

TEMA 3

PROGRAMACIÓN EN JAVASCRIPT

APLICACIONES WEB - GIS - CURSO 2017/18



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

Manuel Montenegro [montenegro@fdi.ucm.es]
Dpto de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

1. **INTRODUCCIÓN**
2. **JAVASCRIPT ES COMO JAVA...**
3. **...PERO NO ES COMO JAVA**
4. **CLASES ESTÁNDAR**
5. **HERRAMIENTAS**
6. **BIBLIOGRAFÍA**

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...

PRINCIPALES 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 ECMASCRIPT

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 (8ª edición) es ECMAScript 2017 y fue publicada en junio de 2017.

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 **fuera 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.



JAVASCRIPT MÁS ALLÁ DE LA WEB

- **Aplicaciones de escritorio**, mediante Node y Electron. Ejemplos: Atom, Visual Studio Code, etc.
- **Aplicaciones móviles**, mediante Apache Cordova.
- **Extensiones de entornos de escritorio**, como GNOME Shell o Windows 10.

UN PROGRAMA DE EJEMPLO

```
// planets.js
// -----

"use strict";

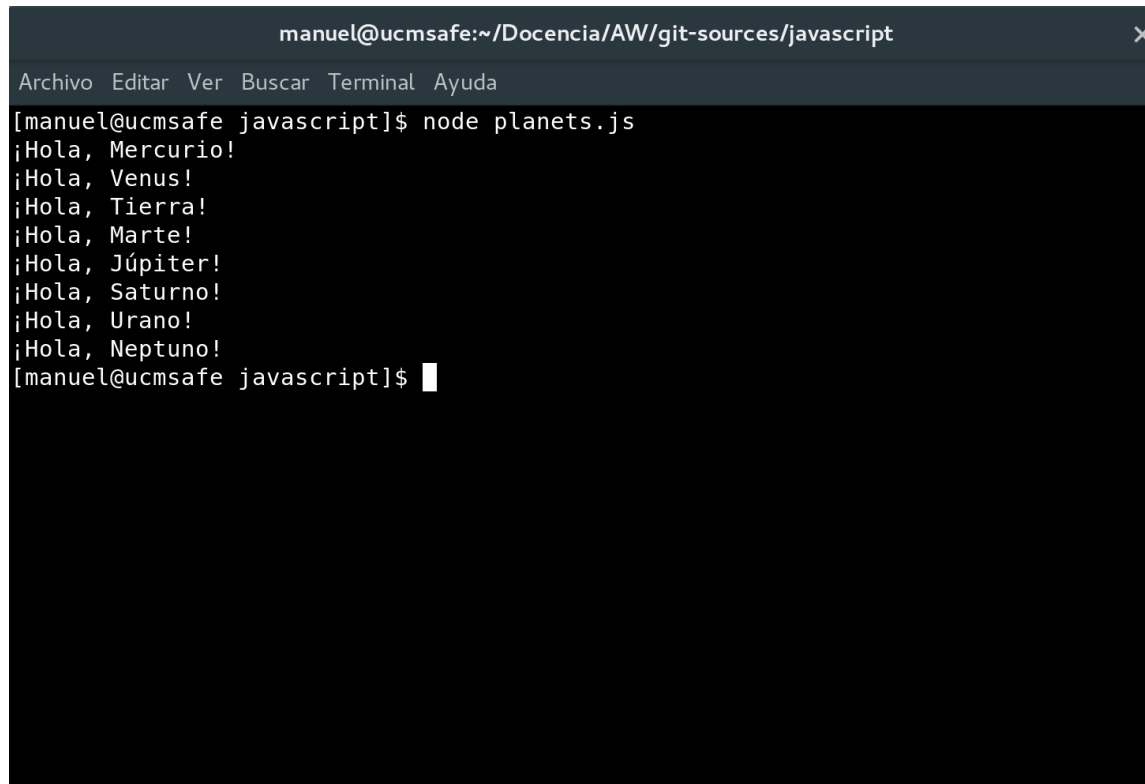
let planetas = [
    "Mercurio", "Venus", "Tierra",
    "Marte", "Júpiter", "Saturno",
    "Urano", "Neptuno"
];

planetas.forEach(p => {
    console.log(`¡Hola, ${p}!`);
});
```

EJECUCIÓN DESDE NODE

Si se tiene Node instalado, basta con ejecutar desde una línea de comandos:

`node planets.js`

A terminal window with a dark background. The title bar shows 'manuel@ucmsafe:~/Docencia/AW/git-sources/javascript' and a close button. The menu bar includes 'Archivo', 'Editar', 'Ver', 'Buscar', 'Terminal', and 'Ayuda'. The terminal content shows the command '[manuel@ucmsafe javascript]\$ node planets.js' followed by eight lines of output: '¡Hola, Mercurio!', '¡Hola, Venus!', '¡Hola, Tierra!', '¡Hola, Marte!', '¡Hola, Júpiter!', '¡Hola, Saturno!', '¡Hola, Urano!', and '¡Hola, Neptuno!'. The prompt '[manuel@ucmsafe javascript]\$' is visible at the bottom with a cursor.

```
manuel@ucmsafe:~/Docencia/AW/git-sources/javascript
Archivo Editar Ver Buscar Terminal Ayuda
[manuel@ucmsafe javascript]$ node planets.js
¡Hola, Mercurio!
¡Hola, Venus!
¡Hola, Tierra!
¡Hola, Marte!
¡Hola, Júpiter!
¡Hola, Saturno!
¡Hola, Urano!
¡Hola, Neptuno!
[manuel@ucmsafe javascript]$
```

EJECUCIÓN DESDE UN NAVEGADOR

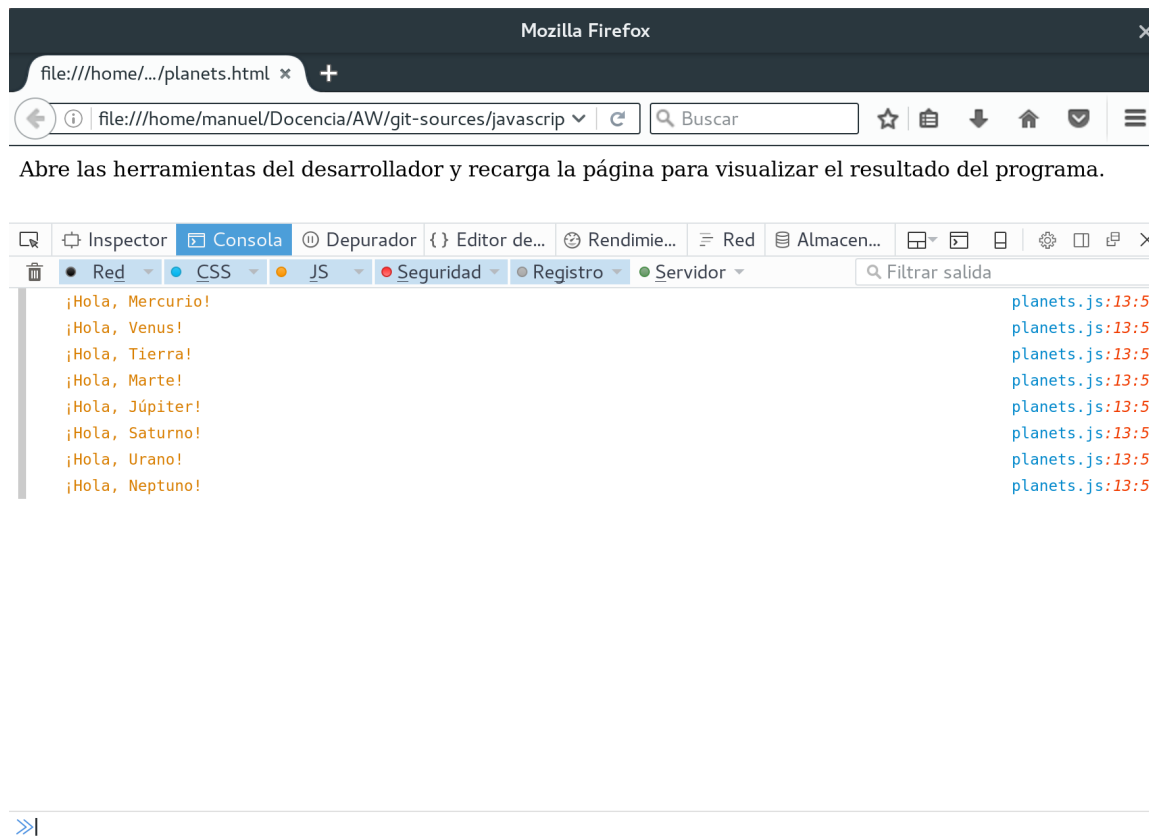
Si, alternativamente, se quiere ejecutar un programa desde el navegador, ha de importarse el fichero *Javascript* desde un documento HTML, y abrir éste último desde el navegador.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <script src="planets.js" type="text/javascript"></script>
  </head>
  <body>
    Abre las herramientas del desarrollador y recarga la página
    para visualizar el resultado del programa.
  </body>
</html>
```

Una vez cargada la página, abrir la consola de Javascript.

En Firefox: *Desarrollador* → *Consola Web* (Ctrl + Mayús + K)

En Chrome: *Herramientas del desarrollador* → *Consola*



¿"use strict"?

Javascript es un lenguaje muy flexible.

...pero esto conlleva una gran responsabilidad.

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

Esto hace que programar en Javascript fuese una tarea propensa a errores.

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

Este modo hace que el compilador detecte y prohíba el uso de características demasiado permisivas del lenguaje:

- Utilizar variables sin declararlas.
- Borrar una variable mediante `delete`.
- Duplicidades en nombres de atributos y parámetros.
- Estructuras poco recomendables: `with`.

1. INTRODUCCIÓN
2. JAVASCRIPT ES COMO JAVA...
3. ...PERO NO ES COMO JAVA
4. CLASES ESTÁNDAR
5. HERRAMIENTAS
6. BIBLIOGRAFÍA

EL «JAVA» EN JAVASCRIPT

La sintaxis de Javascript está inspirada en la de Java.

- Comentarios:

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

- Declaraciones de variables y asignaciones:

```
let x = 3;  
let y = x + 1;  
y += 3;
```

- Pre/postincremento,
pre/postdecremento:

```
z = x++;  
--x;
```


DECLARACIÓN DE VARIABLES

Hay distintas variantes:

- Declaraciones **const** para constantes.

