO-prototipos

Constructores.

Creación literal.

Creación a partir de otro
objeto (prototipo).

CREACIÓN LITERAL

Se utiliza el mecanismo que se muestra en la figura.

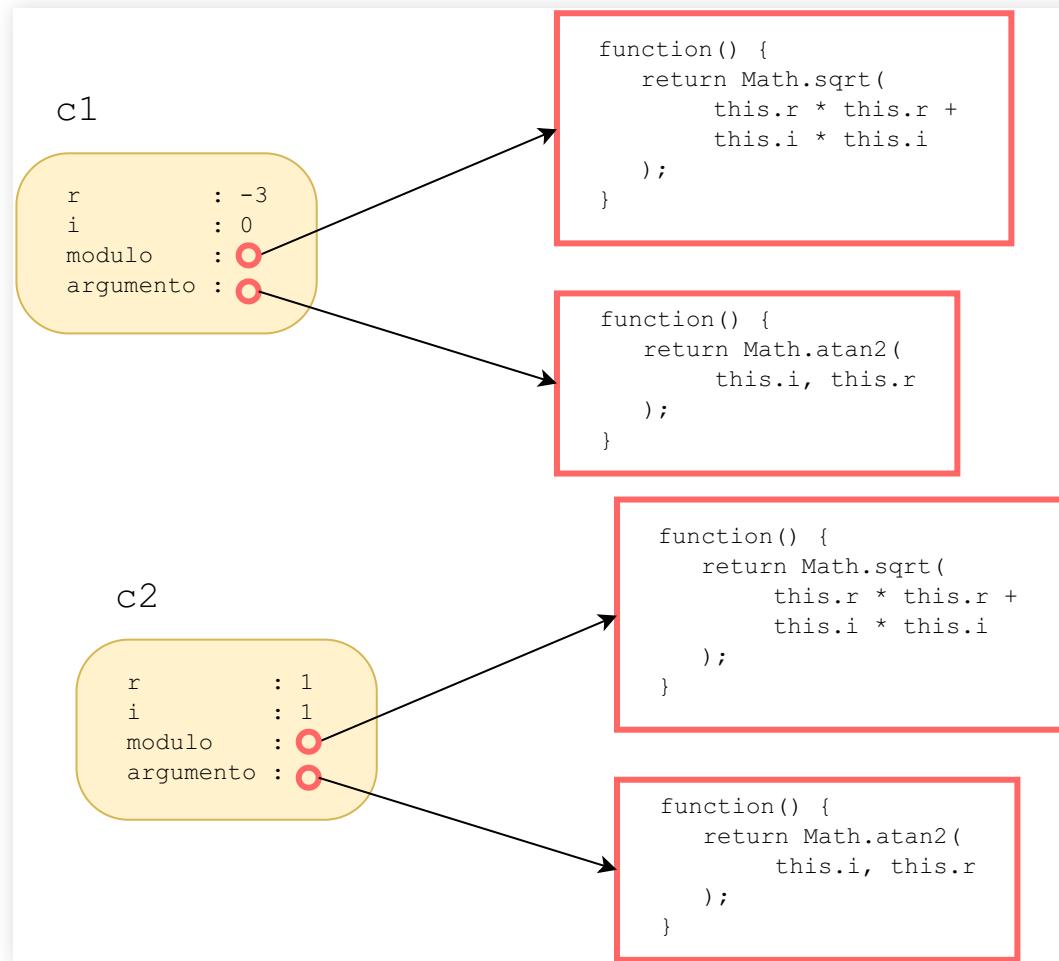
```
let c = {  
    r:1,  
    i:1,  
    modulo: function() {  
        return Math.sqrt(this.r*this.r +  
                        this.i*this.i);  
    },  
    argumento: function() {  
        return Math.atan2(this.i,this.r);  
    }  
};
```

Se crea el objeto **c** que contiene **sus** atributos(**r** e **i**) y **sus** métodos (**modulo** y **argumento**).

Se puede encapsular la creación de objetos mediante funciones constructoras.

```
function construirComplejo(real, imag) {  
    return {  
        r : real,  
        i : imag,  
  
        modulo: function() {  
            return Math.sqrt(this.r * this.r + this.i * this.i);  
        },  
  
        argumento: function() {  
            return Math.atan2(this.i, this.r);  
        }  
    }  
}  
  
var c1 = construirComplejo(-3, 0);  
console.log(c1.argumento()); // → 3.141592653589793  
var c2 = construirComplejo(1, 1);  
console.log(c2.modulo()); // → 1.4142135623730951
```

Problema: duplicidad de objetos función para cada objeto.



¿No podrían **c1** y **c2** compartir los métodos?

Possible solución:

```
function moduloComplejo() {  
    return Math.sqrt(this.r * this.r + this.i * this.i);  
}  
  
function argumentoComplejo() {  
    return Math.atan2(this.i, this.r);  
}  
  
function construirComplejo(real, imag) {  
    return {  
        r : real,  
        i : imag,  
        modulo: moduloComplejo,  
        argumento: argumentoComplejo  
    }  
}
```

c1

r : -3
i : 0
modulo : ○
argumento : ○

moduloComplejo

```
function() {  
    return Math.sqrt(  
        this.r * this.r +  
        this.i * this.i  
    );  
}
```

c2

r : 1
i : 1
modulo : ○
argumento : ○

argumentoComplejo

```
function() {  
    return Math.atan2(  
        this.i, this.r  
    );  
}
```

Si se añade un método nuevo a **c1**.

```
var c1 = construirComplejo(-3, 0);
var c2 = construirComplejo(1, 1);

// ...

c1.coordenadasPolares = function() {
    console.log(`(${this.modulo()}, ${this.argumento()})`);
}
```

Este método existe solamente dentro de **c1**.

¿Existe alguna manera de añadir un método simultáneamente a todos los objetos que hubiesen sido creados mediante **construirComplejo**?

Sí. Se puede hacer mediante **prototipos**.

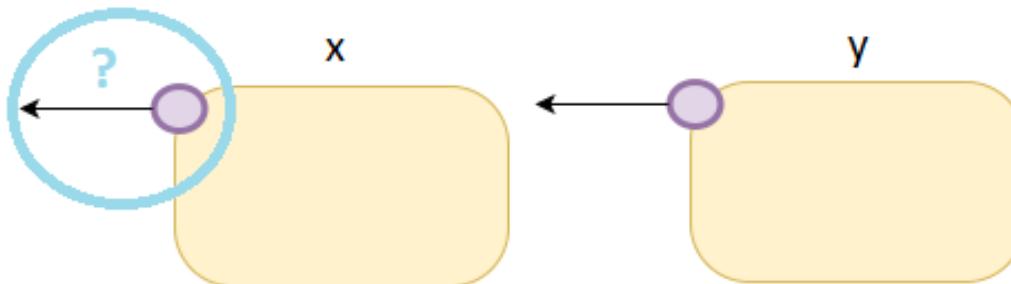
CREACIÓN A PARTIR DE OTRO OBJETO (PROTOTIPO)

Se utiliza la función `Object.create()`

La siguiente sentencia:

```
let y = Object.create(x);
```

crea un objeto `y` que tiene a `x` como prototipo.



