



# Introducción a JavaScript 6-7-8-9 (ES6 o ES2015, ES7 o ES2016, ..)

# Índice

- Funciones
- Arrays
- Objetos

A horizontal yellow bar at the top of the slide, consisting of two segments of different lengths joined together.

Funciones, array arguments, valores  
por defecto y operador spread

# Función

## Definición de la función

```
function my_preferred_movies () {  
  console.log();  
  console.log("My preferred movies:");  
  console.log(" - Jurassic Park by Steven Spielberg (1993)");  
  console.log(" - King Kong by Merian C. Cooper (1933)");  
  console.log(" - Citizen Kane by Orson Wells (1941)");  
  console.log();  
}
```

```
my_preferred_movies();
```

## Invocación (ejecución) de la función

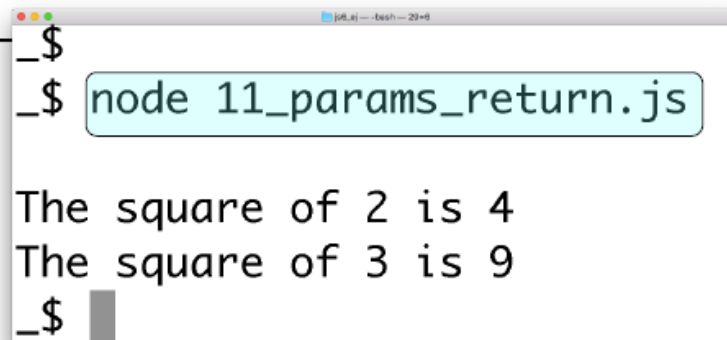
```
_ $  
_ $ node 10_function_movies.js  
  
My preferred movies:  
- Jurassic Park by Steven Spielberg (1993)  
- King Kong by Merian C. Cooper (1933)  
- Citizen Kane by Orson Wells (1941)  
  
_ $
```

Ejecución del programa  
10\_function\_movies.js  
con node.

- ◆ Una **función** encapsula código y lo representa por un **nombre**
  - Una función debe definirse primero, para poder invocarla (ejecutarla) posteriormente
    - ◆ Documentación: <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Funciones>
- ◆ La **definición** de la función comienza por la palabra reservada: **function**
  - A continuación viene el **nombre** de la función, que debe ser único en el programa
    - ◆ En tercer lugar vienen los parámetros entre paréntesis: ( ) indica sin parámetros en este ejemplo
      - Por último viene el bloque de código, entre corchetes { }
- ◆ La **invocación** de la función ejecuta el bloque de código de la función
  - Se invoca con el nombre y el operador paréntesis (), por ej. **my\_preferred\_movies()**

# Parámetros de invocación y de retorno

```
function square (x) {  
  return x*x ;  
}  
  
console.log();  
console.log("The square of " + 2 + " is " + square(2));  
console.log("The square of " + 3 + " is " + square(3));
```



A terminal window titled 'js:11 --- bash --- 20:0' showing the command 'node 11\_params\_return.js' being executed. The output of the command is 'The square of 2 is 4' followed by 'The square of 3 is 9'. The prompt character is '\_\$'.

```
_$_  
_$_ node 11_params_return.js  
  
The square of 2 is 4  
The square of 3 is 9  
_$_
```

- ◆ Una **función** recibe parámetros de entrada (parámetro **x** del ejemplo)
  - Y devuelve un valor con la sentencia: **return <expr>**
    - ◆ Esta sentencia finaliza la ejecución de la función y devuelve el valor resultante de evaluar **<expr>**
  - Si la **función** llega a final del bloque **sin ejecutar** return, finaliza y devuelve **undefined**
- ◆ Un **parámetro** de una función es similar a la definición de una **variable**
  - El parámetro solo es **visible** dentro del **bloque de la función**
    - ◆ El parámetro se **inicia** con el **valor pasado al invocar la función**, en el ejemplo con los valores 2 y 3
- ◆ Una función puede usarse en expresiones como otro valor más
  - La función se ejecutará y se sustituirá por el valor devuelto en la expresión
    - ◆ En el ejemplo (return **x\*x** ;) devuelve el **cuadrado** del valor pasado en el parámetro **x**

# Número de parámetros de una función

Antes de ES6 los strings se concatenaban así:  
**greeting + " " + person + ", how are you?"**  
(es prácticamente equivalente, pero menos compacto)

```
function greet (greeting, person) {  
  return `${greeting} ${person}, how are you?`;  
};
```

```
greet ("Good morning", "Peter");    // => "Good morning Peter, how are you?"
```

```
greet ("Hi", "Peter");              // => "Hi Peter, how are you?"
```

```
greet ("Hi", "Peter", "Bill");      // => "Hi Peter, how are you?"
```

```
greet ("Hi");                       // => "Hi undefined, how are you?"
```

```
greet ();                           // => "undefined undefined, how are you?"
```

## ◆ Una función **se puede invocar** con un **número variable de parámetros**

- Un parámetro definido, pero **no pasado** en la invocación, toma el valor **undefined**
  - ◆ Un parámetro pasado en la invocación, pero **no utilizado**, no tiene utilidad

## ◆ La función **greet(..)** genera un saludo utilizando 2 parámetros

- El ejemplo ilustra como procesa JavaScript parámetros no pasados o no utilizados

# arguments: el array con los parámetros

```
function greet () {  
    return `${arguments[0]} ${arguments[1]}, how are you?`;  
};  
  
greet ("Good morning", "Peter");    // => "Good morning Peter, how are you?"  
  
greet ("Hello", "Peter");           // => "Hello Peter, how are you?"
```

