ÍNDICE

Funciones predefinidas del lenguaje

- JavaScript cuenta con una serie de funciones integradas en el lenguaje.
- Dichas funciones se pueden utilizar sin conocer todas las instrucciones que ejecuta.
- Simplemente se debe conocer el nombre de la función y el resultado que se obtiene al utilizarla.

Funciones predefinidas del lenguaje

- Las siguientes son algunas de las principales funciones predefinidas de JavaScript:

Funciones Predefinidas	
`encondeURI()`	`Number()`
`eval()`	`String()`
`isFinite()`	`parseInt()`
`isNaN()`	`parseFloat()`

Funciones predefinidas del lenguaje

Este método sustituye a escape()

- **`encodeURI`** ( ) : recibe como argumento una cadena de caracteres y devuelve un identificador de recursos uniforme válido. Excepto: , / ? : @ & = + $ #.

  Se debe usar con URI completas que puede tener algunos caracteres que deben codificarse.

```
var url =encodeURI("http://www.example.org/a file with spaces.html");
```

```
"http://www.example.org/a%20file%20with%20spaces.html"
```

Si usamos encodeURIComponent "destrozará" la url completa:

```
http%3A%2F%2Fwww.example.org%2Fa%20file%20with%20spaces.html
```

-- Existe también la función **decodeURI**() que es la función opuesta a encodeURI(). Esta función decodifica los caracteres que estén codificados.

Funciones predefinidas del lenguaje

- **`encodeURIComponent`** ( ) : recibe como argumento una cadena de caracteres y devuelve un identificador de recursos uniforme válido.

Se usa cuando se quiere codificar el valor de un parámetro de una URL

```
var p1 = encodeURIComponent("http://example.org/?a=12&b=55")
```

```
var url = "http://example.net/?param1=" + p1 + "&param2=99";
```

```
http://example.net/?param1=http%3A%2F%2Fexample.org%2F%Ffa%3D12%26b%3D55&param2=99
```

-- Existe también la función **decodeURIComponent**() que es la función opuesta a encodeURIComponent().
Esta función decodifica los caracteres que estén codificados.

Funciones predefinidas del lenguaje

- **`eval`** ( ) : convierte una cadena que pasamos como argumento en código JavaScript ejecutable.

```
<script type="text/javascript">
  var input = prompt("Introduce una operación numérica");
  var resultado = eval(input);
  alert ("El resultado de la operación es: " + resultado);
</script>
```

```
<script type="text/javascript">
  var string1 = "foo";
  var string2 = "bar";
  var funcName = string1 + string2;

 function foobar(){ alert( 'Hello World' ); }
 eval( funcName + '()' );  // Hello World
</script>
```

Funciones predefinidas del lenguaje

- **`isFinite`** ( ) : verifica si el número que pasamos como argumento es o no un número finito (ni infinity, ni -infinity ni NaN).

```
if(isFinite(argumento)) {
  //instrucciones si el argumento es un número finito
}else{
  //instrucciones si el argumento no es un número finito
}
```

Funciones predefinidas del lenguaje

- **`isNaN`**`()` : comprueba si el valor que pasamos como argumento es un de tipo numérico.

```
<script type="text/javascript">
  var input = prompt("Introduce un valor numérico: ");
  if (isNaN(input)){
    alert("El dato ingresado no es numérico.");
  }else{
    alert("El dato ingresado es numérico.");
  }
</script>
```

Funciones predefinidas del lenguaje

- **`String`**`()` : convierte el objeto pasado como argumento en una cadena que represente el valor de dicho objeto.

```
<script type="text/javascript">
  var fecha = new Date()
  var fechaString = String(fecha)
  alert("La fecha actual es: "+fechaString);
</script>
```

Funciones predefinidas del lenguaje

- **`Number`** ( ) : convierte el objeto pasado como argumento en un número que represente el valor de dicho objeto.
- Si el parámetro es un objeto de tipo Date, la función devolverá el número de milisegundos transcurridos desde la medianoche del 1 de enero de 1970.
- Si la conversión falla, devolverá NaN

```
<script>
  function myFunction() {
  var n =
  Number(true) + "<br>" +
  Number(false) + "<br>" +
  Number(new Date()) + "<br>" +
  Number("999") + "<br>" +
  Number("999 888");
  document.write(n);
}
</script>
```

1
0
1446031755055
999
NaN

Funciones predefinidas del lenguaje

- **`parseInt`** ( ) : convierte la cadena que pasamos como argumento en un valor numérico de tipo entero con la base especificada. Por defecto, la base 10.

```
parseInt(" 0xF", 16);       parseInt(" F", 16);
parseInt("17", 8);          parseInt("15px", 10);
parseInt("015", 10);        parseInt(15.99, 10);
parseInt("1111", 2);        parseInt("15*3", 10);
parseInt("15e2", 10);       parseInt("12", 13);
```

Todos devuelven 15

```
Devuelve NaN:
parseInt("Hello", 8); // Not a number at all
parseInt("546", 2); // Digits are not valid for binary representations
```

Funciones predefinidas del lenguaje

- **`parseFloat`** ( ) : convierte la cadena que pasamos como argumento en un valor numérico de tipo flotante.

```
<script type="text/javascript">
  var a = parseFloat("10") + "<br>";
  var b = parseFloat("10.00") + "<br>";
  var c = parseFloat("10.33") + "<br>";
  var d = parseFloat("34 45 66") + "<br>";
  var e = parseFloat(" 60 ") + "<br>";
  var f = parseFloat("40 years") + "<br>";
  var g = parseFloat("He was 40") + "<br>";
</script>
```

parseFloat("10") = 10
parseFloat("10.00") = 10
parseFloat("10.33") = 10.33
parseFloat("34 45 66") = 34
parseFloat("60") = 60
parseFloat("40 years") = 40
parseFloat("He was 40") = NaN

Funciones del usuario

- Es posible crear funciones personalizadas diferentes a las funciones predefinidas por el lenguaje.
- Con estas funciones se pueden realizar las tareas que queramos.
- Una tarea se realiza mediante un grupo de instrucciones relacionadas a las cuales debemos dar un nombre.
- Lo ideal: Una función - Una tarea

Funciones del usuario

- **Definición de funciones:**
  - El mejor lugar para definir las funciones es dentro de las etiquetas HTML `<head>` y `</head>`.
  - El motivo es que el navegador carga siempre todo lo que se encuentra entre estas etiquetas.
  - La definición de una función consta de cinco partes:
    - La palabra clave `function`.
    - El nombre de la función.
    - Los parámetros/argumentos utilizados.
    - El grupo de instrucciones.
    - La palabra clave `return`.