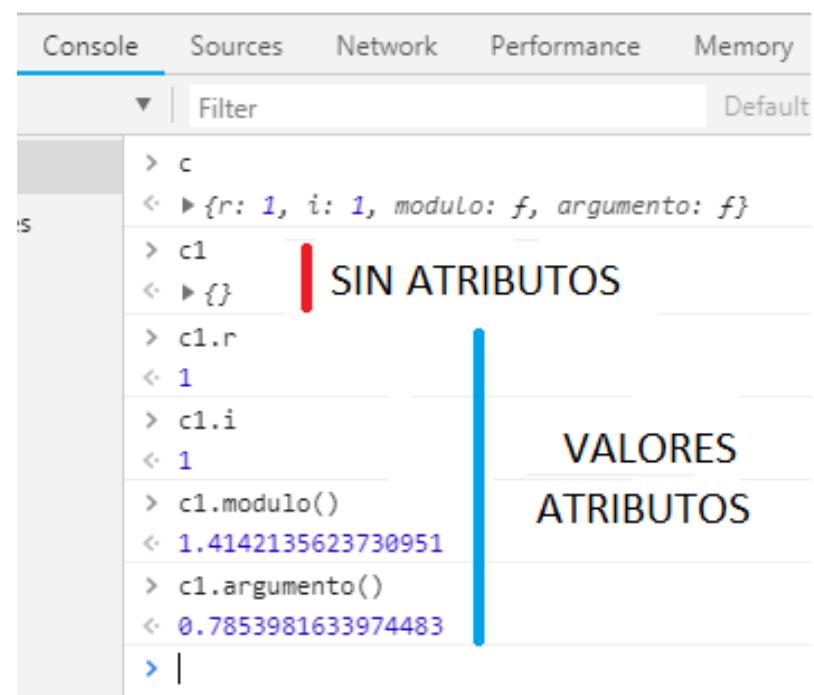
El objeto `x` tendrá su prototipo.

Ejemplo

Creación del objeto **c1** a partir del objeto **c**.

```
let c = {  
    r:1,  
    i:1,  
    modulo: function() {  
        return Math.sqrt(this.r*this.r  
                        this.i*this.i  
    },  
    argumento: function() {  
        return Math.atan2(this.i,this.r)  
    }  
};  
  
let c1 = Object.create(c);
```

Visualización de los objetos en la consola.

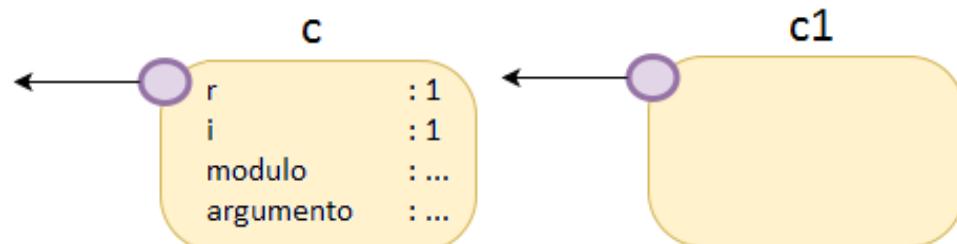


The screenshot shows the browser's developer tools Console tab. At the top, there are tabs for Console, Sources, Network, Performance, and Memory. Below the tabs is a filter input field with the value "Default". The main area displays the following interactions:

- > c
- < ↵ {r: 1, i: 1, modulo: f, argumento: f}
- > c1
- < ↵ {} **SIN ATRIBUTOS**
- > c1.r **VALORES**
- < ↵ 1
- > c1.i **ATRIBUTOS**
- < ↵ 1
- > c1.modulo()
- < ↵ 1.4142135623730951
- > c1.argumento()
- < ↵ 0.7853981633974483
- > |

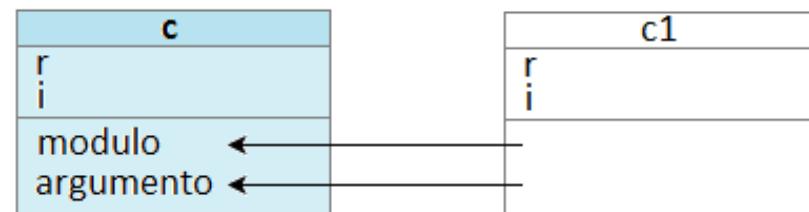
Creación de un objeto en JavaScript con `Object.create()`.

```
let c1 = Object.create(c)
```



Creación de un objeto en un lenguaje basado en clases.

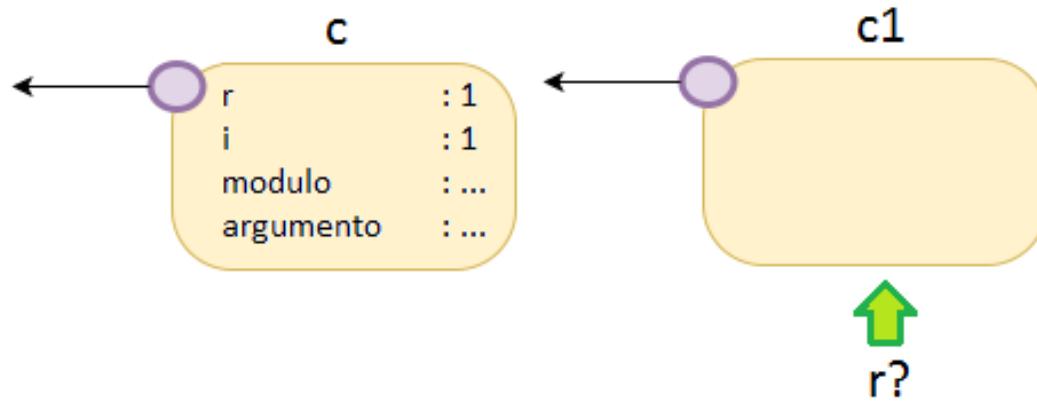
```
class c {  
    r ...  
    i ...  
    modulo ...  
    argumento ...  
}  
c1 = new c(...)
```



BÚSQUEDA DE ATRIBUTOS

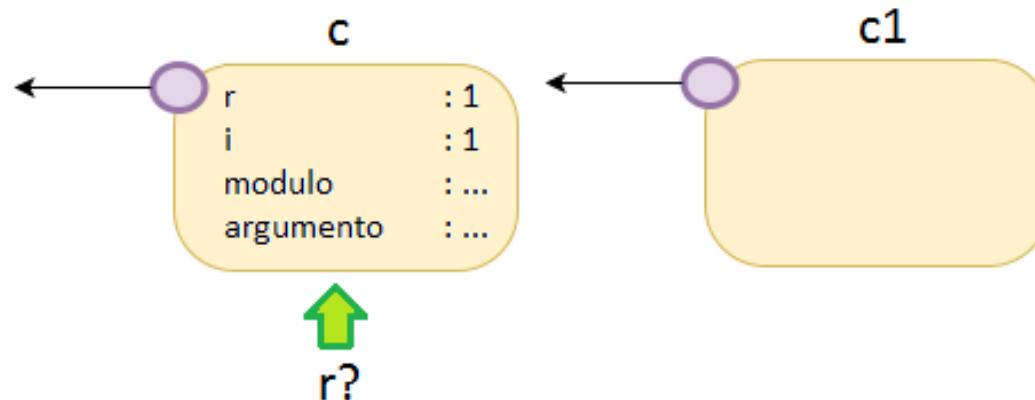
¿Qué ocurre cuando se accede al atributo **r** de **c1**?

En primer lugar se intenta buscar **r** dentro de **c1**.



r no se encuentra en **c1** porque como se ha visto en la consola, el objeto **c1** no tiene atributos. Se sigue buscando...

... dentro del prototipo de `c1`, que es `c`.

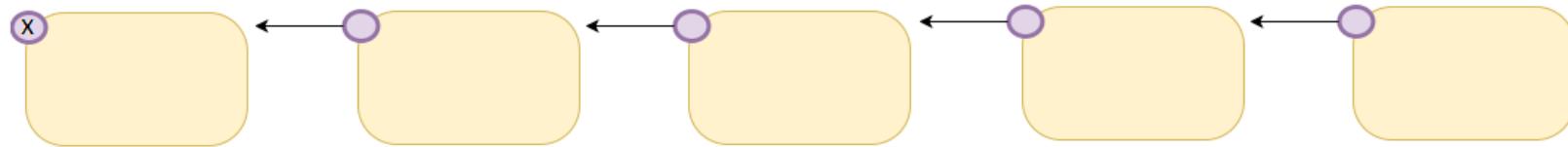


El atributo `r` se encuentra en `c` y se detiene la búsqueda.

Si no se hubiera encontrado, la búsqueda continuaría por la **cadena de prototipos** hasta que:

- Se encuentre el atributo en algún objeto de la cadena.
- Se llegue al final de la cadena. En ese caso la expresión `c1.r` se evaluaría a `undefined`.