- ◆ Una función tiene predefinida un array de nombre **arguments**
  - **arguments** contiene los valores asignados a los parámetros en la invocación
    - ◆ Aquí se define la función **greet** utilizando **arguments** en vez de parámetros explícitos
      - Por último viene el bloque de código, entre corchetes {...}
- ◆ Una función se puede invocar con un número variable de parámetros
  - El array **arguments** permite saber su número y acceder a todos



# Resto de parametros en ES6: ...x

- ◆ Operador spread (...x) da acceso al resto de los parámetros de una función en ES6
  - Los parámetros están accesibles a través del array asociado al operador
    - ♦ Parámetros explícitos y operador rest pueden mezclarse entre sí, por ejemplo: `function f1 (x, y, ...resto) {...}`
      - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest\\_parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters)
      - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator)
- ◆ los ejemplos muestran 2 definiciones equivalentes de la función greet

```
function greet (...args) {  
  return `${args[0]} ${args[1]}, how are you?`;  
};
```

```
greet ("Good morning", "Peter");    // => "Good morning Peter, how are you?"  
greet ("Hello", "Peter");           // => "Hello Peter, how are you?"
```

```
function greet (greeting, ...more) {  
  return `${greeting} ${more[0]}, how are you?`;  
};
```


```
greet ("Good morning", "Peter");    // => "Good morning Peter, how are you?"  
greet ("Hello", "Peter");           // => "Hello Peter, how are you?"
```



# Valores por defecto de parámetros (ES6)

```
function greet (greeting = "Hi", person = "my friend") {  
    return `${greeting} ${person}, how are you?` ;  
};  
  
greet ("Hello");           // => "Hello my friend, how are you?"  
greet ();                  // => "Hi my friend, how are you?"
```

- ◆ **ES6** permite asignar valores por defecto a parámetros de funciones
  - Los valores por defecto se asignan al parámetro en la definición
    - ◆ utilizando el operador =, como en las definiciones de variables
- ◆ El valor por defecto se utiliza si la invocación no incluye ese parámetro

A horizontal yellow bar at the top of the slide, consisting of two segments of different lengths.

Arrays, spread y métodos sort,  
reverse, concat, join, indexOf, slice,  
splice, push y pop

# Arrays

## ◆ Array

- Es una colección ordenada de elementos
  - ◆ Se suele crear con el literal de array: [7, 4, 2, 23]
    - El operador corchetes agrupa elementos en arrays
- **toString()** devuelve un string con los elementos

## ◆ Los elementos de un array de tamaño n

- se acceden con un índice entre 0 y n-1
  - ◆ **a[k]** accede al elemento k+1

## ◆ **a.length** indica el tamaño del array

- Un array tiene un máximo de  $2^{32}-2$  elementos

## ◆ Cambiar **length** cambia el tamaño del array

- Por ejemplo, **a.length = 2** reduce el tamaño de **a** a 2
  - ◆ Quedando solo los dos primeros elementos

## ◆ El **operador spread (...x)** nuevo en ES6

- Inserta los elementos de un array en otro array

```
let a = [7, 4, 1, 23];
```

```
a[0]      => 7
```

```
a[1]      => 4
```

```
a[2]      => 1
```

```
a[3]      => 23
```

```
a.toString() => "7,4,1,23"
```

```
a.length  => 4
```

```
let a = [7, 4, 1, 23];
```

```
a.length = 2    => 2
```

```
a              => [7, 4]
```

```
let a = [7, 4, 1];
```

```
let b = [0, 0, ...a];
```

```
b              => [0, 0, 7, 4, 1]
```

```
b.length      => 5
```

# Métodos para ordenar, invertir, concatenar o buscar

## ◆ **sort()**

Estos métodos no modifican el array original, solo devuelven el resultado como parámetro retorno.

- devuelve el array ordenado

```
[1, 5, 3].sort() // => [1, 3, 5]
```

## ◆ **reverse()**

- devuelve el array invertido

```
[1, 5, 3].reverse() // => [3, 5, 1]
```

## ◆ **concat(e1, ..., en)**

- devuelve un nuevo array con **e1, ..., en** añadidos al final

```
[1, 5, 3].concat(9) // => [1, 5, 3, 9]  
[1, 5, 3].concat(9, 3) // => [1, 5, 3, 9, 3]
```

## ◆ **join(<separador>)**

- concatena elementos en un string
  - ◆ introduce <separador> entre elementos

```
[1, 5, 3, 7].join(';') // => '1;5;3;7'  
[1, 5, 3, 7].join('') // => '1537'
```

## ◆ **indexOf(elem, offset)**

- devuelve índice de primer **elem**
  - ◆ **offset**: comienza búsqueda (por defecto 0)

```
[1, 5, 3, 5, 7].indexOf(5) // => 1  
[1, 5, 3, 5, 7].indexOf(5, 2) // => 3
```

```
[1, 5, 3].concat(2).sort().reverse() // => [5, 3, 2, 1]
```

Los métodos encadenados aplican el segundo método sobre retorno del primero.