Funciones del usuario

- Definición de funciones – Sintaxis:

```
function nombre_función ([argumentos]){
      grupo_de_instrucciones;
     [return valor;]
}
```

Funciones del usuario

- Definición de funciones – Nombre:
  - El nombre de la función se sitúa al inicio de la definición y antes del paréntesis que contiene los posibles argumentos.
    - Deben usarse sólo letras, números o el carácter de subrayado.
    - Debe ser único en el código JavaScript de la página web.
    - No pueden empezar por un número.
    - No puede ser una de las palabras clave/reservadas del lenguaje.

Funciones del usuario

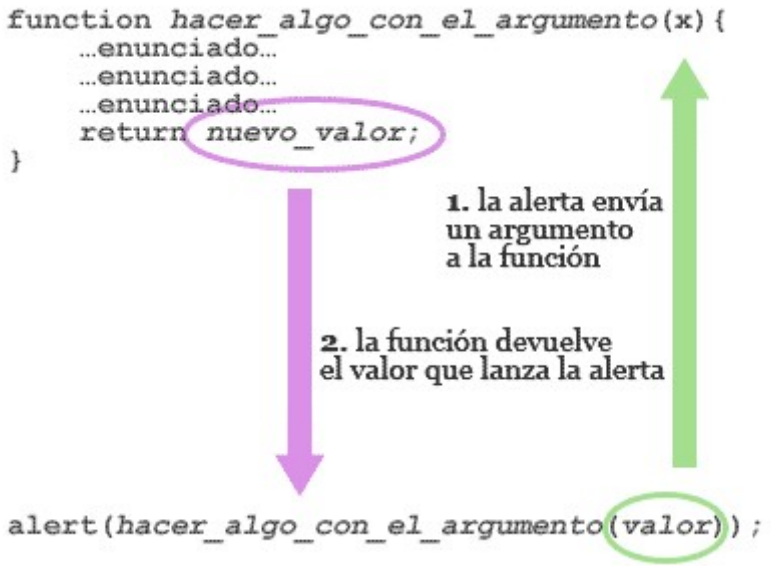
- Definición de funciones – Parámetros:
  - Los parámetros se definen dentro del paréntesis situado después del nombre de la función.
  - No todas las funciones requieren parámetros, con lo cual el paréntesis se deja vacío.

Funciones del usuario

- Definición de funciones – Grupo de instrucciones:
  - El grupo de instrucciones es el bloque de código JavaScript que se ejecuta cuando invocamos a la función desde otra parte de la aplicación.
  - Las llaves { } delimitan el inicio y el fin de las instrucciones.

Funciones del usuario

- Definición de funciones – `return`:
  - La palabra clave `return` es opcional en la definición de una función.
  - Indica al navegador que devuelva un valor a la sentencia que haya invocado a la función. Si no se pone devuelve "undefined"



Funciones del usuario

- Definición de funciones – `return`:
  - `return` detiene la ejecución de la función

```
function isPrime(integer){
    for (var x = 2; x < integer; x++){
        if (integer % x === 0) {
            console.log (integer + "is divisible by " + x);
            return false;
        }
    }
    return true;
}
```

Funciones del usuario

- Invocación de funciones:
  - Una vez definida la función es necesaria llamarla para que el navegador ejecute el grupo de instrucciones.
  - Se invoca usando su nombre seguido del paréntesis.
  - Si tiene argumentos, se deben especificar en el mismo orden en el que se han definido en la función.
  - Los parámetros primitivos se le pasan a las funciones **siempre por valor** , es decir, si la función modifica el valor que se le pasa ésto no afecta a la variable al terminar la ejecución de la función. Si nos interesa que la función devuelva algún valor debe haber una sentencia con la palabra reservada return seguida del valor a devolver.

    Si le pasas un objeto (p. ej. un valor no primitivo, como un Array o un objeto definido por el usuario) como parámetro, y la función cambia las propiedades del objeto, este cambio es visible desde fuera de la función.

Funciones del usuario

```
function myFunc(theObject) {
  theObject.make = "Toyota";
}

var mycar = {make: "Honda", model: "Accord", year: 1998},
    x,
    y;

x = mycar.make;     // x gets the value "Honda"

myFunc(mycar);
y = mycar.make;     // y gets the value "Toyota"
                    // (the make property was changed by the function)
```

```
function myFunc(theObject) {
  theObject = {make: "Ford", model: "Focus", year: 2006};
}

var mycar = {make: "Honda", model: "Accord", year: 1998},
    x,
    y;

x = mycar.make;     // x gets the value "Honda"

myFunc(mycar);
y = mycar.make;     // y still gets the value "Honda"
```

Funciones del usuario

- Invocar una función desde JavaScript:

```
<html>
  <head>
    <title>Invocar función desde JavaScript</title>
    <script type="text/javascript">
      function mi_funcion([args]){
        //instrucciones
      }
    </script>
  </head>
  <body>
    <script type="text/javascript">
      mi_funcion([args]);
    </script>
  </body>
</html>
```

Funciones del usuario

- Invocar una función desde HTML:

```
<html>
  <head>
    <title>Invocar función desde JavaScript</title>
    <script type="text/javascript">
      function mi_funcion([args]){
        //instrucciones
      }
    </script>
  </head>
  <body onload="mi_funcion([args])"></body>
</html>
```

Funciones del usuario

- Invocar una función desde otra función:

```
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Document</title>
   <script type="text/javascript">
     function comprobarHumano(){
          var resultado = prompt("Introduce el resultado de 120/6: ");
          var humano = verificacion(resultado);
          if(humano){ document.write("Correcto");}
          else{ alert("Error");}
     }

     function verificacion(res){
          var compResult;
          if(res == 20){  compResult = true;      }
          else{compResult = false;}
          return compResult;
     }
    </script>
</head>
<body>
          <script type="text/javascript">
                    comprobarHumano();
          </script>
</body>
</html>
```

Funciones del usuario

- Invocar una función:

```
<SCRIPT>
miFuncion()
function miFuncion(){
    //hago algo...
    document.write("Esto va bien")
}
</SCRIPT>
```

OK

```
<HTML>
<HEAD>
   <TITLE>MI PÁGINA</TITLE>
<SCRIPT>
function miFuncion(){
    //hago algo...
    document.write("Esto va bien")
}
</SCRIPT>
</HEAD>
<BODY>

<SCRIPT>
miFuncion()
</SCRIPT>

</BODY>
</HTML>
```