```
const x = 1;
```

Su valor no puede cambiar a lo largo del programa.

- Declaraciones **let** y **var** para variables.

```
let x = 1;  
var x = 1;
```

Se recomienda el uso de **let**, porque tienen una noción de ámbito más «familiar» para programadores/as que provienen de otros lenguajes.

SENTENCIAS CONDICIONALES

- If-then-else:

```
if (x < y) {  
    return x;  
} else {  
    return y;  
}
```

- Switch:

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

BUCLES

- Bucle while:

```
while (x <= 0) {  
    n = n * x;  
    x--;  
}
```

- Bucle do-while:

```
do {  
    mult *= a[x];  
    x++;  
} while (a[x] == 0)
```

- Bucles for:

```
for (let i = 0; i < b.length; i++) {  
    sum += b[i] + c[i];  
}
```

- Bucles for...of para recorrer arrays:

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

Evita utilizar los bucles for...in

- break y continue:

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

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

FUNCIONES

- Definición de funciones:

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

- Llamadas a funciones y métodos:

```
x = abs(-3);  
console.log(`El valor absoluto de x es ${x}`);
```

MANEJO DE EXCEPCIONES

- Bloques try-catch o try-catch-finally:

```
try {  
    funcion_no_existe();  
} catch (e) {  
    console.error(e.message);  
} finally {  
    console.log("Esto se ejecuta siempre");  
}
```

No se especifica el tipo de excepción

- Lanzamiento de excepciones:

```
throw new Error("Fichero no encontrado");
```

- 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:

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

o bien

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

- Acceso al carácter i-ésimo:

```
str[2]  // → "t"
```

- Métodos:

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


CADENAS PLANTILLA

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

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

Corchetes en lugar de {}

```
let z = [];
```

Vector 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
```

1. INTRODUCCIÓN
2. JAVASCRIPT ES COMO JAVA...
3. ...**PERO NO ES COMO JAVA**
4. CLASES ESTÁNDAR
5. HERRAMIENTAS
6. BIBLIOGRAFÍA

ONCE RAREZAS DE JAVASCRIPT

(para alguien que viene de Java)

1. Javascript es dinámicamente tipado.
2. Los objetos no son necesariamente instancias de clases.
3. Valores indefinidos, nulos y NaN.
4. Las conversiones invisibles.
5. Argumentos que sobran y faltan.
6. Las funciones son ciudadanos de primera clase.
7. Las clases llegaron veinte años tarde.
8. La herencia basada en prototipos.
9. Los arrays son flexibles, y también son objetos.
10. Las funciones de orden superior sobre arrays.
11. Los módulos aún están por aterrizar.

RAREZA 1

JAVASCRIPT ES UN LENGUAJE
DINÁMICAMENTE TIPADO

DECLARACIONES DE TIPO

En Java es necesario declarar el tipo de una variable antes de su primer uso:

```
int x;  
String z = "It's something";  
List<Integer> lista;
```

En Javascript no se indica el tipo de la variable a declarar. Se utilizan las palabras **let**, **const** o **var**.

```
let x;  
const z = "It's something";  
let lista;
```

¿Qué ventajas e inconvenientes tiene indicar el tipo de las variables en el programa?

LENGUAJES ESTÁTICAMENTE TIPADOS

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

Por ejemplo, el compilador avisa de errores como:

```
"pepe" * 24
```

- Estos lenguajes pueden requerir declaraciones de tipos por parte del programador.

Por ejemplo: Java, C, etc.

- En algunos lenguajes el compilador **infiere** los tipos.

Por ejemplo: Haskell, C++, etc.

LENGUAJES DINÁMICAMENTE TIPADOS

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

Por ejemplo, en Javascript:

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

El error debido a la expresión `3 * "foo"` solamente se manifestará si la condición del `if` se cumple.

EJEMPLOS

TIPOS DISPONIBLES EN JAVASCRIPT

- Tipo numérico (**number**).

El sistema de tipos no distingue entre enteros y coma flotante.

- Tipo booleano (**boolean**).

Incluye los valores **true** y **false**.

- Tipo cadena (**string**).
- Tipo objeto (**object**).

Incluye también a arrays, funciones y expresiones regulares.

- Tipo del valor indefinido (**undefined**)
- Tipo del puntero nulo (**null**)

TIPOS PRIMITIVOS VS TIPOS OBJETO

- Tipos primitivos
 - Numérico
 - Booleano
 - Cadena
 - Indefinido
 - Nulo
- Tipos objeto
 - Objeto

Veamos cuál es la diferencia.

Los tipos primitivos son **inmutables**

```
let str1 = "Cadena";  
let str2 = str1.slice(0, 3);  
console.log(str1); // → Cadena1  
console.log(str2); // → Cad
```

No modifica la cadena **str1**

Las comparaciones entre tipos primitivos se hacen por **valor**, no por referencia

```
let str1 = "Cadena";  
let str2 = "Cadena";  
let str3 = "Otra cadena";  
str1 == str2; // → true  
str1 === str2; // → true  
str1 != str3 // → true  
str1 !== str3 // → true
```

En Java **str1 == str2** devuelve **false**, pues son objetos distintos. En cambio, **str1.equals(str2)** devuelve **true**.

Las cadenas pueden compararse lexicográficamente mediante los operadores relacionales `<=`, `<`, `>=` y `>`:

```
"Pablo" < "Diana"      // → true
"Pablo" < "Paolo"      // → true
"alma" <= "Pablo"      // → false
"Águeda" < "Pablo"    // → false
```

Se utiliza `localeCompare()` para comparar según el abecedario del idioma correspondiente.

```
"alma".localeCompare("Pablo");    // → -1 ('alma' es menor)
"Sergio".localeCompare("Pablo");  // → 1  ('Pablo' es menor)
"Sergio".localeCompare("Sergio"); // → 0  (iguales)
```

COMPROBACIÓN DE TIPOS

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

```
let x = 3;
console.log(typeof(x));           // → number
console.log(typeof("Hola"));      // → string
console.log(typeof(variable_que_no_existe)); // → undefined
console.log(typeof(2 < 9));        // → boolean
console.log(typeof([1, 3, 5]));    // → object
console.log(typeof(null));         // → object
```

Aunque `null` es un tipo básico, `typeof(null)` devuelve `"object"`.

RAREZA 2

**LOS OBJETOS NO TIENEN POR QUÉ SER
INSTANCIAS DE CLASES**

OBJETOS EN JAVA

En Java, un objeto es una **instancia de una clase**.

```
class Persona {  
    public String nombre;  
    public String apellidos;  
    public int edad;  
}  
...  
p = new Persona();
```

La clase es la que define los atributos y métodos del objeto.

```
p.nombre = "Ana María";  
System.out.println(p.edad);  
p.poblacion = "Barcelona";
```

¡Error! Atributo no declarado en clase

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  
};
```

Aquí se definen los atributos **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  
y.nombre // → 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 (igualdad al estilo de Java)

```
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
```

Recuerda: en Javascript, las cadenas no son objetos; son tipos básicos.

RAREZA 3

VALORES INDEFINIDOS, NULOS, Y NaN

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
```

EL VALOR NOT-A-NUMBER (NaN)

Se devuelve como resultado de operaciones aritméticas incorrectas:

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

LOS VALORES INFINITOS

Infinity y **-Infinity** se utilizan para desbordamientos, o para operaciones que devuelven $\pm\infty$

```
Math.pow(2, 10000) // → Infinity  
Math.log(0)        // → -Infinity
```

¡Cuidado con las comparaciones de 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
```

RAREZA 4

LAS CONVERSIONES INVISIBLES

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
```

¿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, en caso de no estar seguro/a 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`, se devuelve `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("")           // → true  
Boolean(34)           // → false
```

OPERADORES DE IGUALDAD

x === y - Igualdad estricta

x e **y** son del mismo tipo y tienen el mismo valor.

x == y - Igualdad flexible

x e **y** pueden convertirse al mismo tipo, de modo que tras hacer la conversión tienen el mismo valor.

```
"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)
```

También se definen **!=** y **!==** como la negación de **==** y **===** respectivamente.

MORALEJA

Utiliza siempre `===` y `!==`

RAREZA 5

ARGUMENTOS QUE SOBRAN Y
ARGUMENTOS QUE FALTAN

DEFINICIÓN DE UNA FUNCIÓN

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

LLAMADA A UNA FUNCIÓN

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

Resultado:

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

El número de argumentos en la llamada a la función no ha de coincidir necesariamente con el número de parámetros en la definición

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

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

```
p1: uno  
p2: dos  
p3: tres
```

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

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

```
p1: uno  
p2: dos  
p3: undefined
```

Esto nos permite definir funciones con parámetros opcionales.

```
/*  
    El parámetro 'color' es opcional. Su valor por defecto  
    es 'negro'.  
  
    El parámetro 'trazo' especifica el grosor del trazo y  
    también es opcional. Su valor por defecto es 1.  
*/  
function pintar_circulo(x, y, color, trazo) {  
    if (color === undefined) color = "negro";  
    if (trazo === undefined) trazo = 1;  
  
    console.log(`Pintar círculo en (${x}, ${y}) de color ${color}` +  
                ` y trazo de grosor ${trazo}`;  
}
```


Aunque hay una sintaxis específica para ello:

```
/*  
    El parámetro 'color' es opcional. Su valor por defecto  
    es 'negro'.  
  