# Extraer, modificar o añadir elementos al array

◆ **slice(i,j):** devuelve la rodaja entre i y j

- Índice negativo (j) es relativo al final
  - ◆ índice "-1" es igual a a.length-2
- No modifica el array original

◆ **splice(i, j, e1, e2, ..., en)**

- sustituye j elementos desde i en array
  - ◆ por e1, e2, ...,en
- Devuelve rodaja eliminada

◆ **push(e1, ..., en)**

- añade e1, ..., en al final del array
  - ◆ devuelve el tamaño del array (a.length)

◆ **pop()**

- elimina último elemento y lo devuelve

```
[1, 5, 3, 7].slice(1, 2) => [5]
[1, 5, 3, 7].slice(1, 3) => [5, 3]
[1, 5, 3, 7].slice(1, -1) => [5, 3]
```

```
let a = [1, 5, 3, 7];
```

```
a.splice(1, 2, 9) => [5, 3]
a                 => [1, 9, 7]
```

```
a.splice(1,0,4,6) => []
a                 => [1, 4, 6, 9, 7]
```

```
let b = [1, 5, 3];
```

```
b.push(6, 7)      => 5
b                 => [1, 5, 3, 6, 7]
```

```
b.pop()           => 7
b                 => [1, 5, 3, 6]
```

A horizontal yellow bar at the top of the slide, consisting of two segments of different lengths joined together.

Arrays, spread/rest (...x) y asignación múltiple (destructuring assignment)

# Asignación múltiple en arrays (ES6)

- ◆ **ES6** añade una sentencia que asigna los elementos de un array a variables individuales
  - Se puede utilizar para asignar valores iniciales en definiciones de variables o en la asignación
    - ♦ Las variables deben agruparse entre corchetes y se relacionan por posición
      - Por ejemplo `let [x, y, z] = [5, 1, 3]` o `[y, z] = [4, 5]`
- ◆ La asignación múltiple puede utilizar valores por defecto
  - Por ejemplo `let [x, y, z=3] = [5, 1]`
- ◆ Se denomina asignación múltiple o también asignación desestructuradora (destructuring)
  - Permite hacer programas más cortos y legibles

```
// Inicializar con los 3  
// primeros elementos del  
// array
```

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

<b>x</b>	=>	5
<b>y</b>	=>	1
<b>z</b>	=>	3

```
// Intercambiar contenidos
```

```
let x = 5, y = 1;
```

```
[x, y] = [y, x];
```

<b>x</b>	=>	1
<b>y</b>	=>	5

```
// Con valores por defecto e  
// indefinidos
```

```
let [x, y, z=1, t=2, v] = [5, , ,10]
```

<b>x</b>	=>	5
<b>y</b>	=>	undefined
<b>z</b>	=>	1
<b>t</b>	=>	10
<b>v</b>	=>	undefined

Buen tutorial sobre destructuring assignment y spread/rest: <https://javascript.info/destructuring-assignment>

Destructuring assignment: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

Spread/rest syntax: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator)



# Operador spread/rest: ...x

## ◆ ES6 añade el operador spread/rest (...x)

- Tiene semántica spread (esparcir) o rest (resto) dependiendo del contexto

## ◆ Operador spread (...x) esparce los elementos del **array x** en otro array

- Actúa así cuando se aplica al constructor de array o en la invocación de una función
  - ♦ Por ejemplo, `[ x, ...y, z, ...t]` o `mi_funcion( x, ...y, z, ...t)`

## ◆ Operador rest (...x) agrupa un conjunto de valores en el **array x**

- Agrupa en un array el resto de los elementos asignados de una lista
  - ♦ Por ejemplo, `[ x, y, ...resto] = [ 1, 2, 3, 4, 5]` o `function f( x, y, ...resto) {..}`
    - La variable agrupadora debe ir al final y agrupa los últimos elementos de la lista

```
....  
const a = [2, 3];  
const b = [0, 1, ...a];  
b => [0, 1, 2, 3]  
  
f(0, 1, ...a) => f(0, 1, 2, 3)
```


```
let [x, y, ...rest] = [0, 1, 2, 3, 4];  
x      => 0  
y      => 1  
rest   => [2, 3, 4]  
  
function f(x, y, ...z) {....}  
f(0, 1, 2, 3) => f(0, 1, [2, 3])
```

```
let x, y, z;  
[y, z, ...x] = [2, 3, 4, 5];  
x      => [4, 5];  
y      => 2  
z      => 3
```

Buen tutorial sobre destructing assignment y spread/rest: <https://javascript.info/destructuring-assignment>

Destructing assignment: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

Spread/rest syntax: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator)

A horizontal yellow bar at the top of the slide, consisting of two segments of different lengths.

Iteradores de arrays (forEach, find, findIndex, filter, map y reduce),  
bucles for...of y for...in

# Métodos iteradores de Array: forEach

## ◆ Método iterador

- Métodos que ejecutan una función para cada elemento de un array (u objeto iterable)
  - ◆ La **función** recibe como parámetro los **elementos** del array que debe procesar en esa invocación
    - Empiezan por el elemento de índice **0** y lo van incrementando hasta llegar a **length-1**

## ◆ Los métodos iteradores equivalen a **bucles**

- Ejecutan cíclicamente la función iterando en cada elemento de un array (u objeto iterable)

## ◆ **forEach**(function(element, index, array){...}) o **forEach**((element, index, array)=>{...})

- Invoca la función con 3 parámetros: elemento actual, su índice y el array
  - ◆ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/forEach](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach)
- Estos dos ejemplos son equivalentes: ambos suman los elementos del array

```
let n = [7, 4, 1, 23];  
let add = 0;  
  
for (let i=0; i < n.length; ++i){  
  add += n[i];  
}  
  
add      // => 35 (7+4+1+23)
```

```
let n = [7, 4, 1, 23];  
let add = 0;  
  
n.forEach(elem => add += elem)  
  
add      // => 35
```