OK

```
<SCRIPT>
miFuncion()
</SCRIPT>

<SCRIPT>
function miFuncion(){
    //hago algo...
    document.write("Esto va bien")
}
</SCRIPT>
```

ERROR

Funciones del usuario

- Funciones con número variable de argumentos - ES5:
  - **Arguments** →pseudo-array que contiene todos los argumentos de la función (obligatorios y opcionales)
  - nombreFuncion.**length** →devuelve el número de parámetros de la función
  - arguments.**length** → devuelve el número de argumentos pasados

```
for (n = 0; n < ArgTest.length; n++)
{
     document.write(ArgTest.arguments[n], "<br>");
}
```

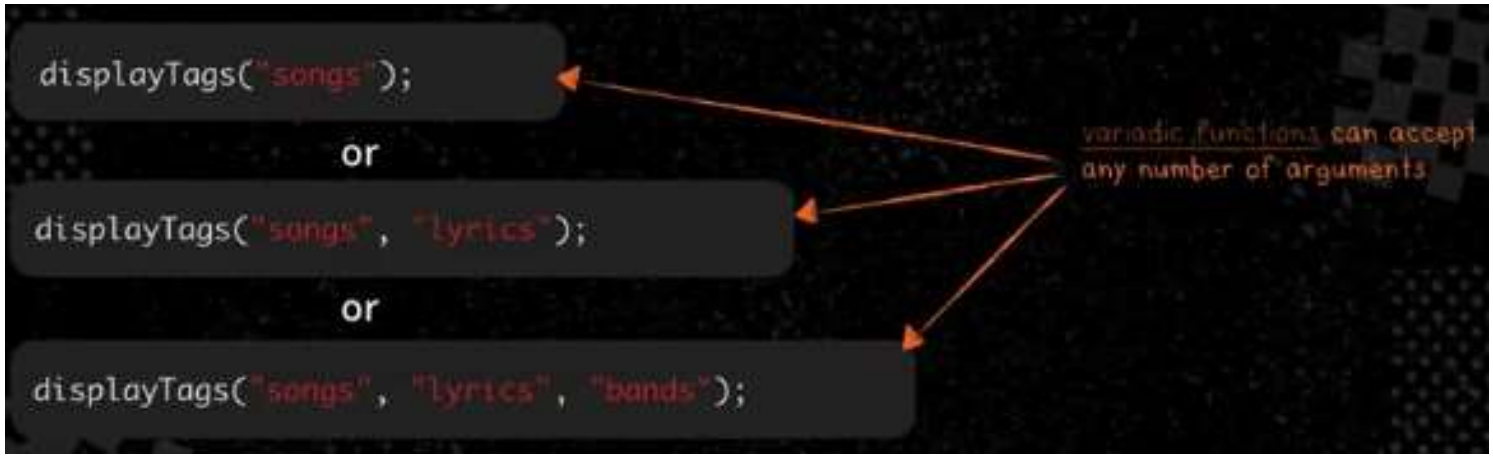
→ Obligatorios:
nº parámetros función
nombreFuncion.length

```
for (n = ArgTest.length; n < arguments.length; n++)
{
   document.write (ArgTest.arguments[n], "<br>");
}
```

→ Optativos:
nº argumentos - nº de parámetros

Funciones usuario - Novedades ES6

- **Funciones con número de parámetros variable**



Funciones usuario - Novedades ES6

- **Funciones con número de parámetros variable (parámetros REST - parámetros agrupados)**
  - Solución anterior a ES6 (no es la ideal)

Funciones usuario - Novedades ES6

- **Funciones con número de parámetros variable (parámetros REST – parámetros agrupados)**
  - Permiten representar un número indefinido de argumentos como un array
  - Los parámetros REST:
    - Se indican anteponiendo 3 puntos
    - Deben ir siempre al final

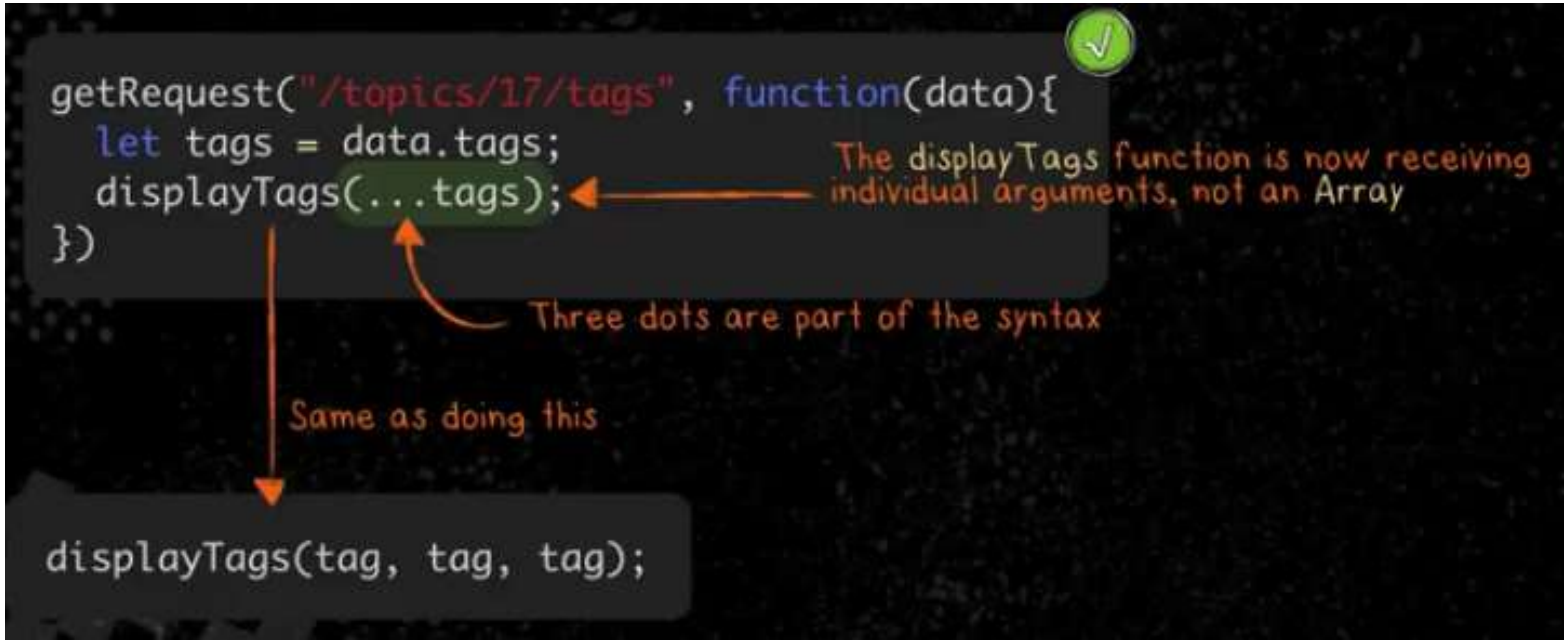
Funciones usuario - Novedades ES6

```
/********
 * Parámetros REST
*********/

function miListaES5 (a, b){
    console.log("ARGUMENTS");
    console.log("a=" + a);
    console.log("b=" + b);
    console.log("arguments=");
    console.log(arguments);
    console.log("Tamaño arguments: " + arguments.length);
}
 miListaES5("pera", "manzana", "uva", "pan", "lima");

 function miListaES6 (a, b, ...otros){
    console.log("REST");
    console.log("a=" + a);
    console.log("b=" + b);
    console.log("Otros= ");
    console.log(otros);
    console.log("tamaño otros= " + otros.length);
    console.log("arguments=");
    console.log(arguments);
    console.log("Tamaño arguments: " + arguments.length);
}
miListaES6 ("pera", "manzana", "uva", "pan", "lima");
```

```
ARGUMENTS
a=pera
b=manzana
arguments=
▼Arguments(5) ["pera", "manzana", "uva", "pan", "lima", call
    0: "pera"
    1: "manzana"
    2: "uva"
    3: "pan"
    4: "lima"
  ▶ callee: ƒ miListaES5(a, b)
    length: 5
  ▶ Symbol(Symbol.iterator): ƒ values()
  ▶ __proto__: Object
Tamaño arguments: 5
REST
a=pera
b=manzana
Otros=
▼(3) ["uva", "pan", "lima"]
    0: "uva"
    1: "pan"
    2: "lima"
    length: 3
  ▶ __proto__: Array(0)
tamaño otros= 3
arguments=
▶Arguments(5) ["pera", "manzana", "uva", "pan", "lima", call
Tamaño arguments: 5
```