- Cada objeto mantiene un enlace a otro objeto que es su **prototipo**.
- Ese objeto prototipo tiene su propio prototipo, y así sucesivamente formando una **cadena de prototipos**.
- Hasta que se alcanza un objeto cuyo prototipo es **null**, y actúa como raíz de la cadena de prototipos.



- El acceso a un atributo de un objeto se hace partiendo del propio objeto y ascendiendo por la cadena de prototipos.

En el ejemplo previo, `c1` no tiene **atributos propios**.

Para que un objeto tenga como propio un atributo existente en su cadena de prototipos, hay que crear dicho atributo asignándole un valor. Este valor "apantalla" al presente en la cadena de prototipos.

```
c1.i = 8;  
c1.r = 0;
```

The screenshot shows a browser's developer tools Console tab. On the left, there is a code editor-like area with the following code:

```
c1.i = 8;  
c1.r = 0;
```

To the right is the console output:

Console Output	Annotations
> c	
< ▶ {r: 1, i: 1, modulo: f, argumento: f}	
> c1.r = 8;	
< 8	ASIGNACIÓN DE VALORES
> c1.i = 0;	
< 0	
> c1	
< ▶ {r: 8, i: 0}	ATRIBUTOS "PROPIOS"

A vertical green line highlights the assignment `c1.r = 8;`. To its right, the text "ASIGNACIÓN DE VALORES" is written vertically. Another vertical green line highlights the assignment `c1.i = 0;`. To its right, the text "ATRIBUTOS 'PROPIOS'" is written vertically.

AÑADIR ATRIBUTOS AL PROTOTIPO

Se define el objeto `circulo` y se usa como prototipo para la creación de los objetos `c1` y `c2`.

```
let circulo = {  
    centro: { x: 0, y: 0 },  
    radio: 5  
};  
  
let c1 = Object.create(circulo);  
let c2 = Object.create(circulo);
```

Si se añaden nuevos atributos a `circulo`, los objetos `c1` y `c2` los heredan automáticamente.

```
circulo.grosorBorde = 2;  
  
console.log(c1.grosorBorde); // → 2  
console.log(c2.grosorBorde); // → 2
```

```

let circulo = {
    centro: {x: 0, y: 0},
    radio: 5
};

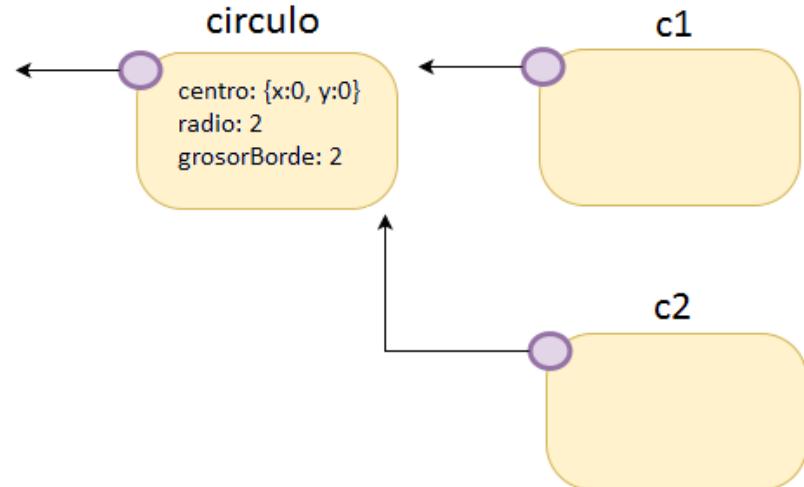
let c1 = Object.create(circulo);
let c2 = Object.create(circulo);

circulo.grosorBorde = 2;

console.log(c1.grosorBorde); // → 2
console.log(c1.grosorBorde); // → 2

console.log(c1.radio); // → 5
console.log(c2.centro); // → {x:0, y:0}

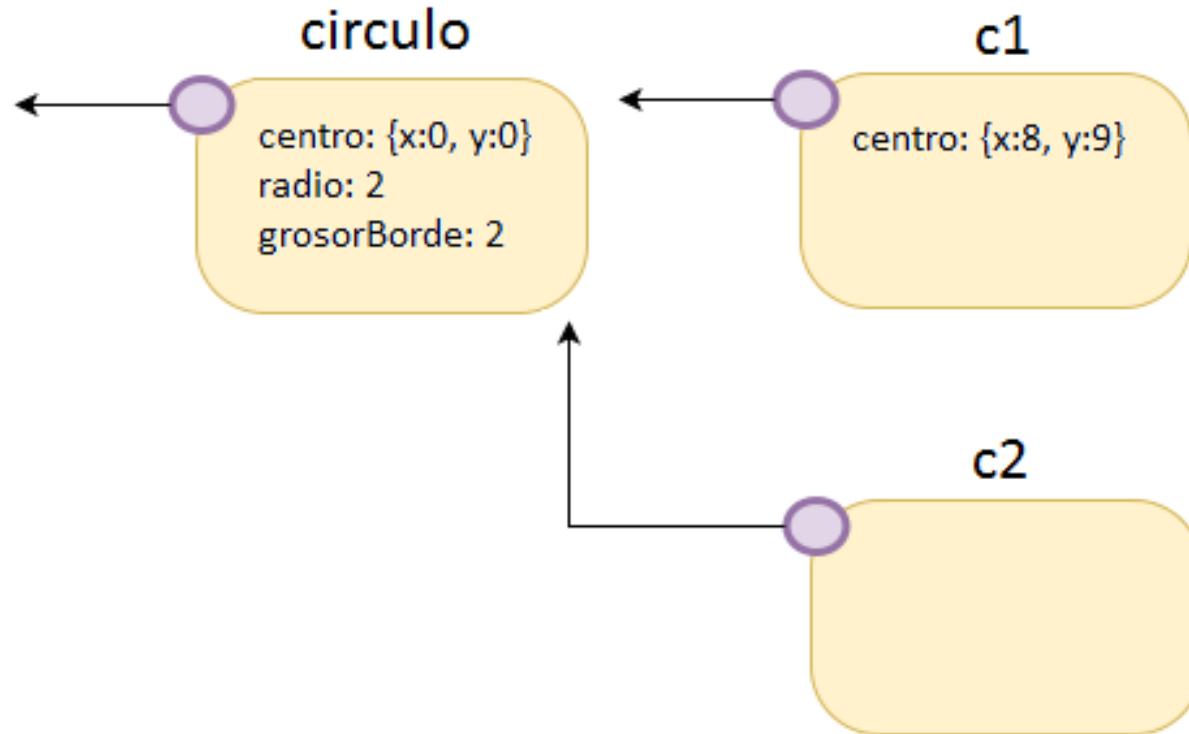
```



Los objetos `c1` y `c2` no tienen atributos propios.
 Cualquier acceso a los atributos desencadena la búsqueda
 en la cadena de prototipos.