# Otros métodos iteradores de Array

◆ Estos métodos invocan la función también con los mismos 3 parámetros

- **elem**: elemento del array accesible en la invocación en curso
- **i**: índice al elemento del array accesible en la invocación en curso
- **a**: array completo sobre el que se invoca el método

◆ **find**(function(elem, i, a){...})

```
[7, 4, 1, 23].find(elem => elem < 3); // => 1
```

- devuelve el 1<sup>er</sup> elemento donde la función retorna true

- ♦ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/find](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find)

◆ **findIndex**(function(elem, i, a){...})

```
[7, 4, 1, 23].findIndex(elem => elem < 3); // => 2
```

- devuelve el índice del 1<sup>er</sup> elem. donde la función retorna true

- ♦ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/findIndex](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex)

◆ **filter**(function(elem, i, a){...})

```
[7, 4, 1, 23].filter(elem => elem > 5); // => [7, 23]
```

- elimina los elementos del array donde la función retorna false

- ♦ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/filter](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter)

◆ **map**(function(elem, i, a){...})

```
[7, 4, 1, 23].map(elem => -elem); // => [-7, -4, -1, -23]
```

- sustituye cada elemento del array por el resultado de invocar la función

- ♦ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map)

# Método reduce

- ◆ El método **reduce** añade el parámetro **accumulator** a element, index y array
  - **accumulator**: variable con valor retornado por invocación anterior de la función
    - ◆ además están los 3 parámetros típicos de los métodos iteradores: **element**, **index** y **array**
- ◆ **reduce**(function(accumulator, element, index, array){...}, initial\_value)
  - Inicializa accumulator con **initial\_value** e itera de **0** a **array.length-1**
    - ◆ **accumulator** recibe en cada nueva iteración el valor de retorno de la función
      - [https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\\_globales/Array/reduce](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/reduce)
  - si **initial\_value** se omite inicia accumulator con **array[0]** e itera de **1** a **array.length-1**

```
// Example of addition of numbers with reduce  
[7, 4, 1, 23].reduce((acc, elem) => acc += elem, 0);    // => 35
```

```
// Example which orders first the array and eliminates then duplicated numbers  
[4, 1, 4, 1, 4].sort().reduce((ac, el, i, a) => el !== a[i-1] ? ac.concat(el) : ac, []); // => [1, 4]
```

```
// sort(..) and reduce(..) are composed in series, where each one performs the following  
[4, 1, 4, 1, 4].sort(); // => [1, 1, 4, 4, 4]  
[1, 1, 4, 4, 4].reduce((ac, el, i, a) => el !== a[i-1] ? ac.concat(el) : ac, []); // => [1, 4]
```



# Bucles for...in y for...of

## ◆ JavaScript incluye el bucle **for...in** que itera en las propiedades de un objeto

- El bucle **for...of** de **ES6** itera con una función generadora en los elementos de un objeto iterable
  - ♦ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of>
  - ♦ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in>
- Los arrays son objetos y son iterables por lo que pueden procesarse con ambos bucles

## ◆ La sentencia **for (prop in object) {..bloque..}** (**object** es un array u objeto)

- Ejecuta el bloque para cada **propiedad** (accesible en la variable **prop**) del objeto o array
  - ♦ Los **índices** de los elementos de un array son **propiedades** especiales (su nombre es el número)

## ◆ La sentencia **for (elem of object) {..bloque..}** (**object** debe ser un obj. o array iterable)

- Ejecuta el bloque para cada **elemento** (accesible en la variable **elem**) del objeto o array
  - ♦ **object** debe ser un iterable que define el orden del recorrido
    - Por ejemplo, en un array el recorrido empieza en el elemento de índice **0** y termina en el de **length-1**
- Estos 2 ejemplos de suma de elementos de Array con los nuevos bucles equivalen a los 3 ya vistos

```
let n = [7, 4, 1, 23];  
let add = 0;
```

```
for (let i in n){  
  add += n[i];  
}
```

```
add // => 35
```

```
let n = [7, 4, 1, 23];  
let add = 0;
```

```
for (let elem of n){  
  add += elem;  
}
```

```
add // => 35
```

```
let n = [7, 4, 1, 23];  
let add = 0;
```

```
for (let i=0; i < n.length; ++i){  
  add += n[i];  
}
```

```
add // => 35
```

```
let n = [7, 4, 1, 23];  
let add = 0;
```

```
n.forEach(elem => add += elem)
```

```
add // => 35
```

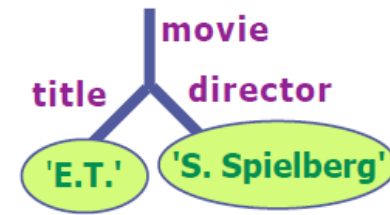
```
[7, 4, 1, 23].reduce((acc, elem) => acc += elem, 0); // => 35
```

A solid yellow horizontal bar spanning the width of the slide, positioned above the text.

Objetos, propiedades, métodos  
propios y this.



# Propiedades de un objeto



## ◆ Objeto (solo con propiedades)

- **Agregación de variables** (denominadas **propiedades**)
- Suelen crearse con el literal de objeto
  - ◆ { propiedad\_1:valor\_1, ....., propiedad\_n:valor\_n }

## ◆ Los nombres de propiedades de un objeto

- deben ser **todos diferentes**
- deben tener la misma **sintaxis que las variables**
  - ◆ a, \_method, \$1, una\_piña, ....