Funciones usuario - Novedades ES6

- **Funciones con número de parámetros variable (operador SPREAD – parámetros distribuidos)**
  - Se conoce como operador de propagación
  - Se utiliza para separar el array en elementos individuales



Funciones usuario - Novedades ES6

- **Funciones con número de parámetros variable (operador SPREAD – parámetros distribuidos)**
  - Esta sintaxis extendida permite:
    - **A elementos iterables (array, cadena...):** ser expandidos donde se esperan 0 o más argumentos (para llamadas a función) o elementos (para Array literales)
    - **A un objeto:** Ser expandido en un lugar donde se esperan 0 o más pares de valores clave (para literales tipo Objeto)
  - Sintaxis:
    - **Arrays literales o cadenas:** [...objetoIterable, 'a', 'hola', 7];
    - **Funciones:** myFunction(...objetoIterable)
    - **Literales tipo Object**: let objClone = {...obj}

Funciones usuario - Novedades ES6

- **Funciones con número de parámetros variable (operador SPREAD – parámetros distribuidos)**

```
//ARRAYS
console.log(Math.max(2, 4, 8));  //Devuelve 8
let array = [2, 4, 8];
console.log(Math.max(array));    //Devuelve NaN, ya que Math.max() no trabaja con arrays
console.log(Math.max(...array)); //Devuelve 8

// El operador SPREAD (...) puede ir en cualquier posición.
let array2 = [3, 5, 9];
console.log(...array, 1, ...array2, 6); //Devuelve 2 4 8 1 3 5 9 6
console.log(Math.max(...array, 1, ...array2, 6)) // Devuelve 9

//Concatenar dos arrays
let arrayResultante = [...array, ...array2]; //Devuelve  [2, 4, 8, 3, 5, 9]

// Copiar un array en otro
let arrayCopia = [...array2];//Copia los valores del array [3, 5, 9]
let arrayCopia2 = array2; //Copia la referencia al array   [3, 5, 9]

array2[0]=0;
console.log(arrayCopia);//Devuelve [3, 5, 9]
console.log(arrayCopia2);//Devuelve [0, 5, 9] Paso por referencia
```