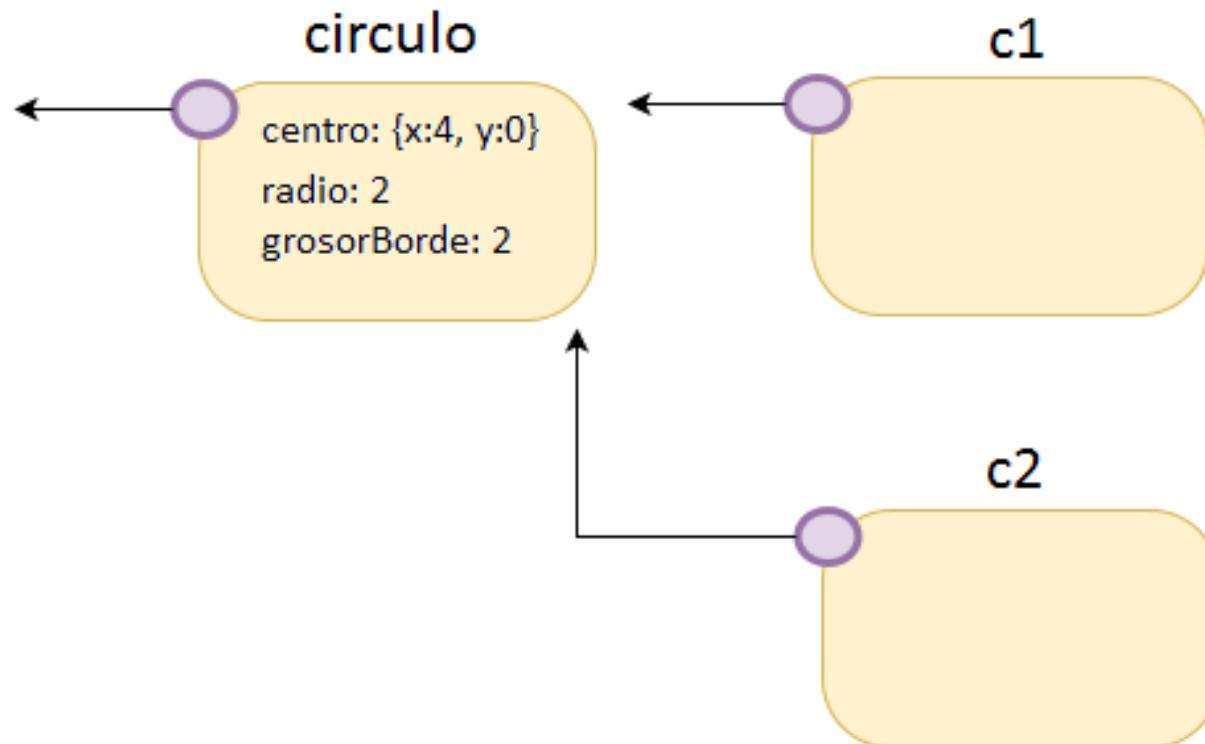
En el objeto **c1** se asigna valor al atributo **centro**.

```
c1.centro = {x:8, y:9};  
  
console.log(c1.centro); // → {x:8, y:9}  
console.log(c2.centro); // → {x:0, y:0}
```



En el objeto **c1** se asigna valor al atributo **centro.x**.

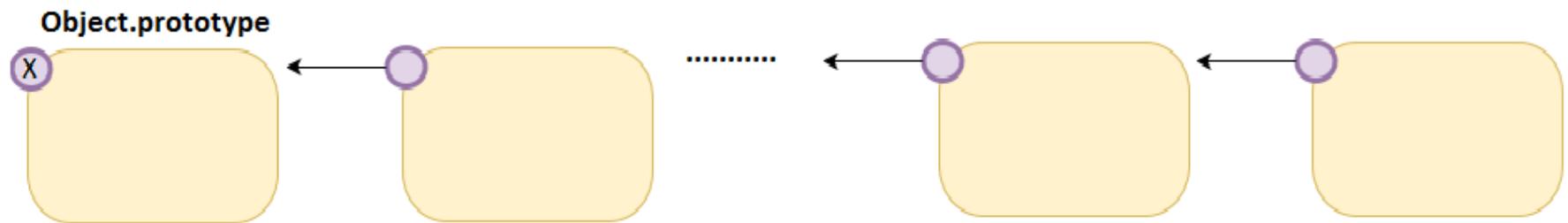
```
c1.centro.x = 4;  
  
console.log(circulo.centro); // → {x:4, y:0}  
console.log(c1.centro); // → {x:4, y:0}  
console.log(c2.centro); // → {x:4, y:0}
```



EL OBJETO `Object.prototype`

En la raíz de la cadena de prototipos se encuentra el objeto `Object.prototype`.

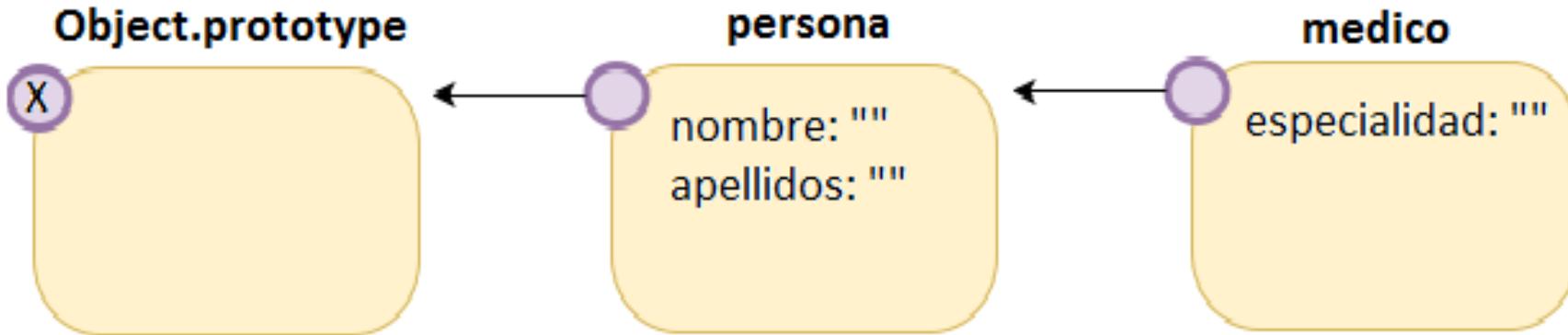
El prototipo de `Object.prototype` es `null`



LA FUNCIÓN `Object.getPrototypeOf()`.

Permite consultar el prototipo de un objeto.

```
let persona = {  
    nombre: "",  
    apellidos: ""  
}  
  
let medico = Object.create(persona);  
medico.especialidad = "";  
  
console.log(Object.getPrototypeOf(medico) === persona); // → true  
console.log(Object.getPrototypeOf(persona) === Object.prototype); // -
```



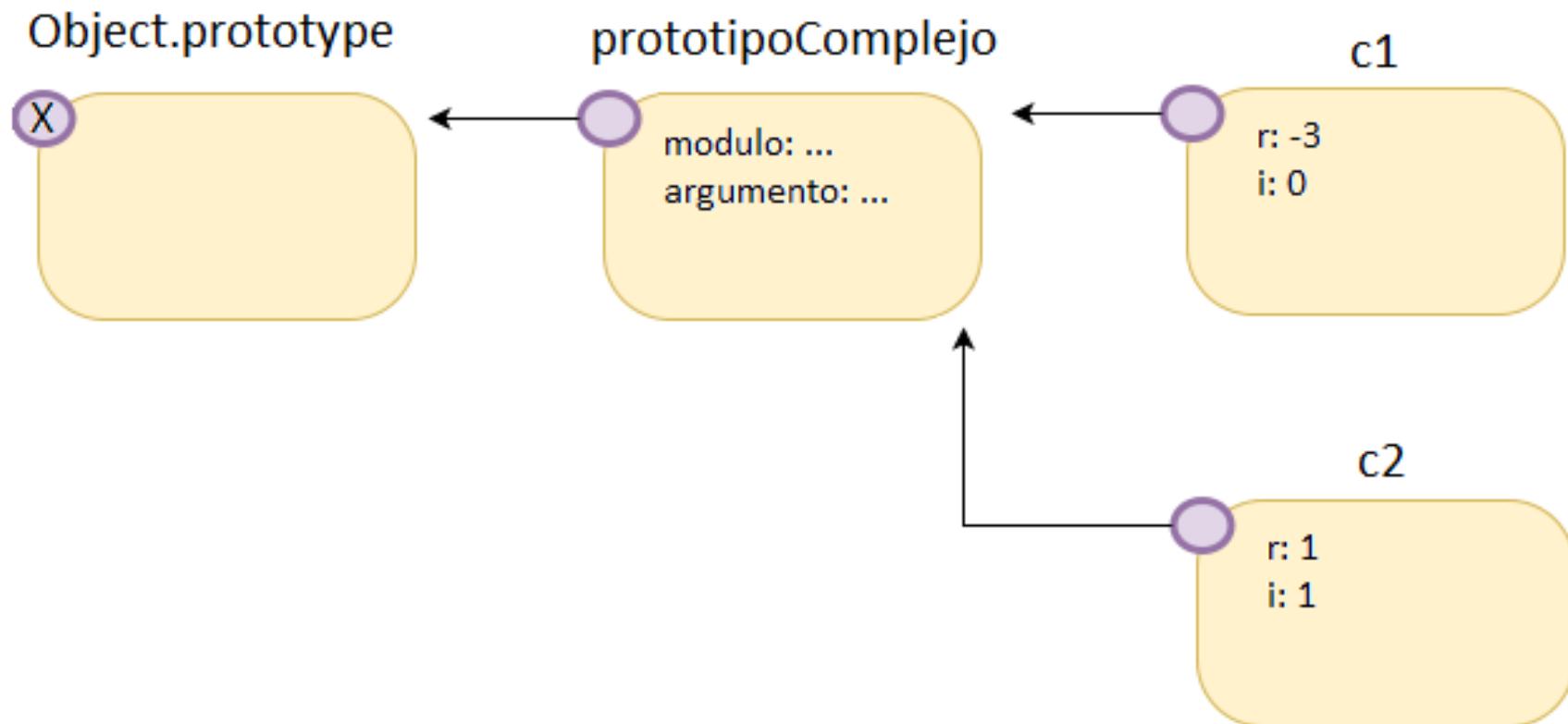
EJEMPLO DE CREACIÓN DE OBJETOS "PARECIDOS" A INSTANCIAS

```
// SE CREA UN OBJETO-PROTOTIPO QUE CONTIENE LOS MÉTODOS
let prototipoComplejo = {
    modulo: function() {
        return Math.sqrt(this.r * this.r + this.i * this.i);
    },
    argumento: function() {
        return Math.atan2(this.i, this.r);
    }
};

// SE PROGRAMA UNA FUNCIÓN QUE CREA UN OBJETO A PARTIR DEL PROTOTIPO
// Y LE CREA SUS PROPIOS ATRIBUTOS
function construirComplejo(real, imag) {
    var resultado = Object.create(prototipoComplejo);
    resultado.r = real;
    resultado.i = imag;
    return resultado;
}
```

```
var c1 = construirComplejo(-3, 0);
var c2 = construirComplejo(1, 1);
```