## ◆ Operador punto

- **objeto.propiedad**
  - ◆ Accede al contenido de propiedades por **nombre**

## ◆ Operador array

- **objeto["propiedad"]**
  - ◆ La propiedad puede ser un string en una variable
    - **ES6** permite incluir expresiones arbitrarias

## ◆ Notación array extiende la notación punto

## ◆ Propiedades inexistentes devuelven undefined

- Pero el operador punto (.) aplicado a **undefined**
  - ◆ provoca **error de ejecución**

```
var movie = {title:'E.T.', director:'S. Spielberg'};
```

```
// Access to properties
```

```
movie.title           // => 'E.T.'
movie.director        // => 'S. Spielberg'
movie['title']         // => 'E.T.'
movie['director']      // => 'S. Spielberg'
```

```
// Access by means of variables with [..]
```

```
var t = 'title';      // contains string 'title'
movie[t]              // => 'E.T.'
movie['ti' + 'tle']    // => 'E.T.'
movie.t               // => undefined
```

```
// nonexistent properties are undefined
```

```
movie.premiere        // => undefined
movie['premiere']      // => undefined
```

```
// Execution errors
```

```
undefined.t           // => error, program stops
undefined[t]          // => error, program stops
```

# Nombres extendidos de propiedades

## ◆ Nombre extendido de propiedad

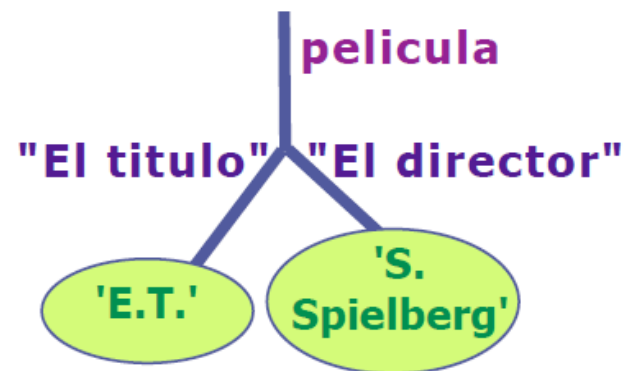
- Es un string arbitrario que **no sigue las reglas sintácticas de las variables**, es decir
  - ◆ Empezar por **letra**, **\_** o **\$** y continuar por alguno de estos caracteres o **dígitos decimales**

## ◆ Utilizando literales de objeto y notación array

- Es posible manejar objetos con **nombres extendidos** de propiedades
  - ◆ La notación punto ('.') solo permite nombre con sintaxis de variable

## ◆ El literal de objeto permite crear objetos

- Utilizando strings con **nombres extendidos** de propiedades
  - ◆ {"El titulo": 'E.T.', "El director": 'S. Spielberg'}



## ◆ La notación array es otra forma de referenciar propiedades

- Puede utilizar **nombres extendidos** de propiedades
  - ◆ película["El director"], objeto[""] o a["%43"]
- Los **índices de arrays** son nombres especiales de propiedades de un objeto array
  - ◆ Por ejemplo, el elemento de índice 2 de un array se referencia como: a[2] o a["2"]

## ◆ OJO! normalmente es conveniente utilizar nombres para notación punto

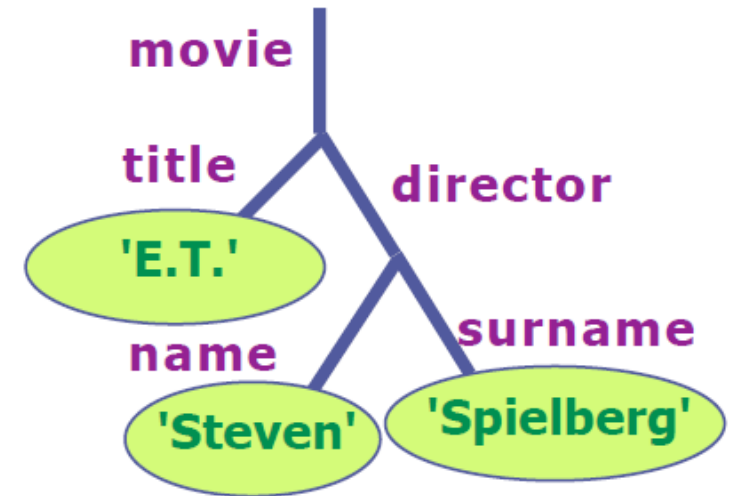
- Strings arbitrarios pueden ser útiles en objetos tipo diccionario o similares

# Objetos anidados: árboles

- ◆ Los objetos pueden **anidarse** entre sí
  - Los objetos anidados representan **árboles**
- ◆ La notación punto o array puede **encadenarse**
  - Representando un **camino en el árbol**
    - ◆ Las siguientes expresiones se evalúan así:

■ movie.title	=> 'E.T.'
■ movie.director.name	=> 'Steven'
■ movie['director']['name']	=> 'Steven'
■ movie['director'].surname	=> 'Spielberg'
■ movie.director	=> {name:'Steven', surname: 'Spielberg'}
■ movie.premiere	=> undefined
■ movie.premiere.year	=> <b>Error_de_ejecución</b>

```
var movie = {  
  title: 'E.T.',  
  director: {  
    name: 'Steven',  
    surname: 'Spielberg'  
  }  
};
```