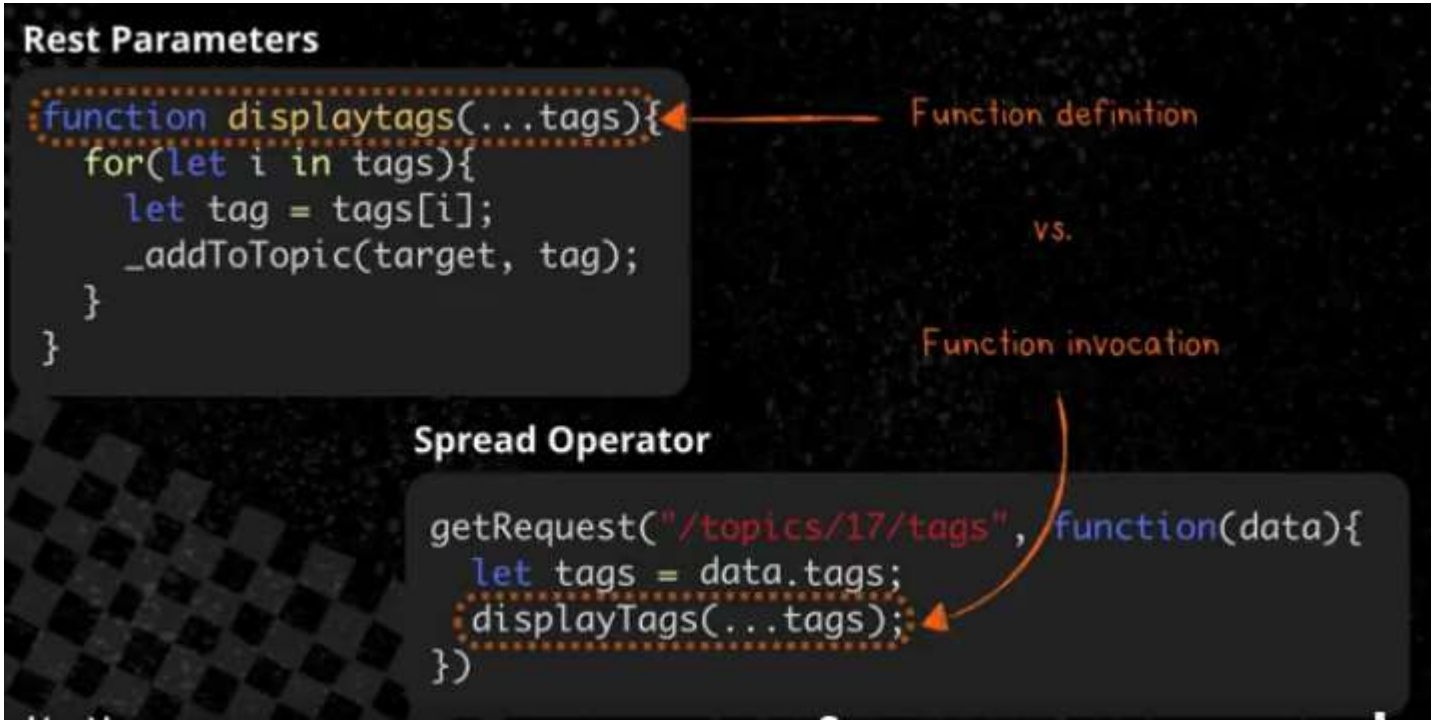
Funciones usuario - Novedades ES6

- **Funciones con número de parámetros variable (operador SPREAD – parámetros distribuidos)**

```
//CADENAS
let saludo = "Hola";
console.log(...saludo); //Devuelve H o l a

//FUNCIONES
function suma (a, b, c){
    return a+b+c;
}
const valores = [1, 2, 3];
console.log(suma(...valores)); //Devuelve 6

//OBJETOS
let persona1 = {nombre: "John", apellido: "Doe"};
let clonJohn = {...persona1}; // Copia por valor de persona1
```

Funciones usuario - Novedades ES6

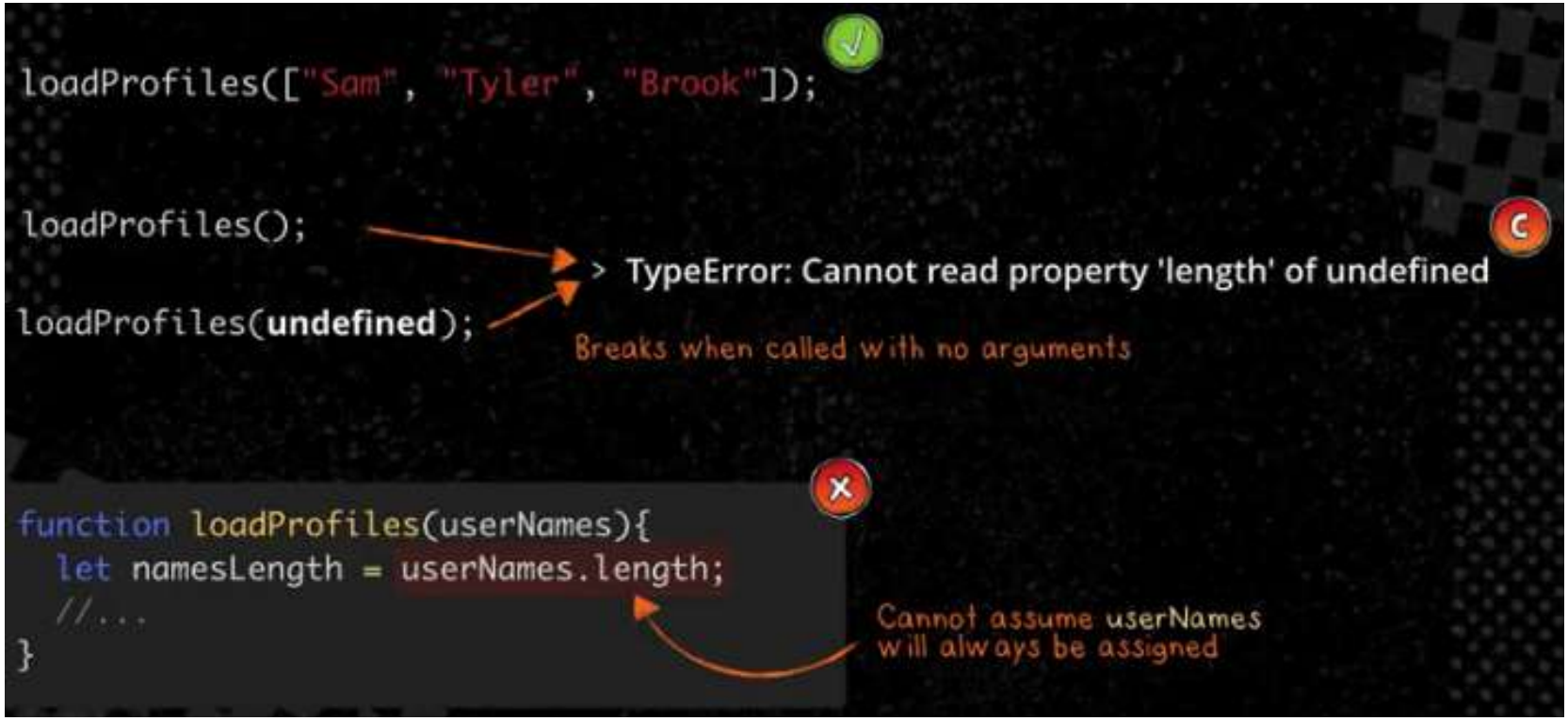
- **Funciones con número de parámetros variable (operador SPREAD vs parámetros REST)**



Funciones del usuario

- Parámetros por defecto:

En ES5 había que tenerlos en cuenta en el código de nuestra función



Funciones del usuario

- Parámetros por defecto:

  En ES5 había que tenerlos en cuenta en el código de nuestra función

```
//ES5
function(valor) {
    valor = valor || "foo";
}
```

```
if (valor){
    valor = valor;
} else {
    valor = "foo";
}
```

```
function multiply(a, b) {
  b = typeof b !== 'undefined' ?  b : 1;

  return a*b;
}

multiply(5); // 5
```

Nota Shorthands:

```
// Longhand
let dbHost;
if (process.env.DB_HOST) {
  dbHost = process.env.DB_HOST;
} else {
  dbHost = 'localhost';
}

// Shorthand
const dbHost = process.env.DB_HOST || 'localhost';
```

```
// If you have to execute a function only if the condition is true:
if (condition) {
    doSomething();
}

// You can use short-circuit:
condition && doSomething();
```