    El parámetro 'trazo' especifica el grosor del trazo y  
    también es opcional. Su valor por defecto es 1.  
*/  
function pintar_circulo(x, y, color = "negro", trazo = 1) {  
    console.log(`Pintar círculo en (${x}, ${y}) de color ${color}` +  
        ` y trazo de grosor ${trazo}`;  
}
```

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  
    opcionales.  
*/  
function abrir_fichero(nombre, ops = {}) {  
    // Inicialización de los parámetros opcionales 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:

```
/*  
    El objeto 'ops' tiene como atributos los parámetros  
    opcionales.  
*/  
function abrir_fichero(nombre,  
    {solo_lectura = true, binario = false} = {}) {  
    // Cuerpo de la función  
    console.log(`Abriendo fichero ${ops.binario ? "binario " : ""}` +  
        `${nombre} en modo ` +  
        (ops.solo_lectura ? "lectura" : "lectura/escritura"))`;  
}
```

Más información: [Destructuring assignment](#)

RAREZA 6

**LAS FUNCIONES SON CIUDADANOS DE
PRIMERA CLASE**

¿QUÉ SIGNIFICA ESO?

Que las funciones son tratadas como cualquier otro valor.

En particular:

- Se puede asignar una función a una variable.
- Pueden pasarse funciones como parámetros.
- Pueden recibirse funciones como resultados.

Esta característica es compartida por muchos lenguajes funcionales e imperativos:

Haskell, Scala, Erlang, C, C++, Java 8, etc.

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 la variable **f**:

```
let f = incrementar;  
console.log(f(5));  
// Imprime: 6  
  
f = factorial;  
console.log(f(10));  
// Imprime: 3628800
```

¡Cuidado con los paréntesis!

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

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

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

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 utilizar una definición de función en cualquier sitio donde se espere 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));
```

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?

NOTACIÓN LAMBDA

Existe una sintaxis más sencilla para denotar funciones anónimas.

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);
```

o incluso:

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

CLAUSURAS

Una función puede hacer referencia a variables declaradas en un ámbito superior

```
let y = 3; // variable global  
  
let f = (x => x + y);  
  
console.log(f(5));  
// Imprime: 8
```

Referencia a la variable global y

¿Y si cambio el valor de la variable **y** después de definir **f**?

```
y = 9;  
console.log(f(2));  
// Imprime: 11
```

FUNCIONES DENTRO DE OBJETOS

Como las funciones son ciudadanos de primera clase, pueden ser asignadas a los atributos de un objeto:

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

Función anónima sin parámetros

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

Se puede añadir métodos a un objeto ya construido:

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

EL OBJETO `this`

```
empleado.saludar();
```

En toda llamada a método se distinguen tres componentes:

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

Cuando llamamos 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: () => {  
    console.log(`¡Hola, ${this.nombre}!`);  
  }  
  
  cambiarNombre: 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.

¿Qué ocurre al ejecutar el siguiente código?

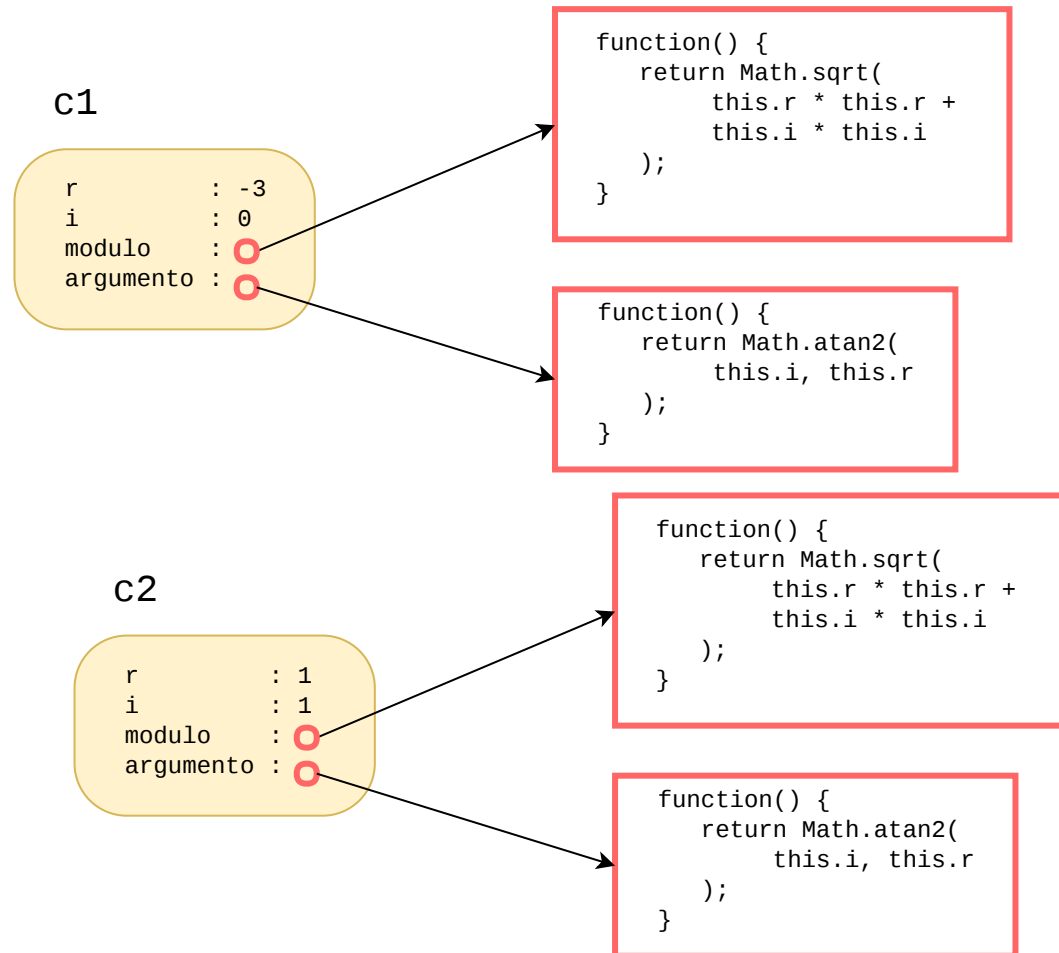
```
var f = empleado.saludar;  
f();
```

Podemos encapsular la creación de objetos mediante funciones constructoras:

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

(Más adelante veremos otra forma de crear funciones constructoras)

Problema: duplicidad de objetos función para cada objeto.





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

Posible 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  
    );  
}
```

Ahora añadimos 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**?

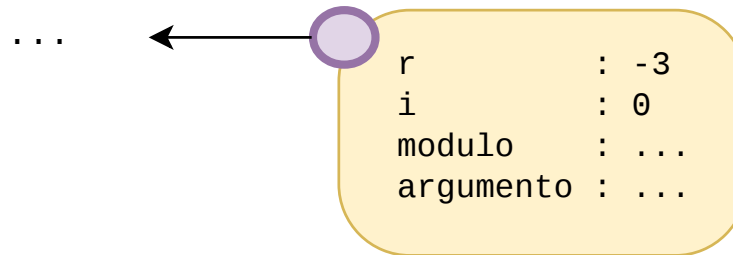
Sí. Se puede hacer mediante **prototipos**.

RAREZA 7

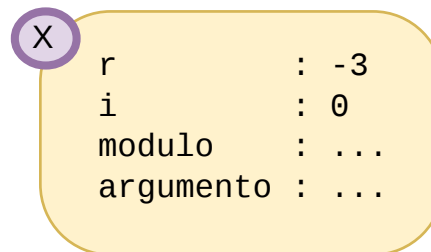
LA HERENCIA BASADA EN PROTOTIPOS

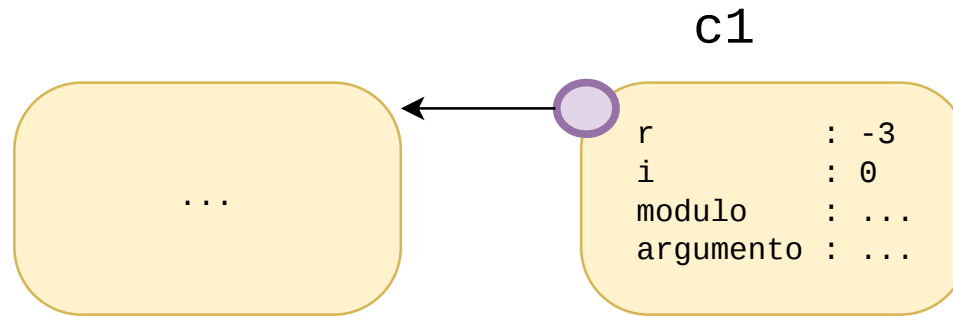
PROTOTIPO DE UN OBJETO

Todo objeto en Javascript tiene un puntero «secreto» que puede apuntar a otro objeto:



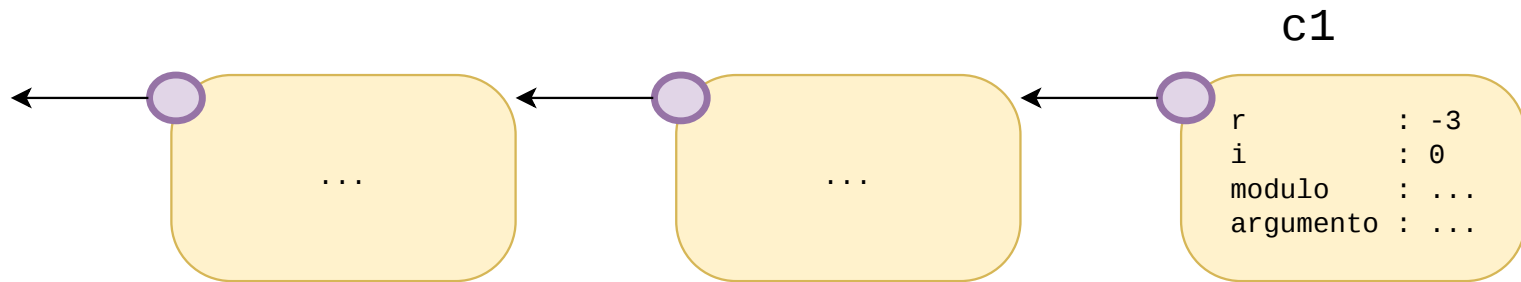
... o puede ser un puntero nulo:





En el primer caso decimos que el objeto apuntado es **prototipo** de **c1**.