# Propiedades dinámicas

## ◆ Las propiedades se pueden crear y destruir

- Para ello se utilizan 3 sentencias
  - ◆ Asignación de valores
  - ◆ Borrado de propiedades
  - ◆ Comprobar si existe una propiedad

## ◆ Asignar a (y crear) propiedades: **x.c = 4**

- asigna 4 -> si la propiedad c **existe**
- **crea c** y le asigna 4 -> si la propiedad c **no existe**

## ◆ Borrar propiedades:

- elimina x.c -> si la propiedad x.c **existe**
- no hace nada -> si la propiedad x.c **no existe**

## ◆ ¿Existe la propiedad?:

- devuelve **true** -> si la propiedad x.c **existe**
- devuelve **false** -> si la propiedad x.c **no existe**

La propiedad ya existe y **solo cambia el valor** a 7

La propiedad no existe y **se crea** con el valor 5

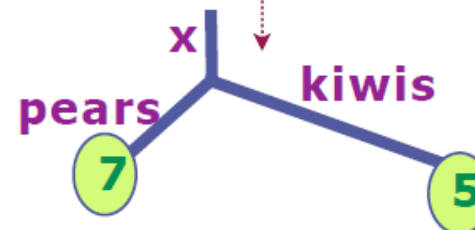
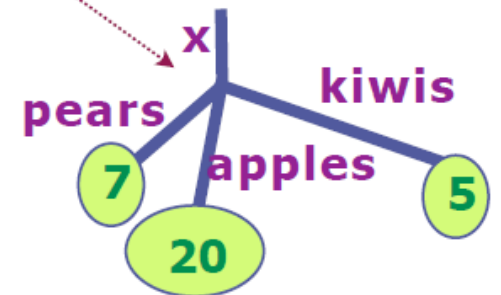
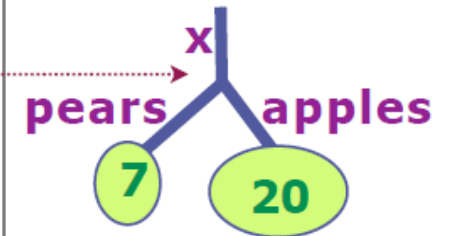
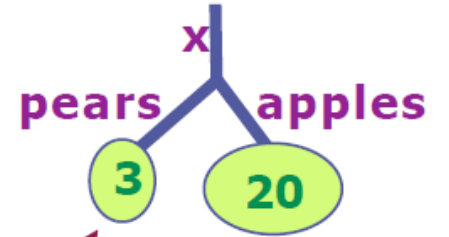
```
var x = { pears:3, apples:20};
```

```
x.pears = 7;
```

```
x.kiwis = 5;
```

```
delete x.apples;
```

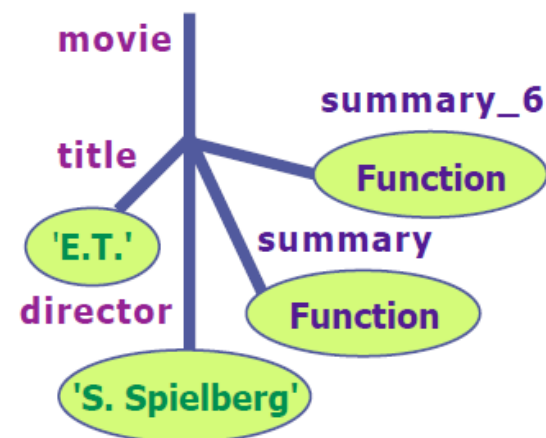
La propiedad se destruye



# Definición de métodos propios

- ◆ Un **método propio** es una función que se guarda en una propiedad del objeto
  - Un método propio **solo existe** en el **objeto en el que ha sido definido**
    - ♦ Se invoca sobre ese objeto con los operadores punto y paréntesis, por ej. `movie.summary()`
- ◆ **this** es una referencia al objeto sobre el que se invoca el método
  - En el ejemplo, **this.title** referencia la propiedad **title** del objeto **movie**
    - ♦ **this** puede omitirse si no hay ambigüedad y en el ejemplo podría utilizarse solo **title** o **director**
      - <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this>
- ◆ **ES6** añade una sintaxis simplificada que omite ":" y "function"
  - Por ejemplo, **summary\_6** define un método equivalente a **summary** con sintaxis **ES6**

```
var movie = {  
  title: 'E.T.',  
  director: 'S. Spielberg',  
  summary: function() {  
    return "The director of " + this.title + " is " + this.director;  
  },  
  summary_6 () {  
    return "The director of " + this.title + " is " + this.director;  
  }  
}  
  
movie.summary() // => "The director of E.T. is S. Spielberg"  
movie.summary_6() // => "The director of E.T. is S. Spielberg"
```



Estos dos métodos se denominan **propios** porque se han definido directamente en un objeto y solo se pueden invocar en él.