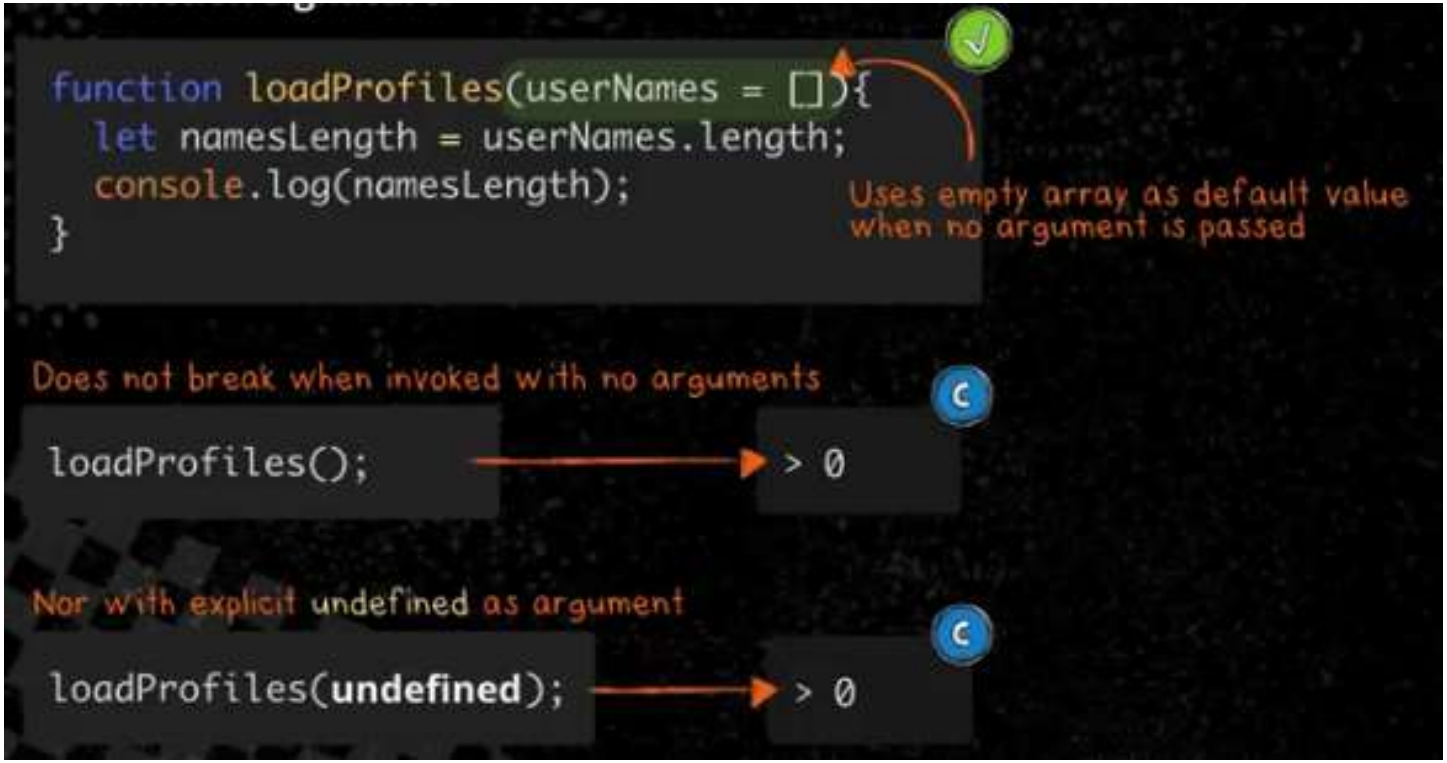
Logical OR assignment *only assigns to a* ***falsy*** *value - great for implementing a "defaut/fallback" logic*

```
user.name ||= 'Anonymous'

// instead of:
if (!user.name) {
    user.name = 'Anonymous'
}
```

Funciones del usuario

- Parámetros por defecto:

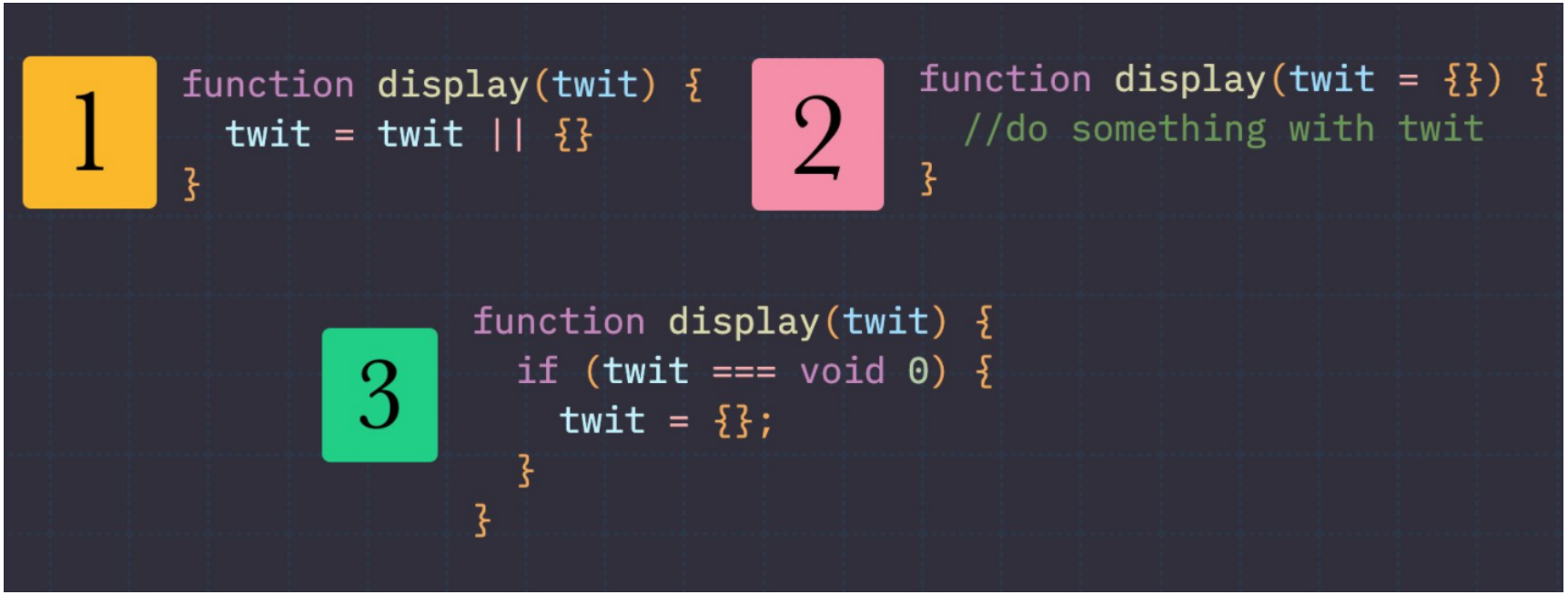
  En ES6 permite establecerlos en la declaración de nuestra función