La cadena de prototipos es la siguiente:



El prototipo tiene los métodos y los objetos los atributos.

Todas las funciones/métodos que se añadan al prototipo estarán disponibles automáticamente para todos los objetos que hayan sido creados previamente por `construirComplejo()`.

```
// Añadimos una nueva función al prototipo:  
  
prototipoComplejo.mostrarEnPolares = function() {  
    console.log(`(${this.modulo()}, ${this.argumento()})`);  
}  
  
c1.mostrarEnPolares();  
// → (3, 3.141592653589793)  
  
c2.mostrarEnPolares();  
// → (1.4142135623730951, 0.7853981633974483)
```

MÉTODOS DEL OBJETO `Object.prototype`

Como el objeto `Object.prototype` es la raíz de la cadena de prototipos, todos los objetos pueden invocar y sobreescribir sus métodos predefinidos, como por ejemplo:

- `toString()`
- `valueOf()`
- `isPrototypeOf()`
- `hasOwnProperty()`
- `[+]`

EL MÉTODO `toString()`

Este método es invocado automáticamente cada vez que un objeto es referido en un contexto en el que se espera una cadena.

```
prototipoComplejo.toString = function() {  
    return "(" + this.r + "," + this.i + ")";  
}  
  
let c3 = construirComplejo(1, 3);  
  
alert(c3); // muestra una ventana con el texto (1, 3)
```

También se puede invocar al método de manera explícita.

```
console.log(c3.toString()); // → (1, 3)
```

EL MÉTODO valueOf()

Este método es invocado automáticamente cada vez que un objeto se encuentra en una posición en la que se espera un valor primitivo (un número, un booleano, etc).

```
let prototipoPersona = {  
    valueOf: function() {  
        return this.edad;  
    }  
};  
  
function construirPersona(nom, ap, e) {  
    var resultado = Object.create(prototipoPersona);  
    resultado.nombre = nom;  
    resultado.apellido = ap;  
    resultado.edad = e;  
    return resultado;  
}  
var p1 = construirPersona("Juan", "Gómez", 15);  
var p2 = construirPersona("Ana", "Torres", 18);  
  
console.log(p1+p2); // → 33
```

EL MÉTODO `hasOwnProperty()`

Este método permite conocer si un objeto tiene un atributo propio o por el contrario dicho atributo está en la cadena de prototipos.

```
let circulo = {  
    centro: {x: 0, y: 0},  
    radio: 5  
};  
  
let c1 = Object.create(circulo);  
let c2 = Object.create(circulo);  
  
console.log(c1.hasOwnProperty("radio")); // → false  
  
c1.radio = 10;  
console.log(c1.hasOwnProperty("radio")); // → true
```

SINTAXIS DE CREACIÓN DE OBJETOS A PARTIR DE ES6

Antes de ES6, un patrón común para simular las **clases** de otros lenguajes de programación es:

- Tener un objeto prototipo que almacene los métodos de la clase.
- Tener una función que construya las instancias de la clase, enlazándolas con el prototipo.

ES6 incorpora una nueva sintaxis para la creación de objetos, denominada "syntactic sugar" porque asemeja la creación de objetos a los lenguajes de clases pero no cambia el modelo de prototipado.

CLASES

```
class Complejo {  
    constructor(real, imag) {  
        this.r = real;  
        this.i = imag;  
    }  
  
    modulo() {  
        return Math.sqrt(this.r * this.r + this.i * this.i);  
    }  
  
    argumento() {  
        return Math.atan2(this.i, this.r);  
    }  
}
```

Esta declaración crea un objeto llamado **Complejo.prototype** que almacena el **constructor()** y los métodos **modulo()** y **argumento()**.

Se crean instancias mediante el operador **new**.

```
let c1 = new Complejo(-3, 0);
let c2 = new Complejo(1, 1);
```

Esto hace que los objetos **c1** y **c2** tengan a
Complejo.prototype como prototipo.

```
console.log(c1.modulo()); // → 3
console.log(c2.modulo()); // → 1.4142135623730951
```

Complejo.prototype

modulo:

```
function() {  
    return Math.sqrt(  
        this.r * this.r +  
        this.i * this.i  
    );  
}
```

argumento:

```
function() {  
    return Math.atan2(  
        this.i, this.r  
    );  
}
```

c1

r : -3
i : 0

c2

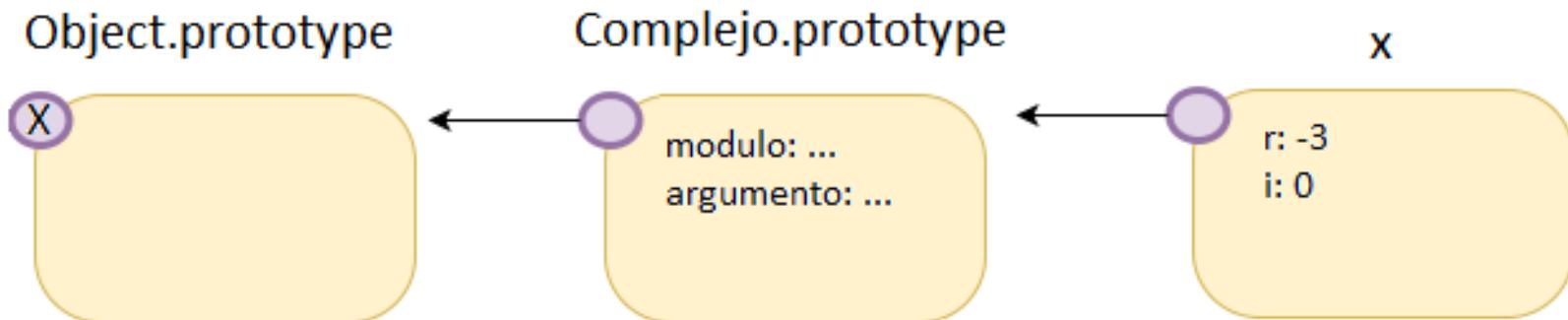
r : 1
i : 1

EL OPERADOR `instanceof`

La expresión `x instanceof C` se evalúa a `true` si el objeto `C.prototype` es alcanzable ascendiendo desde `x` en la cadena de prototipos:

```
var x = new Complejo(-3, 0);

console.log(x instanceof Complejo); // → true
console.log(x instanceof Object);   // → true
console.log(x instanceof Number);  // → false
```