A su vez, el prototipo de **c1** puede tener otro prototipo:

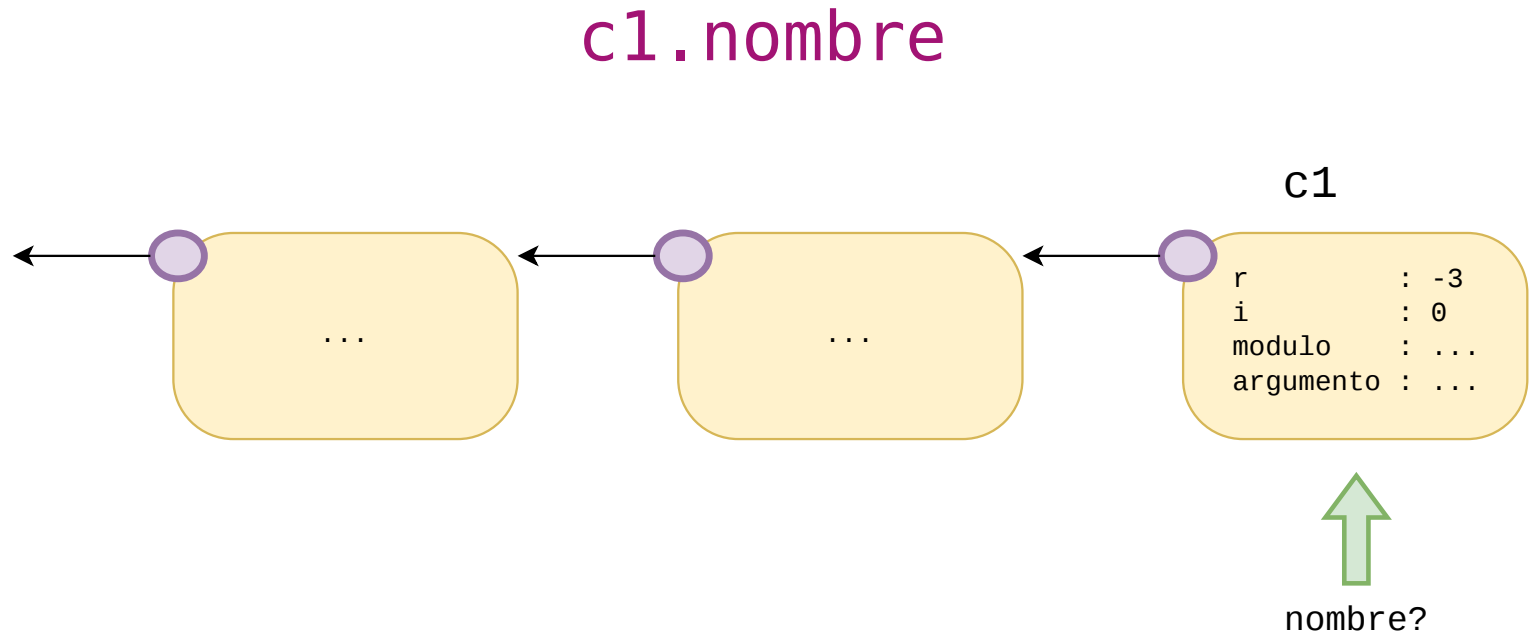


De este modo tenemos una **cadena de prototipos**, que acabará en un objeto no tenga prototipo.

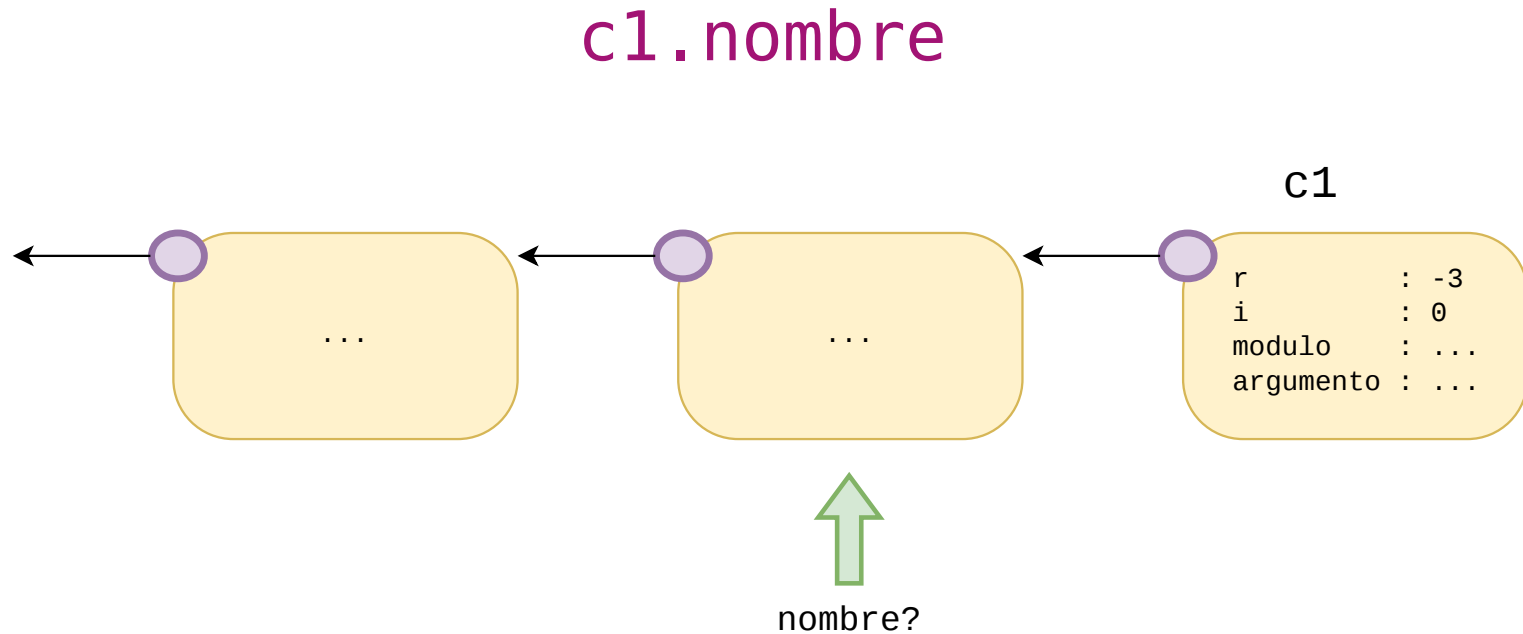
BÚSQUEDA DE ATRIBUTOS

¿Qué ocurre cuando se accede al atributo de un objeto?

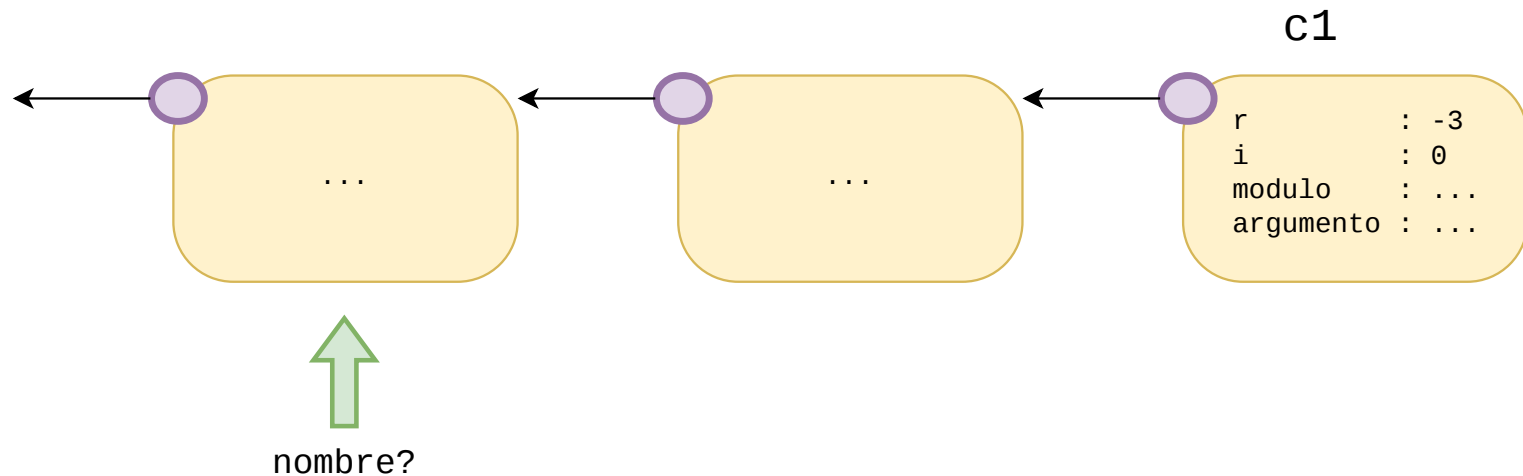
Supongamos la expresión **c1.nombre**



En primer lugar se intenta buscar un atributo llamado `nombre` dentro de `c1`.



Si no se encuentra en **c1**, se busca el atributo dentro del prototipo de **c1**.



Si no se encuentra en el prototipo de **c1**, se busca dentro del prototipo del prototipo de **c1**. Si no se encuentra allí, la búsqueda continúa 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 este caso la expresión **c1.nombre** se evalúa a **undefined**.