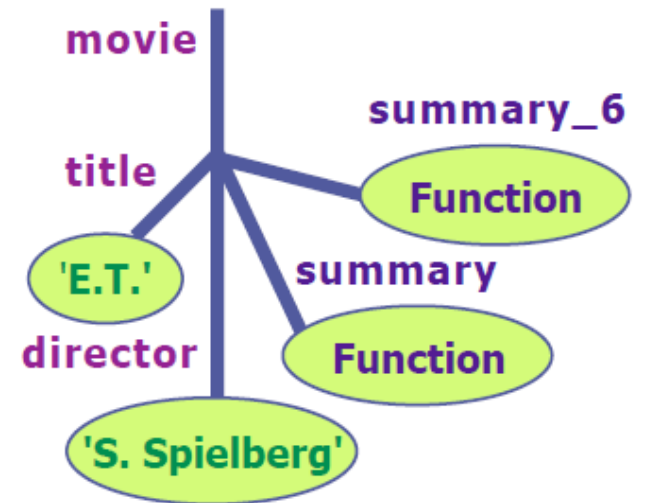
# Creación dinámica de métodos

- ◆ Un **método propio** se puede añadir también dinámicamente a un objeto
  - Debe añadirse una propiedad con la función correspondiente, por ejemplo
    - ◆ `movie.summary = function(){ return "The director of " + this.title + " is " + this.director};`
- ◆ Los métodos `summary` y `summary_6` del objeto `movie` son similares
  - La única diferencia es que se han creado dinámicamente y no con el literal


```
var movie = {  
  title: 'E.T.',  
  director: 'S. Spielberg',  
}
```

```
movie.summary = function() {  
  return "The director of " + this.title + " is " + this.director;  
};
```

```
movie.summary() // => "The director of E.T. is S. Spielberg"
```



Estos dos métodos siguen siendo métodos **propios** porque se han definido sobre este objeto y solo se pueden invocar en él.

A solid yellow horizontal bar spanning the width of the slide, positioned above the text.

Objetos: Literal de ES6, multi-  
asignación, spread/rest (...x),  
for...in y Object.keys(..)



# El literal de objetos ES6: agrupar variables

- ◆ La agrupación de variables en ES5 se realiza con el literal de objetos
  - Por ejemplo, `var obj = {a:a, b:b, c:c}` agrupa en un objeto las variables a, b y c
- ◆ ES6 también permite agrupar o estructurar objetos de forma mas concisa
  - Por ejemplo, `var obj = {a, b, c}` es equivalente en ES6 a lo anterior
    - ◆ El literal de objeto permite incluir solo el nombre de la variable, cuando esta inicializa una propiedad del mismo nombre con la variable en un objeto

```
let a=5, c=3, d=4;
```

```
let obj_ES5 = {a:a, c:c, d:d};
```

```
obj_ES5      => {a:5, c:3, d:4}
```

```
// ES5: agrupar variables en un objeto con  
// propiedades del mismo nombre de las variables
```

```
let obj_ES6 = {a, c, d};
```

```
obj_ES6      => {a:5, c:3, d:4}
```

```
// ES6: Las mismas variables se agrupan así
```

# Asignación múltiple o destructuración

- ◆ La multi-asignación de **ES6** se puede aplicar también a objetos
  - En este caso asigna varias propiedades a variables del mismo nombre
    - ◆ En inglés se denomina 'destructuring', que se ha traducido por destructor
- ◆ Variables y valores asignados se **relacionan por nombre**
  - Las variables a asignar se agrupan con llaves y pueden llevar valores por defecto
    - ◆ Por ejemplo `let {a, b} = {a:5, b:1}` o `({a, b} = {a:1, b:2})`

```
let {a, c=1, d, e} = {a:5, e:3, f:4};
```

<b>a</b>	=>	5
<b>c</b>	=>	1
<b>d</b>	=>	undefined
<b>e</b>	=>	3

```
let a, c, d;
```

```
({a, c=1, d} = {a:5, c, d, e:3});
```

<b>a</b>	=>	5
<b>c</b>	=>	1
<b>d</b>	=>	undefined

La multi-asignación debe ir entre paréntesis por un problema del análisis sintáctico de JavaScript.

Buen tutorial sobre destructuring assignment y spread/rest: <https://javascript.info/destructuring-assignment>

Destructuring assignment: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

Spread/rest syntax: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator)

# Operador rest/spread (...x) para objetos

◆ El operador **rest** (...x) también puede utilizarse con la asignación múltiple de objetos

- Por ejemplo `let {a, ...x} = {a:5, b:1, c:2}`    o    `({a, ...x} = {a:1, b:2})`

```
let {a, ...x} = {a:5, b:1, c:2};
```

a	=>	5
x	=>	{b:1, c:2}

```
let {a, ...x} = {a:5, b:1, c:2};
```

```
({a, ...x} = {a:1, b:2});
```

a	=>	1
x	=>	{b:2}

La multi-asignación debe ir entre paréntesis por un problema del análisis sintáctico de JavaScript.

◆ El operador **spread** (...x) también puede utilizarse para esparcir propiedades en un objeto

- Por ejemplo `let x = {a:5, b:1}`    y    `let y = {...x, c:6, d:7}`

```
let x = {a:5, b:1};
```

```
let y = {...x, c:6, d:7};
```

```
y            =>    {a:5, b:1, c:6, d:7}
```

Buen tutorial sobre destructuring assignment y spread/rest: <https://javascript.info/destructuring-assignment>

Destructuring assignment: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

Spread/rest syntax: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator)

# Multi-asignación con objetos anidados

- ◆ La multi-asignación permite también **objetos anidados**, así como **cambiar nombres**
  - El **nombre de la variable** creada o asignada es el de la **propiedad** que casa en la parte izquierda

```
let {x:a, y} = {x:1, y:10}
```

```
a      => 1
```

```
y      => 10
```

```
let {x, y:{y}} = {x:1, y:{t:10, y:11}}
```

```
x      => 10
```

```
y      => 11
```