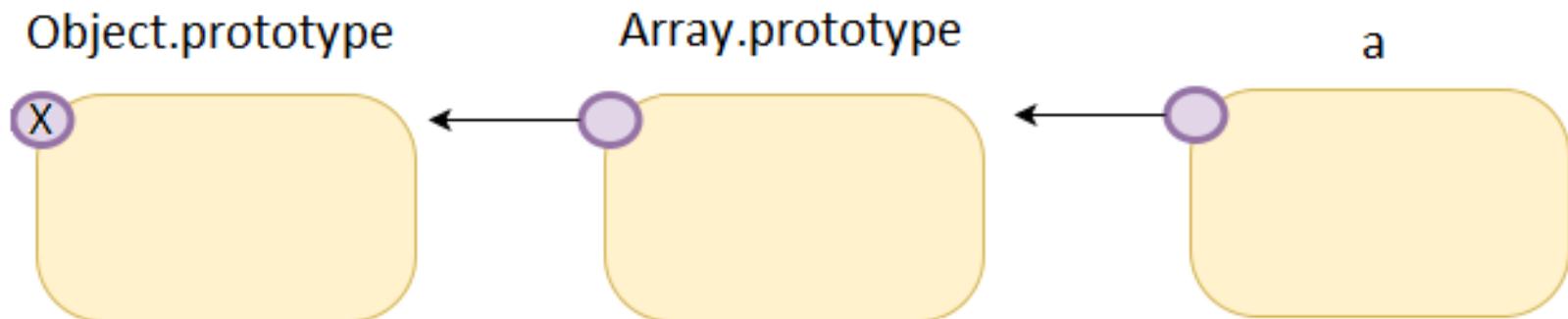
EJEMPLO

```
let a = new Array(3);

console.log(a instanceof Array);          // → true
console.log(a instanceof Object);         // → true

console.log(Object.getPrototypeOf(a) === Array.prototype); // → true
console.log(Object.getPrototypeOf(a) === Object.prototype); // → false

console.log(Array.prototype.isPrototypeOf(a)); // → true
console.log(Object.prototype.isPrototypeOf(a)); // → true
```



FUNCIONES get/set

- Permiten definir métodos cuya invocación es sintácticamente similar a los atributos.
- Permiten asegurar la calidad de los datos.

MÉTODOS INVOCADOS COMO ATRIBUTOS (get)

```
class Complejo {  
  
constructor(real=1, imag=1) {  
    this.r = real;  
    this.i = imag;  
}  
  
modulo() {  
    return Math.sqrt(this.r * this.r  
                    + this.i * this.i)  
}  
  
let c= new Complejo(1,0);  
console.log(c.modulo()); // → 1
```

```
class Complejo {  
  
constructor(real=1, imag=1) {  
    this.r = real;  
    this.i = imag;  
}  
  
get modulo() {  
    return Math.sqrt(this.r * this.r  
                    + this.i * this.i)  
}  
  
let c= new Complejo(1,0);  
console.log(c.modulo); // → 1
```

MÉTODOS INVOCADOS COMO ATRIBUTOS (set)

```
class Complejo {  
  
constructor(real=1, imag=1) {  
    this.r = real;  
    this.i = imag;  
}  
  
escala(e) {  
    if (e>0) {  
        this.r = this.r*e;  
        this.i = this.i*e;  
    }  
}  
  
}  
  
let c= new Complejo(1,0);  
c.escala(2);  
console.log(c); // → {r:2, i:0}
```

```
class Complejo {  
  
constructor(real=1, imag=1) {  
    this.r = real;  
    this.i = imag;  
}  
  
set escala(e) {  
    if (e>0){  
        this.r = this.r*e;  
        this.i = this.i*e;  
    }  
}  
  
}  
  
let c= new Complejo(1,0);  
c.escala=2;  
console.log(c); // → {r:2, i:0}
```

CALIDAD DE LOS DATOS

```
class Persona {  
    constructor(nombre, edad) {  
        this.nombre = nombre;  
        this._edad = edad;  
    }  
    get edad() {  
        return this._edad;  
    }  
    set edad(newEdad) {  
        if (newEdad >= 0) {  
            this._edad = newEdad;  
        }  
    }  
}
```

```
let p = new Persona("Elena", 23);  
p.edad = -2; // No modifica la edad  
console.log(p.edad); // Imprime: 23
```

Existe un convenio tácito por el que considera que los atributos que comienzan por _ son "privados".

MÉTODOS ESTÁTICOS

Son métodos de clase. Van precedidos por **static**.

```
class Complejo {  
    // ...  
  
    static desdePolar(mod, arg) {  
        var real = mod * Math.cos(arg),  
            imag = mod * Math.sin(arg);  
        return new Complejo(real, imag);  
    }  
}
```

```
let c2 = Complejo.desdePolar(1, Math.PI / 4);  
console.log(c2.r); // → 0.7071067811865476
```

Es posible añadir métodos estáticos una vez declarada la clase:

```
Complejo.desdePolar = (mod, arg) => {  
    // ...  
}
```

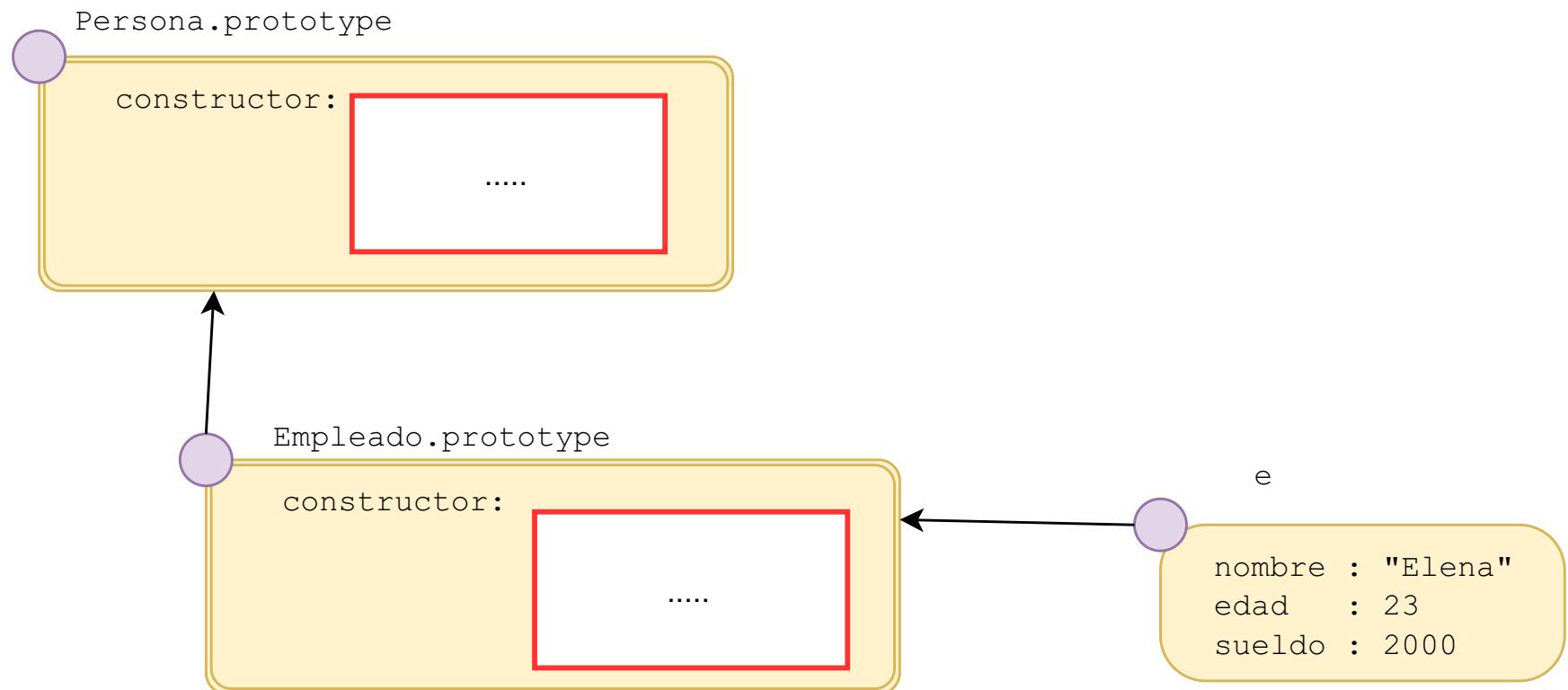
HERENCIA

Se permite herencia simple, al igual que en Java.