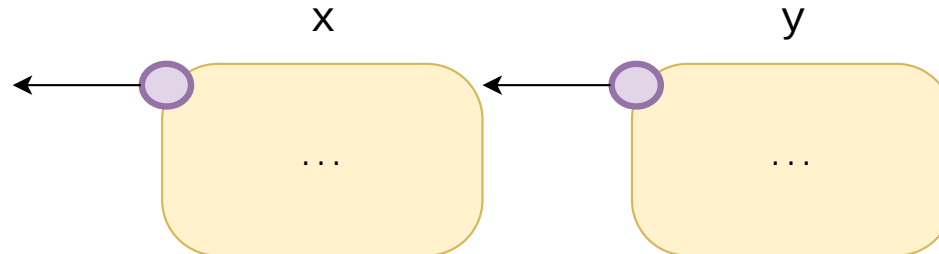
CREAR OBJETOS QUE APUNTEN A UN PROTOTIPO

Se utiliza la función `Object.create()`

La siguiente sentencia:

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

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



Ejemplo:

```
let circulo = {  
  centro: { x: 10, y: 20 },  
  radio: 5  
};
```

```
let circulo_verde = Object.create(circulo);  
circulo_verde.color = "verde";
```

```
let circulo_rojo = Object.create(circulo);  
circulo_rojo.color = "rojo";
```

```
console.log(circulo_rojo.color);      // → "rojo"  
console.log(circulo_verde.radio);     // → 5  
console.log(circulo_verde.centro.x);  // → 10  
console.log(circulo_rojo.centro);     // → { x: 10, y: 20 }
```

CAMBIAR Y AÑADIR ATRIBUTOS AL PROTOTIPO

Añadimos lo siguiente al ejemplo anterior:

```
circulo.grosorBorde = 2;
```

¿Qué ocurre con `circulo_verde` y `circulo_rojo`?

```
console.log(circulo_rojo.grosorBorde);    // → 2  
console.log(circulo_verde.grosorBorde);    // → 2
```

«Heredan» automáticamente el nuevo atributo.

Lo mismo ocurre con las modificaciones al prototipo:

```
circulo.radio = 6;  
  
console.log(circulo_rojo.radio);    // → 6  
console.log(circulo_verde.radio);    // → 6
```

SOBREESCRITURA DE ATRIBUTOS

Un objeto puede sobrescribir cualquier atributo heredado de su prototipo:

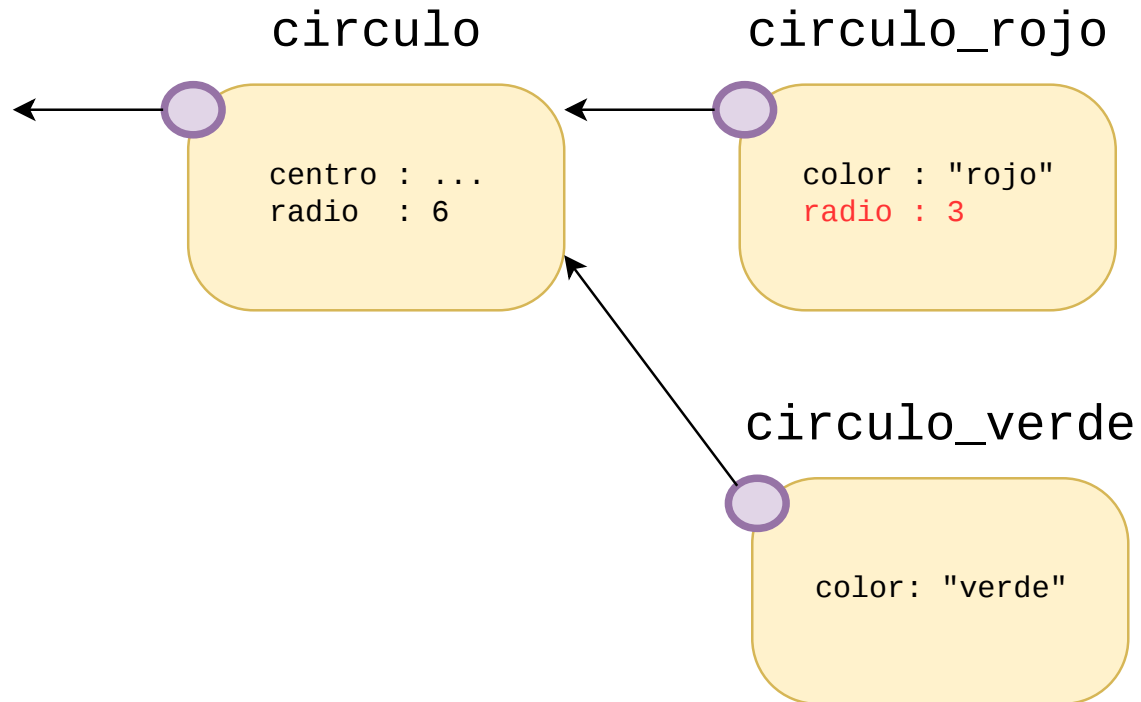
```
circulo_rojo.radio = 3;
```

```
console.log(circulo_rojo.radio);    // → 3
```

```
console.log(circulo_verde.radio);   // → 6
```

El atributo sobrescrito no afecta a los demás objetos que tengan el mismo prototipo.

`circulo_rojo.radio = 3;`





¿Qué ocurre con lo siguiente?

```
circulo_rojo.centro.x = 15;  
console.log(circulo_verde.centro.x);
```


Volviendo al ejemplo de los números complejos



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  
    );  
}
```

```
var prototipoComplejo = {  
  modulo: () => {  
    return Math.sqrt(this.r * this.r + this.i * this.i);  
  },  
  argumento: () => {  
    return Math.atan2(this.i, this.r);  
  }  
};
```

```
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);
```

Todas las funciones 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.coordenadasPolares = () => {
  console.log(`(${this.modulo()}, ${this.argumento()})`);
}

c1.coordenadasPolares();
// → (3, 3.141592653589793)

c2.coordenadasPolares();
// → (1.4142135623730951, 0.7853981633974483)
```

EL OBJETO `Object.prototype`

Por defecto, un objeto tiene como prototipo `Object.prototype`.

`Object.prototype` tiene algunos métodos predefinidos:

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

EJEMPLOS

```
let c3 = construirComplejo(1, 3);

console.log(c3.toString());
// → [object Object]

prototipoComplejo.toString = function() {
  return "(" + this.r + ", " + this.i + ")";
}

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

console.log(prototipoComplejo.isPrototypeOf(c3));
// → true
```

RAREZA 8

**LAS CLASES EN JAVASCRIPT LLEGARON
VEINTE AÑOS TARDE**

Recordemos nuestro ejemplo sobre números complejos

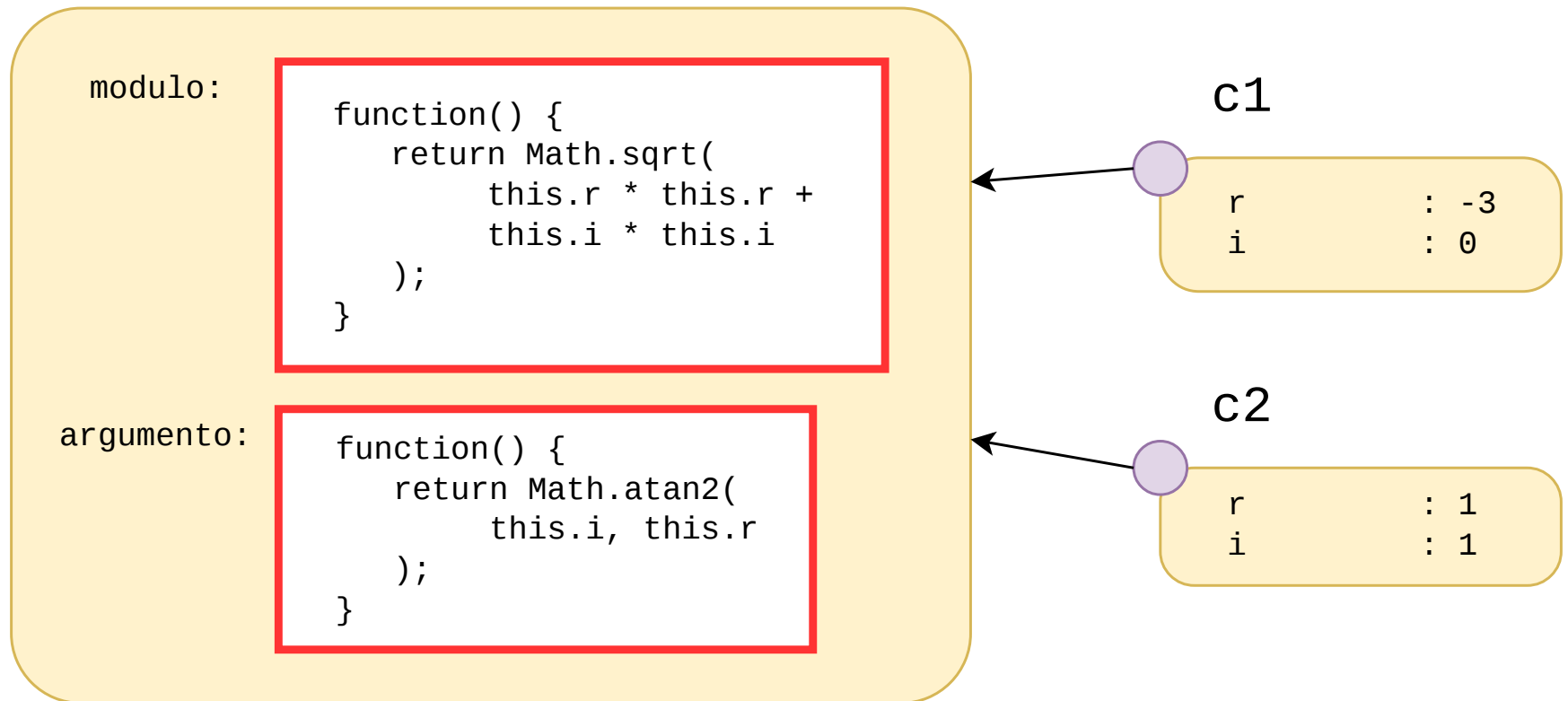
```
var prototipoComplejo = {  
  modulo: () => {  
    return Math.sqrt(this.r * this.r + this.i * this.i);  
  },  
  argumento: () => {  
    return Math.atan2(this.i, this.r);  
  }  
};
```

```
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 situación queda representada así:

prototipoComplejo



Este patrón es bastante común en Javascript para simular las **clases** de otros lenguajes de programación:

- 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.

Era un patrón tan común que Javascript proporcionaba mecanismos para facilitar su uso.

- ECMAScript ≤ 5 : funciones constructoras + operador **new**.
- ECMAScript ≥ 6 : clases + operador **new**.

CLASES

Las clases de ECMAScript 6 son similares a las de Java:

```
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);  
  }  
}
```

Constructora de clase

Método

Método

Esta declaración crea un objeto llamado **Complejo.prototype** que almacena estos dos métodos.

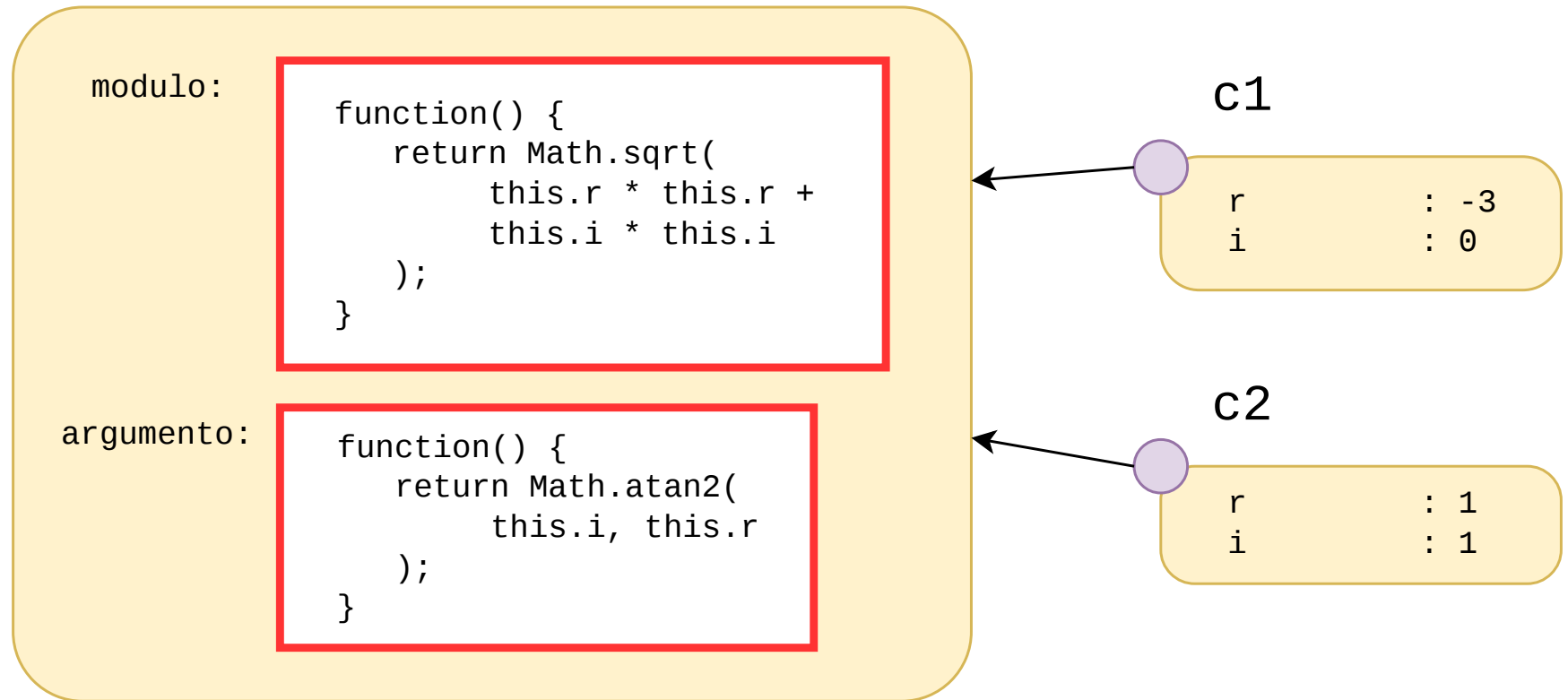
Al igual que en Java, se crean instancias mediante el operador **new**.

```
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





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
```

FUNCIONES get/set

Son funciones llamadas cada vez que se lee o escribe a un determinado atributo.

```
class Complejo {  
  // ...  
  get modulo() {    
    return Math.sqrt(this.r * this.r + this.i * this.i);  
  }  
  
  set modulo(newModulo) {    
    console.log("Llamando a set modulo");  
    if (newModulo >= 0 && this.modulo > 0) {  
      let scaleFactor = newModulo / this.modulo;  
      this.r *= scaleFactor;  
      this.i *= scaleFactor;  
    }  
  }  
}
```


Cada vez que se accede al atributo **modulo**, en realidad se llama a la función **get**:

```
let c1 = new Complejo(1, 1);  
console.log(c1.modulo);      // Imprime: 1.4142135623730951
```

Cada vez que se modifica **modulo**, en realidad se llama a la función **set**, pasando por parámetro el valor asignado.

```
c1.modulo = 3;  
console.log(c1.r);           // Imprime: 2.1213203435596424
```

Esto es útil para encapsular el acceso a atributos:

```
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
```


MÉTODOS ESTÁTICOS

Son métodos de clase.

Van precedidos por la palabra **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); // Imprime 0.7071067811865476
```

Es posible añadir métodos estáticos una vez declarada la clase:

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

```
Complejo.CERO = new Complejo(0, 0);
```

Atributos estáticos

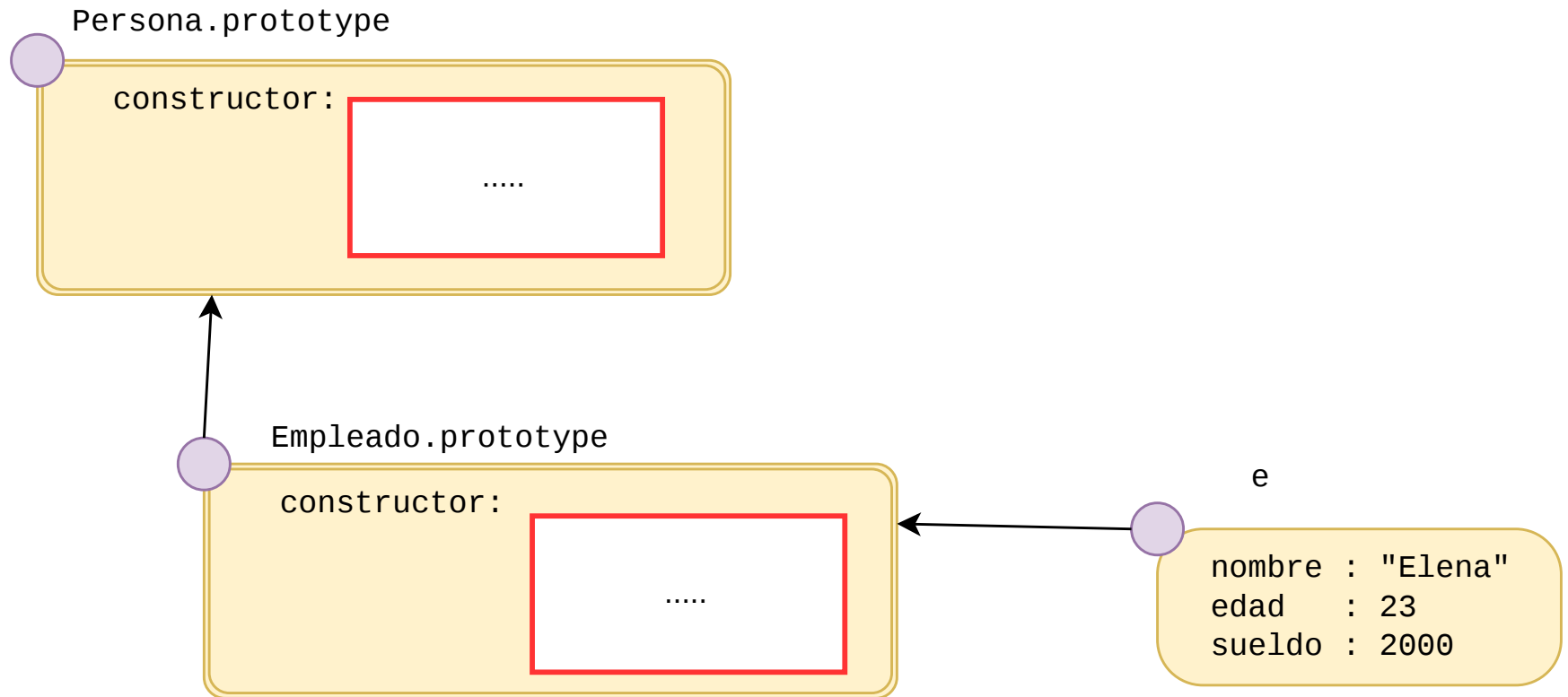
HERENCIA

Se permite herencia simple, al igual que en Java.