- ◆ Formato general permite además **cambios de nombre** y valor por defecto
  - El **nuevo nombre de la variable** debe añadirse como **valor de la propiedad** a la izquierda
  - Y el **valor por defecto** debe asignarse con **=** al **nombre de la variable** a asignar

```
let {x:a, y:{t=7}, y:{y:b}, z:c=4} = {x:1, y:{t:10, y:11}, u:3}
```

```
a      => 1
```

```
t      => 10
```

```
b      => 11
```

```
c      => 4
```

# Sentencias `for...in` y `Object.keys(obj)`

## ◆ Sentencia `for (let p in obj) {..bloque de instrucciones..}`

- Itera en todas las propiedades de `obj`, siguiendo el orden de inserción de propiedades
  - ◆ En cada iteración `p` contiene el nombre (string) de la propiedad para acceso con `obj[p]`
- `for (let elem of obj) {...}` solo permite objetos iterables y no se utilizar con `{a:7, b:4, c:1, d:23}`
- La sentencia itera en las propiedades enumerables del objeto y de sus prototipos
  - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in>

## ◆ `Object.keys(obj)` extrae un array con los nombres de las propiedades

- Permite utilizar los iterados de arrays con objetos
  - ◆ Normalmente conviene utilizar `Object.keys(...)`, en vez de `for (let p in obj) {...}`, porque el array devuelto por `Object.keys` contiene solo las propiedades enumerables propias del objeto, no incluye las de sus prototipos

```
let obj = {a:7, b:4, c:1, d:23};  
let add = 0;
```

```
for (let p in obj) {  
    add += obj[p];  
}
```

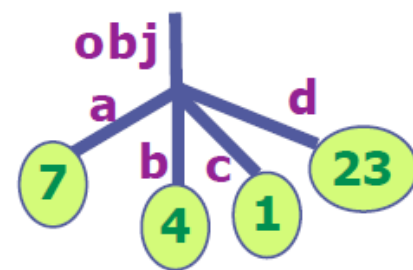
```
add // => 35
```

```
let obj = {a:7, b:4, c:1, d:23};  
let add = 0;
```

```
Object.keys(obj); // => ["a", "b", "c", "d"]
```

```
Object.keys(obj).forEach(p => add += obj[p]);
```

```
add // => 35
```



A horizontal yellow bar at the top of the slide, consisting of two segments of different lengths.

# JSON: JavaScript Object Notation



# Serialización de datos: JSON

## ◆ Serialización de datos

- transformación **reversible** de valores en un string equivalente
- Facilita el almacenamiento y envío de datos, por ejemplo
  - ◆ Almacenar datos en un fichero
  - ◆ Enviar datos a través de una línea de comunicación
  - ◆ Paso de parámetros en interfaces REST

## ◆ JSON - JavaScript Object Notation

- Formato de serialización de valores y objetos JavaScript
  - ◆ Cubre las partes más importantes de los objetos JavaScript
    - <http://json.org/json-es.html>

## ◆ Existen otros formatos de serialización: XML, HTML, XDR(C), ...

- Estos formatos están siendo desplazados por JSON, incluso XML
  - ◆ Existen bibliotecas de JSON para los lenguajes más importantes



# Objeto global JSON

◆ JavaScript tiene el objeto global JSON con métodos de conversión

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/JSON](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON)

◆ **JSON.stringify(<object>)**

- El método **stringify** transforma un objeto (<object>) en un string JSON equivalente

◆ **JSON.parse(<string>)**

- El método **parse** transforma un <string> JSON en el objeto o valor equivalente

<b>JSON.stringify(null)</b>	// => <b>'null'</b>
<b>JSON.parse('null')</b>	// => <b>null</b>
<b>JSON.stringify(127)</b>	// => <b>'127'</b>
<b>JSON.stringify('hola')</b>	// => <b>""hola""</b>
<b>JSON.stringify([1, 2, 3])</b>	// => <b>'[1, 2, 3]'</b>
<b>JSON.stringify({a:27, b:"hola"})</b>	// => <b>'{"a":27,"b":"hola"}'</b>

# Características de JSON

## ◆ JSON puede serializar

- objetos, arrays, strings, números finitos, true, false y null
  - ♦ NaN, Infinity y -Infinity se serializan por defecto a null
  - ♦ Los objetos Date se serializan como un string en formato ISO 8601
    - la reconstrucción devuelve un string y no el objeto original
- No se puede serializar
  - ♦ Funciones, RegExp, errores, undefined

## ◆ parse y stringify admiten filtros para los elementos no soportados

- ver doc de APIs JavaScript:
  - ♦ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/JSON](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON)

```
JSON.stringify(new Date())    => '"2013-08-08T17:13:10.751Z"'
```

```
JSON.stringify(NaN)          => 'null'
```

```
JSON.stringify(Infinity)     => 'null'
```

# Ejemplo de datos en JSON

- ◆ **JSON** es un formato flexible y legible de datos muy utilizado
  - permite insertar espacios en blanco y retorno de línea entre los símbolos
    - ◆ El siguiente ejemplo muestra un array con 4 objetos en JSON

```
[ { "title": "E.T.",  
    "director": "Steven Spielberg"  
  },  
  { "title": "Star Wars",  
    "director": "George Lucas"  
  },  
  { "title": "Psicosis",  
    "director": "Alfred Hitchcock"  
  },  
  { "title": "Plácido",  
    "director": "Luis García Berlanga"  
  }  
]
```