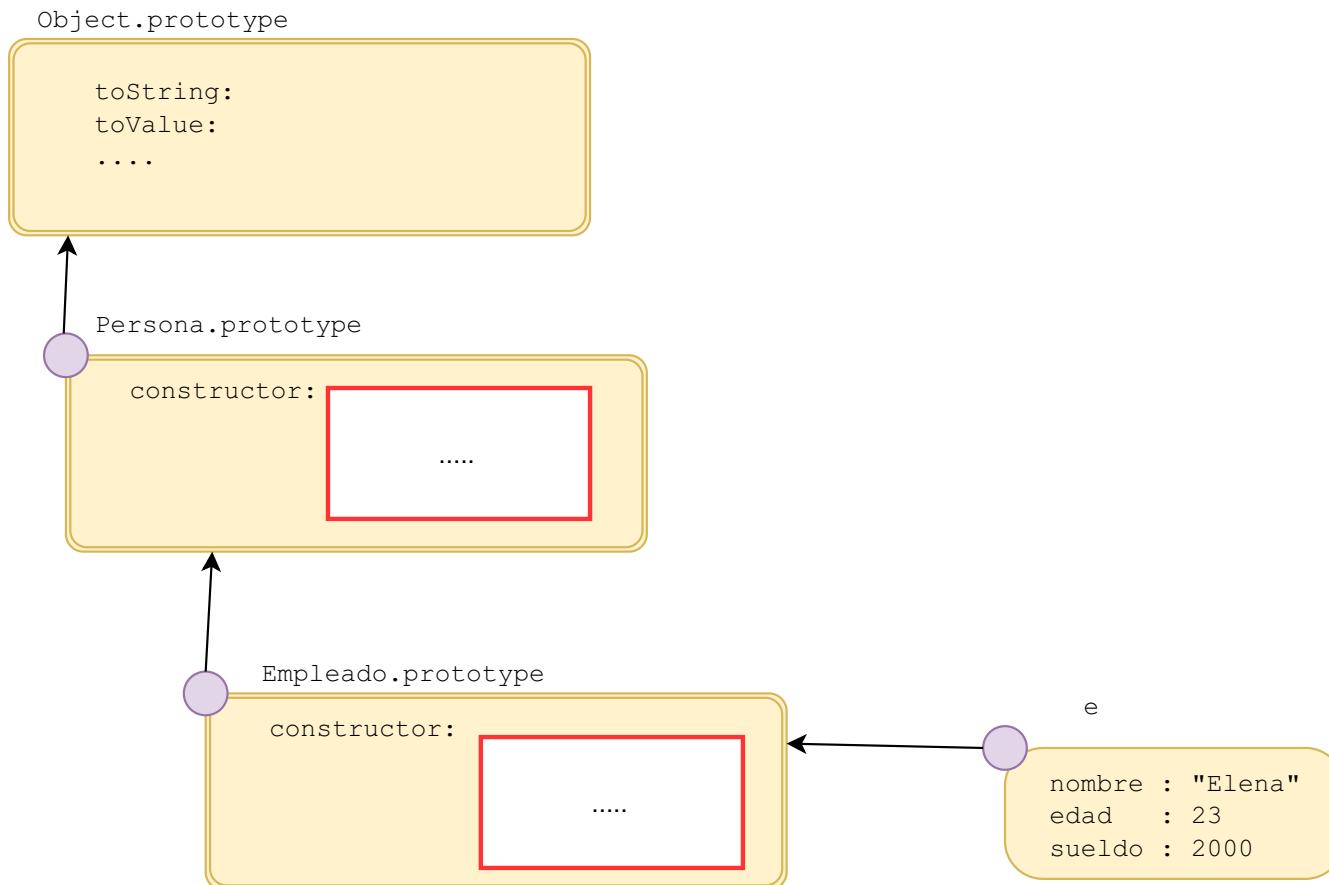
```
class Persona {  
    constructor(nombre, edad) { ... }  
    ...  
}  
  
class Empleado extends Persona {  
    constructor(nombre, edad, sueldo) {  
        super(nombre, edad);  
        this.sueldo = sueldo;  
    }  
    ...  
}
```

```
let e = new Empleado("Elena", 23, 2000);  
console.log(e.edad); // → 23  
console.log(e.sueldo); // → 2000
```

Al decir que **Empleado** extiende a **Persona**, estamos encadenando el prototipo de persona con el de empleado.



Si no se indica cláusula **extends**, la clase hereda automáticamente de **Object**



CLASES ESTÁNDAR

CLASES ESTÁNDAR

El estándar de ECMAScript define las siguientes clases
(disponibles tanto en Node como en los navegadores)

- Manejo de expresiones regulares
- Manejo de fechas
- Trazas y *logging*
- Utilidades varias

LOS OBJETOS Date

Sirven para realizar operaciones con fechas y horas.

```
var ahora = new Date();
ahora.toString();
// → 'Fri Oct 14 2016 14:37:56 GMT+0200 (CEST)'
ahora.getFullYear();
// → 2016
ahora.getMonth();
// → 9
ahora.getSeconds();
// → 56

var fechaInicio = new Date(2016, 09, 26);
fechaInicio.toString();
// → 'Wed Oct 26 2016 00:00:00 GMT+0200 (CEST)'
```

Más información: [\[+\]](#)

EL OBJETO Math

Utilidades matemáticas varias:

- Constantes: `E`, `LN2`, `PI`, etc.
- Máximos y mínimos: `max`, `min`.
- Números aleatorios: `random`.
- Redondeo: `ceil`, `floor`, `trunc`, `round`, etc.
- Potencias: `pow`, `sqrt`, `cbrt`, etc.
- Trigonométricas: `sin`, `sinh`, `cos`, etc.
- Exponenciales y logarítmicas: `exp`, `log`, `log10`, etc.

EL OBJETO `console`

Tiene, entre otros, los métodos:

- `log(str)`
Muestra mensajes de depuración.
- `warn(str)`
Muestra mensajes de aviso.
- `error(str)`
Muestra mensajes de error.
- `assert(cond, str)`
Lanza un `AssertionError(str)` si `cond` no se cumple.

ESTRUCTURAS DE DATOS: Map Y Set

Recordemos que un **objeto** en Javascript no es más que una asociación de atributos con valores.

Esto se parece mucho al TAD Diccionario visto en EDA...

En Java: **HashMap**, **TreeMap**, etc.

Los atributos de un objeto pueden hacer el rol de las claves de un diccionario. ¿Podríamos implementar un diccionario utilizando objetos?

```
class Diccionario {
    constructor() {
        this.dict = {};
    }

    insertar(clave, valor) {
        this.dict[clave] = valor;
    }

    buscar(clave) {
        return this.dict[clave];
    }

    contieneClave(clave) {
        return this.dict[clave] !== undefined;
    }
}
```

```
let d = new Diccionario();
d.insertar(1, "Mireia");
d.insertar(2, "David");
d.buscar(2);           // → David
d.contieneClave(2);   // → 2
d.contieneClave(3);   // → 3
```

Hasta aquí todo funciona bien.

```
let d = new Diccionario();
d.insertar(1, "Mireia");
d.insertar(2, "David");
d.buscar(2);           // → David
d.contieneClave(2);   // → 2
d.contieneClave(3);   // → 3
```

Hasta aquí todo funciona bien.