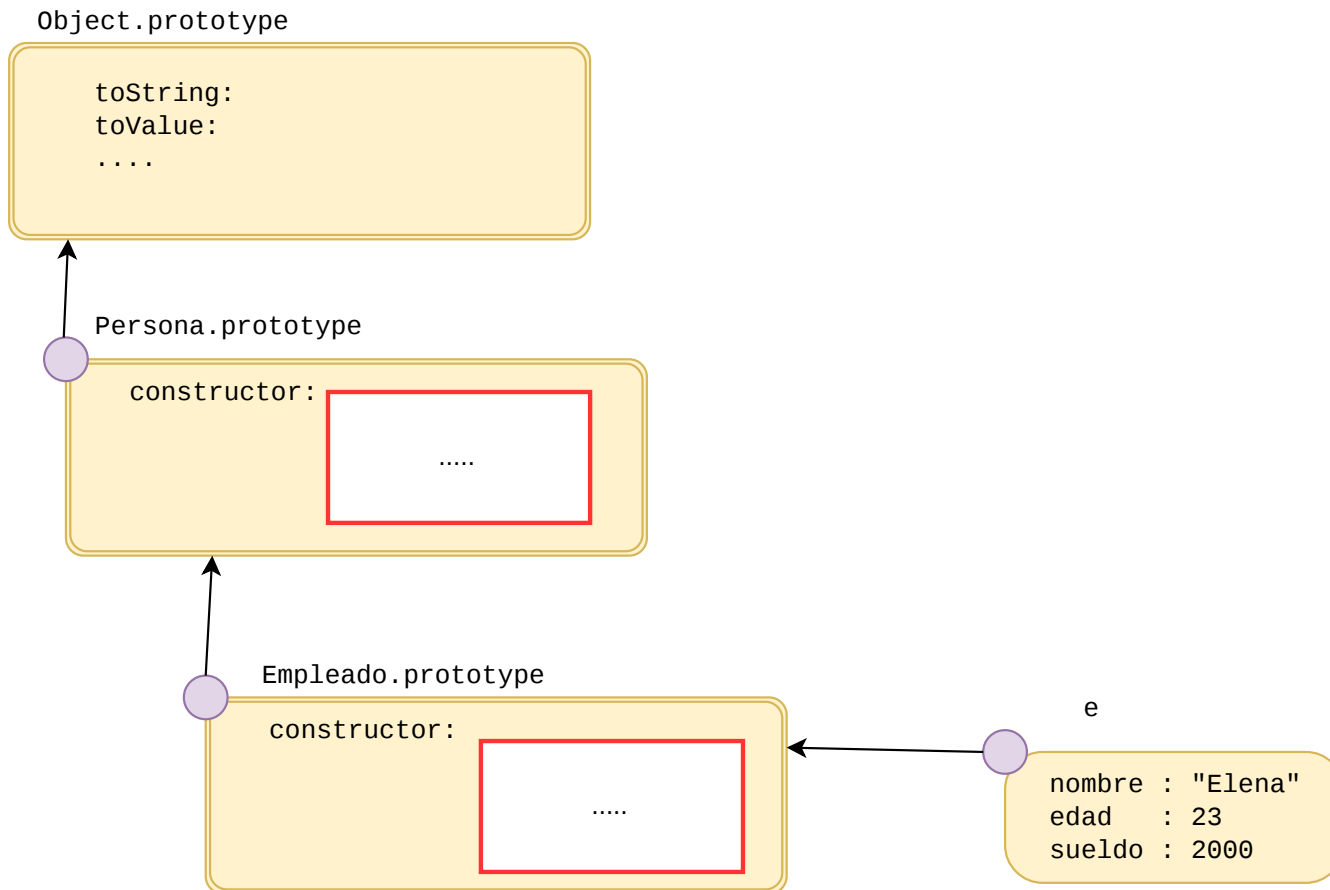
```
class Persona {  
    constructor(nombre, edad) { ... }  
    ...  
}  
  
class Empleado extends Persona {  
    constructor(nombre, edad, sueldo) {  
        super(nombre, edad);  
        this.sueldo = sueldo;  
    }  
    ...  
}
```

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



Podemos sobrescribir métodos heredados:

```
class Complejo {  
    // ...  
  
    toString() {  
        return `${this.r}, ${this.i}`;  
    }  
}
```

El método `toString` está heredado de `Object`.

```
let c1 = new Complejo(-1, 0);  
console.log(`El número es ${c1}`); // Imprime: El número es (-1, 0)
```

RAREZA 9

**LOS ARRAYS SON FLEXIBLES, Y TAMBIÉN
SON OBJETOS**

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);  
// → [ 1, 5, 3, 5, 4, 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 última aparición de `elem` en el array, 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.

RAREZA 10

LAS FUNCIONES DE ORDEN SUPERIOR SOBRE ARRAYS

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.

ITERACIÓN: MÉTODO `forEach`

- `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");  
})
```

FUNCIONES DE TRANSFORMACIÓN

- `map(f)`

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

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

- `filter(f)`

Selecciona los elementos `x` del array tales que `f(x)` devuelve `true`.

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

FUNCIONES DE REDUCCIÓN (I)

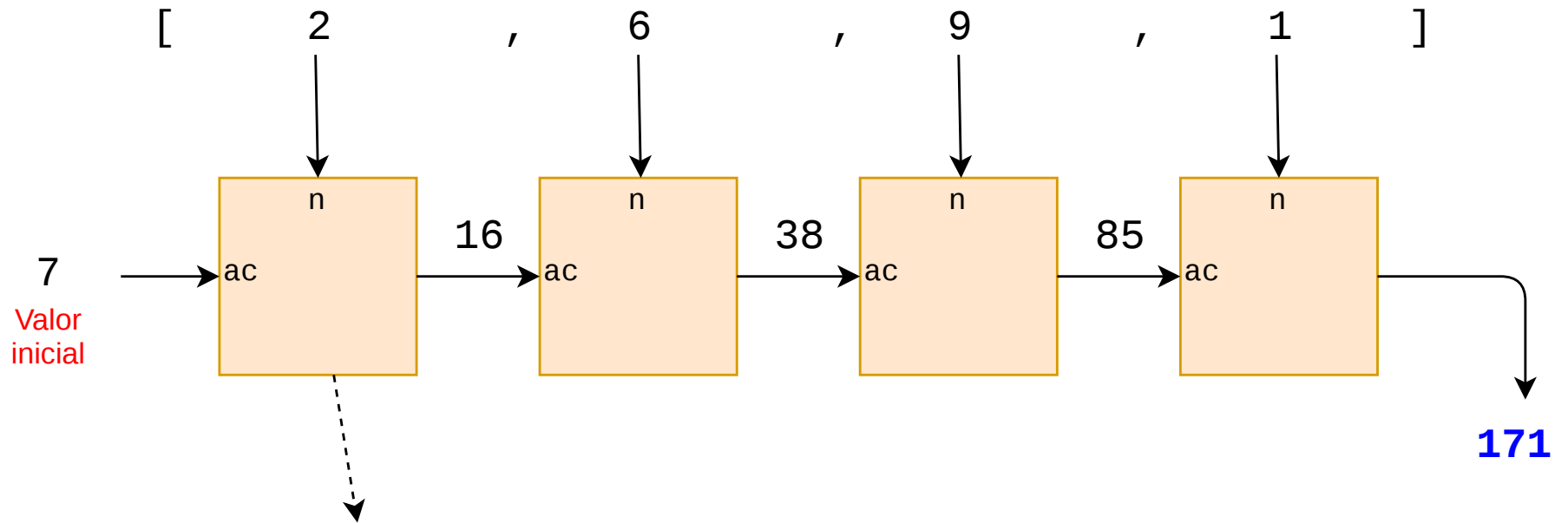
- `every(f)`
Devuelve `true` si para *todo* elemento `x` del array, `f(x)` devuelve `true`.
- `some(f)`
Devuelve `true` si *existe* un elemento `x` en el array tal que `f(x)` devuelva `true`.

FUNCIONES DE REDUCCIÓN (II)

- `reduce(f, [elemInicial])`

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((acum, n) => 2 * acum + n, 7)  
); // → 171
```



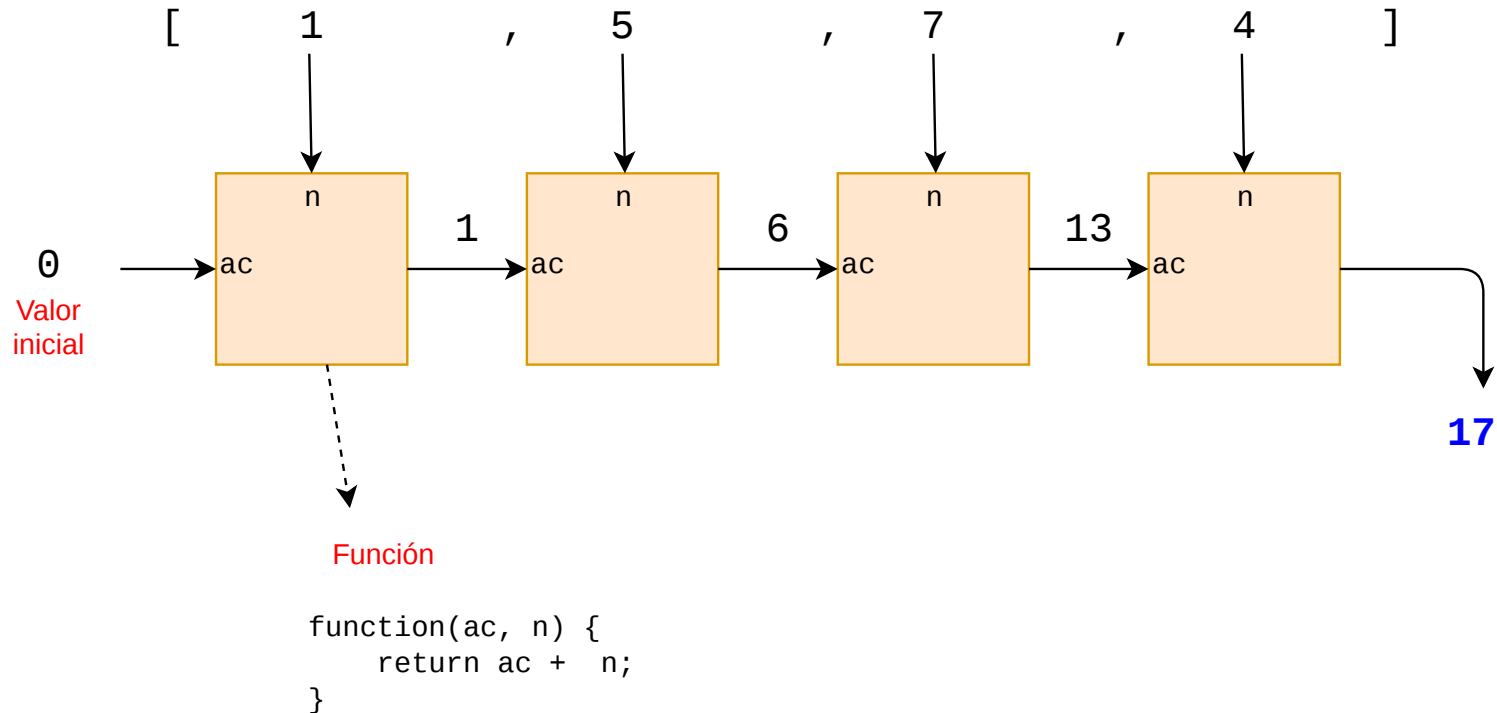
Función

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

OTROS EJEMPLOS

Suma de los elementos de un array

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



OTROS EJEMPLOS

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((acum, x) => Math.max(acum, x), -Infinity);
```