Pero esta implementación tiene un fallo... y gordo.

¿Qué problema tiene?

Para evitar estos problemas, Javascript viene con una clase **Map** que implementa correctamente los diccionarios.

```
let dicc = new Map();
dicc.set(1, "Mireia");
dicc.set(2, "David");
dicc.get(2);           // → David
dicc.has(2);          // → true
dicc.has("toString"); // → false

dicc.forEach((valor, clave) => {
    console.log(` ${clave} ===> ${valor}`)
});
// Imprime:
// 1 ==> Mireia
// 2 ==> David
```

Más información: [Map.prototype](#)

También se proporciona una implementación del TAD de los conjuntos (**Set**).

```
let conj = new Set();
[25, 12, 27, 12, 90].forEach(x => conj.add(x));
conj.has(25);           // → true
conj.delete(12);
conj.size               // → 4

conj.forEach(v => { console.log(v) });
```

Convertir un conjunto en lista:

```
let lista = [..conj];
// Sumamos todos los elementos del conjunto:
lista.reduce((ac, x) => ac + x);    // → 154
```

HERRAMIENTAS

UTILIDADES Y HERRAMIENTAS

- JSDoc
<http://usejsdoc.org/>
- Depuradores
- Herramientas de *testing*

JSDOC

Herramienta de generación de documentación, al estilo de *Javadoc*

```
/**  
 * Las instancias de esta clase representan números complejos.  
 */  
class Complejo {  
    /**  
     * Construye un número complejo a partir de sus partes real  
     * e imaginaria.  
     *  
     * Puede construirse un número a partir de su forma polar  
     * mediante la función {@link Complejo.desdePolar}  
     *  
     * @param {number} real Parte real  
     * @param {number} imag Parte imaginaria  
     */  
    constructor(real, imag) { ... }  
  
    // ...  
}
```



Class: Complejo

Complejo

```
new Complejo(real, imag)
```

Representa un número complejo representado en forma rectangular (parte real + parte imaginaria). Puede construirse un número a partir de su forma polar mediante la función [Complejo.desdePolar](#)

Parameters:

Name	Type	Description
real	number	Componente real.
imag	number	Componente imaginaria.

Source:

[complex_jsdoc.js, line 15](#)

[Home](#)

[Classes](#)

[Complejo](#)

Methods

(static) [desdePolar\(mod, arg\)](#)

DEPURADORES

Existen herramientas de depuración incorporadas, tanto en el entorno del cliente como en el del servidor.

- **Lado del servidor** (Node)

La depuración se realiza mediante un *shell* lanzado desde la línea de comandos, o bien con *node-inspector*, que proporciona una interfaz gráfica:

<https://www.npmjs.com/package/node-inspector>

- **Lado del cliente** (Navegador)

Las herramientas para desarrolladores integradas en Firefox y Chrome proporcionan un depurador.

En cualquiera de los dos entornos puede introducirse un punto de ruptura mediante la siguiente sentencia:

```
debugger;
```

EJEMPLO

```
// sum_square.js
// -----
// Este programa calcula la suma de cuadrados
// del array 'arr'.

let sum = 0;
let arr = [1, 4, 8, 1, 3];

debugger; // Punto de ruptura

for (let i = 0; i < array.length; i++) {
    sum += arr[i] * arr[i];
}

console.log(sum);
```

INICIAR DEPURACIÓN CON NODE

```
node debug sum_squares.js
```

INICIAR DEPURACIÓN CON NODE

```
node debug sum_squares.js
```

COMANDOS

cont

Salta al siguiente punto de ruptura

step

Avanzar paso (metiéndose dentro de funciones o no)

next

Arrancar *shell* para evaluar expresiones

repl

Arrancar *shell* para evaluar expresiones

watch("...")

Visualizar expresión en cada paso de ejecución

DEPURACIÓN CON NODE

Depuración con Node



DEPURACIÓN CON NODE-INSPECTOR

Requiere instalación previa mediante la herramienta **npm**, distribuida junto con Node (ver Tema 4).

```
npm install -g node-inspector
```

Tras la instalación ejecutar:

```
node-debug fichero.js
```

y se abrirá un navegador con una interfaz gráfica de depuración.

Ejemplo:

Depurar con node-inspector



DEPURACIÓN CON FIREFOX

Desarrollador → Depurador (Ctrl+Mayús+S)

Sesión de depuración en Firefox



FRAMEWORKS DE TESTING

- **Mocha**
<https://mochajs.org/>
- **Jasmine**
<http://jasmine.github.io/>
- **Chai**
<http://chaijs.com/>

EJEMPLO: MOCHA

El siguiente módulo contiene un error en la función `insert`

```
/* Inserta el elemento arr[i] en la porción del array comprendida
entre los índices 0 y i-1, suponiendo que dicha porción está
ordenada */
function insert(i, arr) {
    var j = i;
    while (j > 1 && arr[j] < arr[j - 1]) {
        swap(arr, j, j - 1);
        j = j - 1;
    }
}

/* Implementación del algoritmo de ordenación por inserción */
function insertionSort(arr) {
    for (var i = 1; i < arr.length; i++) {
        insert(i, arr);
    }
}

module.exports = {
    insertionSort: insertionSort,
    insert: insert
}
```

EJEMPLO: MOCHA

El siguiente módulo contiene un error en la función `insert`

```
/* Inserta el elemento arr[i] en la porción del array comprendida
entre los índices 0 y i-1, suponiendo que dicha porción está
ordenada */
function insert(i, arr) {
    var j = i;
    while (j > 1 && arr[j] < arr[j - 1]) { ← ¡error!
        swap(arr, j, j - 1);
        j = j - 1;
    }
}

/* Implementación del algoritmo de ordenación por inserción */
function insertionSort(arr) {
    for (var i = 1; i < arr.length; i++) {
        insert(i, arr);
    }
}

module.exports = {
    insertionSort: insertionSort,
    insert: insert
}
```

Creamos una carpeta **test** y añadimos el siguiente fichero **insert_test.js**:

```
// ...
describe("Prueba de ordenación por inserción", () => {
    it("Ordenación de array ascendente", () => {
        let arr = [1, 2, 3, 4];
        testing.insertionSort(arr);
        assert.deepEqual(arr, [1, 2, 3, 4]);
    });

    it("Ordenación de array descendente", () => {
        let arr = [8, 4, 2];
        testing.insertionSort(arr);
        assert.deepEqual(arr, [2, 4, 8]);
    });

    it("Inserción en array desordenado", () => {
        let arr = [3, 2];
        testing.insert(1, arr);
        assert.deepEqual(arr, [2, 3]);
    });
});
```

Ejecutamos los casos de prueba:

```
# mocha  
...  
2 failing
```

- 1) Prueba de ordenación por inserción
Ordenación de array descendente:

```
AssertionError: [ 8, 2, 4 ] deepEqual [ 2, 4, 8 ]  
+ expected - actual
```

```
[  
- 8    ↗ Sobra esto  
     2  
     4  
+ 8    ↗ Falta esto  
]
```

Tras corregir el error:

```
function insert(i, arr) {  
    ...  
    while (j > 0 && arr[j] < arr[j - 1]) {  
        ...  
    }  
}
```

```
# mocha
```

Prueba de ordenación por inserción
✓ Ordenación de array ascendente
✓ Ordenación de array descendente
✓ Inserción en array desordenado

```
3 passing (8ms)
```

BIBLIOGRAFÍA

- A. Rauschmayer
[Speaking Javascript](#)
O'Reilly (2014)
- E. Brown
[Learning Javascript, 3rd edition](#)
O'Reilly (2016)
- [Javascript Reference](#)
MDN - Mozilla Developer Network

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>