```
// o bien
```

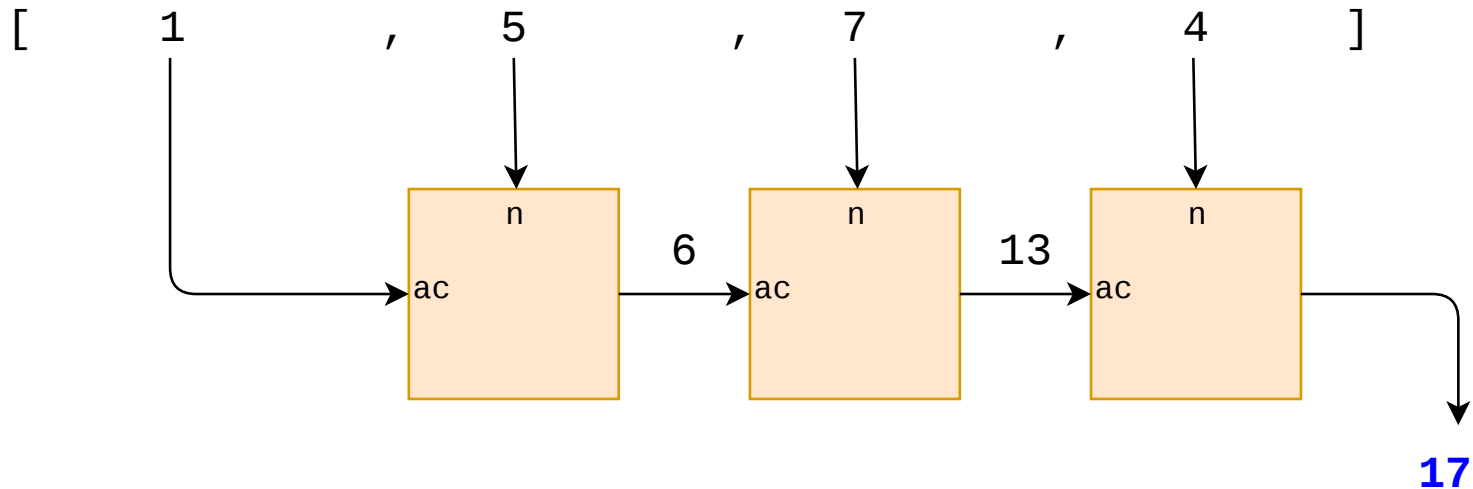
```
[6, 1, 4, 3, 7].reduce(Math.max, -Infinity);
```

VARIANTES

`reduce(f, [ini])`

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

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



LA FUNCIÓN REDUCTORA

`reduce(f, [ini])`

La función `f` puede recibir hasta cuatro parámetros:

1. Valor acumulado hasta el momento.
2. Elemento actual del array.
3. Índice actual del array.
4. Array sobre el que se está haciendo el `reduce`.

VARIANTES

`reduceRight(f, [ini])`

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

RAREZA 11

LOS MÓDULOS AÚN ESTÁN POR ATERRIZAR

Hasta la versión 6 de ECMAScript, el estándar no había especificado ningún sistema para estructurar los programas Javascript complejos en módulos.

En versiones anteriores podían simularse módulos utilizando otros mecanismos disponibles en el lenguaje.

En temas posteriores veremos dos de estos mecanismos:

- **CommonJS**, utilizado en *Node.js*.
- **AMD** (*Asynchronous Module Definition*), utilizado en los navegadores.

Pese a que el sistema de módulos está definido en ECMAScript 6 (2015), los navegadores solo lo soportan de manera experimental.

Por este motivo no lo utilizaremos en este curso.

Y LA LISTA DE RAREZAS NO ACABA AQUÍ

- Número no prefijado de argumentos (`...args`).
- Operador de propagación (`...`).
- La clase `Symbol`.
- Destructuración mediante patrones.
- Iteradores.
- Funciones generadoras (`function*`).

Más información: <http://exploringjs.com/es6.html>

1. INTRODUCCIÓN
2. JAVASCRIPT ES COMO JAVA...
3. ...PERO NO ES COMO JAVA
4. CLASES ESTÁNDAR
5. HERRAMIENTAS
6. BIBLIOGRAFÍA

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

EXPRESIONES REGULARES

Una expresión regular es un patrón que representa una o varias cadenas 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](#)

En Javascript se delimitan las expresiones regulares por
símbolos /

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

Las expresiones regulares son objetos con los siguientes
métodos:

- **test(str)**: Devuelve **true** si en la cadena **str** existe una subcadena que encaja con la expresión, o **false** en caso contrario.
- **exec(str)**: También comprueba el ajuste de alguna subcadena de **str** con el patrón, pero devuelve más información sobre el ajuste producido (o **null** si no hay ajuste).

MÉTODO test

```
/[A-Z][0-9]{3}/.test("A655");  
    // → true  
/[A-Z][0-9]{3}/.test("Otra cosa");  
    // → false  
  
/el|la|los|las/.test("Esta frase tiene un artículo");  
    // → false  
/el|la|los|las/.test("Esta clase tiene un artículo");  
    // → true  
/\b(el|la|los|las)\b/.test("Esta clase tiene un artículo");  
    // → false
```

El símbolo **\b** representa el límite de una palabra.

MÉTODO `exec`

A veces podemos 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 `test` 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.
Varía el comportamiento de ^ y \$. |

EJEMPLOS

```
var r1 = /Hola/i;
r1.test("hola") // → true

var str = "Hola, hola\nHola caracola";

str.match(/hola/);
    // → [ 'hola', index: 6, ... ]
str.match(/hola/i);
    // → [ 'Hola', index: 0, ... ]

// La búsqueda global encuentra todos los resultados
str.match(/hola/gi);
    // → [ 'Hola', 'hola', 'Hola' ]

// El carácter ^ significa 'principio de cadena'
str.match(/^Hola/g);
    // → [ 'Hola' ]

// Pero con el modificador 'm' significa 'principio de línea'
str.match(/^Hola/gm);
    // → [ 'Hola', 'Hola' ]
```

MÉTODOS DE CADENAS RELACIONADOS CON EXPRESIONES REGULARES

- `match(regex)`
Devuelve todas las subcadenas que ajustan con la expresión regular `regex` (si ésta contiene el modificador `g`) o solamente la primera (en caso contrario).
- `search(regex)`
Devuelve el índice de la primera subcadena que ajuste con `regex`, o -1 si no hay ninguna.
- `replace(regex, nuevaCadena)` Reemplaza por `nuevaCadena` las subcadenas que ajusten con `regex`.
- `split(regex)` Divide la cadena en fragmentos, utilizando `regex` como separador.

SOBRE LA FUNCIÓN `replace`

`replace(regex, nuevaCadena)`

La cadena de reemplazo (`nuevaCadena`) puede hacer referencia a los grupos de captura de la expresión regular.

```
var r = /(\d{4})\-([A-Z]{3})/;  
var str = "Mi número de matrícula es 9483-GSD";  
  
str.replace(r, "$2/$1");  
// → "Mi número de matrícula es GSD/9483"
```

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.
- Trigonómicas: `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.

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
```

1. INTRODUCCIÓN
2. JAVASCRIPT ES COMO JAVA...
3. ...PERO NO ES COMO JAVA
4. CLASES ESTÁNDAR
5. **HERRAMIENTAS**
6. BIBLIOGRAFÍA

UTILIDADES Y HERRAMIENTAS

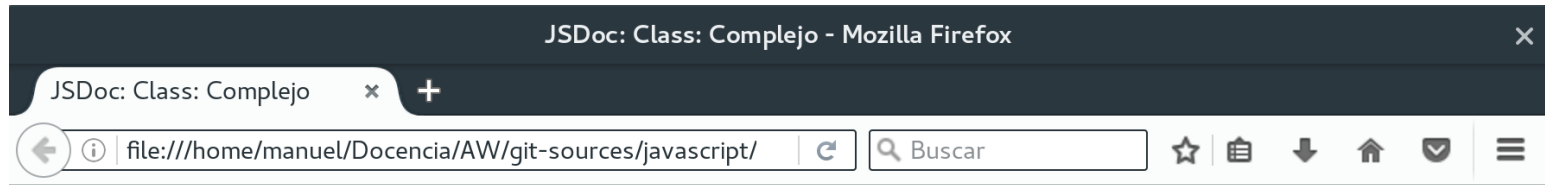
- JSDoc
<http://usejsdoc.org/>
- Depuradores
- Herramientas de
testing

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](#)

Methods

`(static) desdePolar(mod, arg)`

Home

Classes

Complejo

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
```

COMANDOS

<code>cont</code>	Salta al siguiente punto de ruptura
<code>step</code>	Avanzar paso (metiéndose dentro de
<code>next</code>	funciones o no)
<code>repl</code>	Arrancar <i>shell</i> para evaluar expresiones
<code>watch("...")</code>	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
}
```

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  
  2  
  4  
+ 8  
]
```

Sobra esto

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)

1. INTRODUCCIÓN
2. JAVASCRIPT ES COMO JAVA...
3. ...PERO NO ES COMO JAVA
4. CLASES ESTÁNDAR
5. HERRAMIENTAS
6. **BIBLIOGRAFÍA**



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>