Funciones del usuario

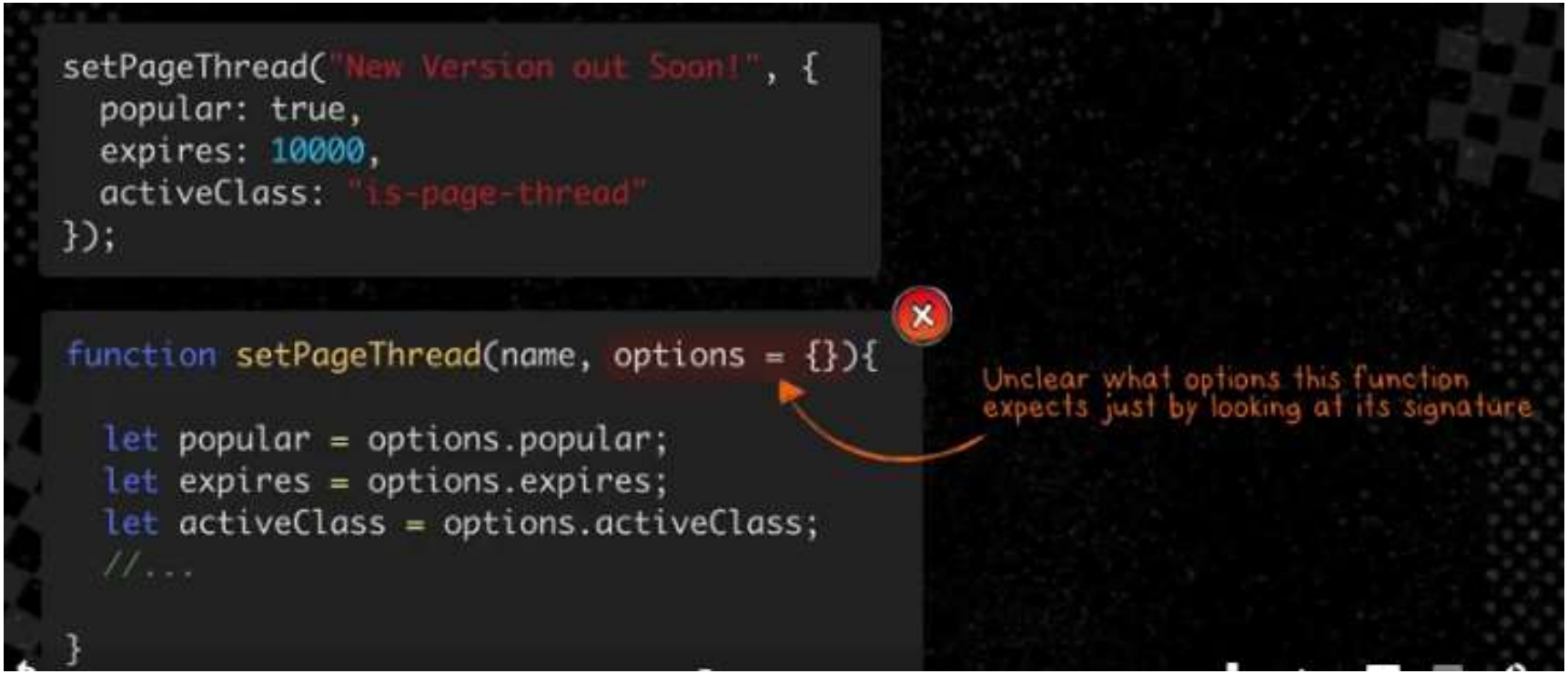
- Parámetros por defecto:

Resumen



Novedades ES6 - funciones

- **Parámetros nombrados**
  - Paso de parámetros a funciones mediante objetos

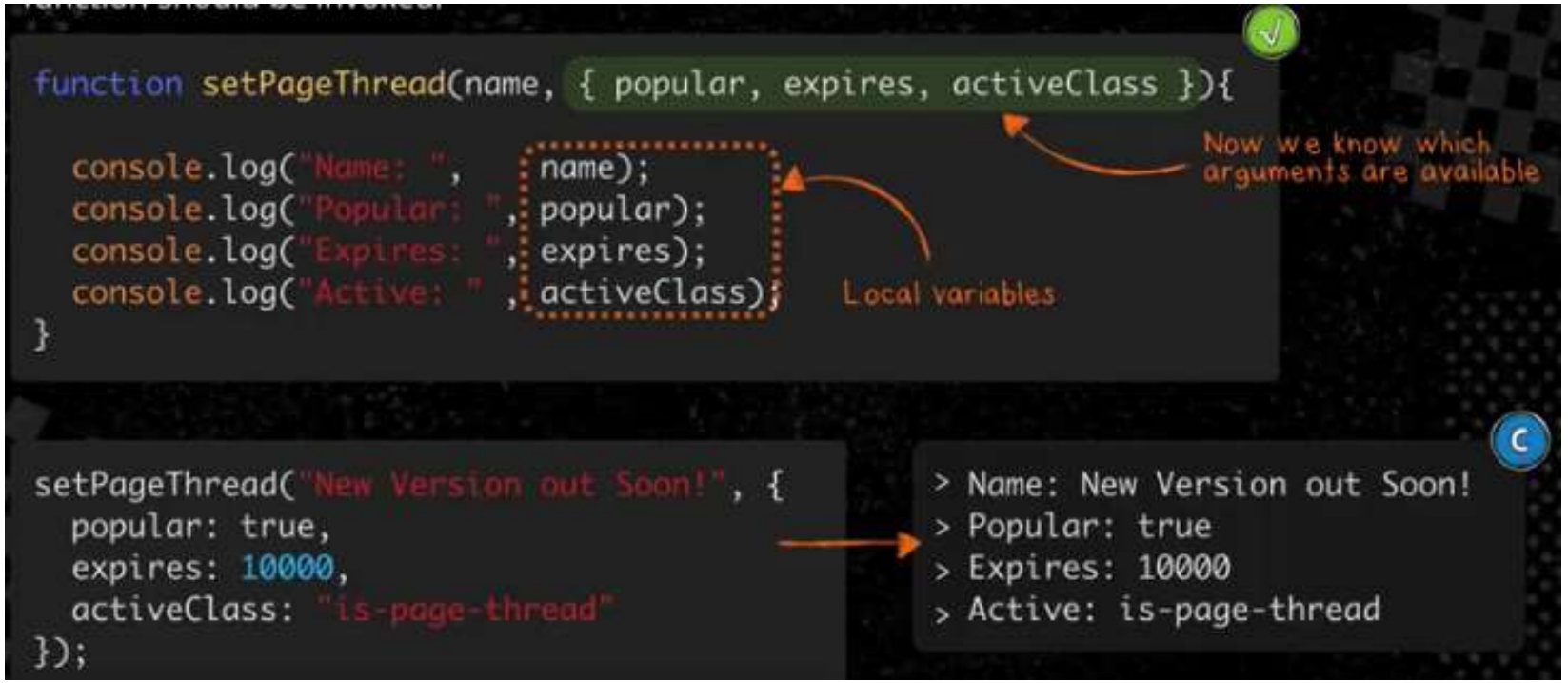


Novedades ES6 - funciones

- **Parámetros nombrados**
  - Paso de parámetros a funciones mediante objetos

Novedades ES6 - funciones

- **Parámetros nombrados**
  - Llamada a la función con todos los parámetros



Novedades ES6 - funciones

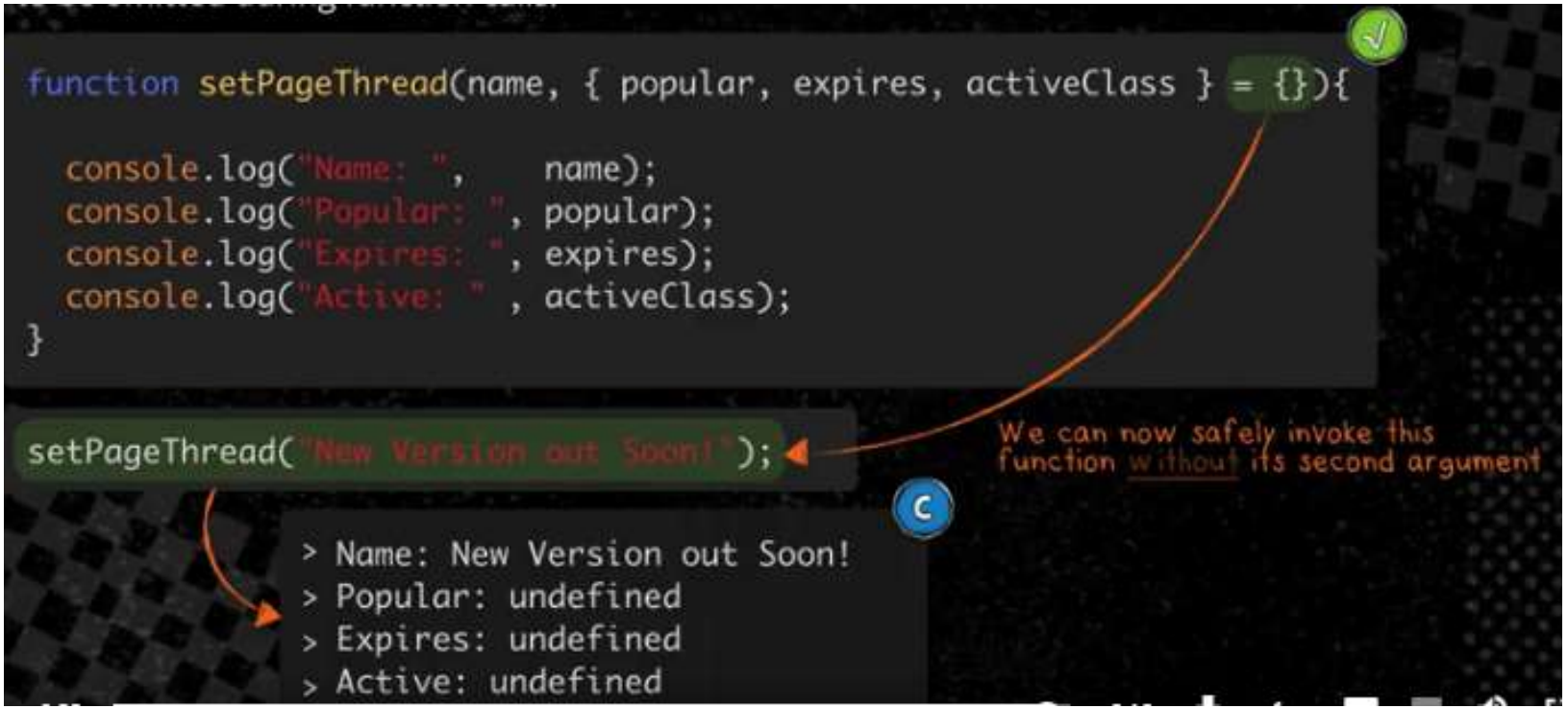
- **Parámetros nombrados**
  - Llamada a la función solo con algunos parámetros

Novedades ES6 - funciones

- **Parámetros nombrados**
  - Llamada a la función solo con algunos parámetros

Novedades ES6 - funciones

- **Parámetros nombrados**
  - Llamada a la función solo con algunos parámetros



Novedades ES6 - funciones

- **Parámetros nombrados**
  - ¿Cuáles de las siguientes llamadas no provocarían un error ?

```
function fetchReplies(topicId, { displayClass, includeAvatar }){
    //...
}
```

```
fetchReplies(12, {
  displayClass: "topic-replies"
});
```

```
let options = {
  displayClass: "topic-replies",
  includeAvatar: true
};
fetchReplies(12, options);
```

```
fetchReplies(12);
```

Novedades ES6 - funciones

- **Parámetros nombrados**
  - ¿Cuáles de las siguientes llamadas no provocarían un error ?

```
function setPageThread(name, {popular, expires, activeClass} = {}){
  // ...
}
```

```
setPageThread("ES2015", {
  popular: true
});
```

```
setPageThread("ES2015", {});
```

```
setPageThread("ES2015");
```

Funciones del usuario

- Funciones utilizadas como datos
  - En JavaScript una función se ejecuta cuando la llamamos usando los paréntesis. Sin paréntesis se obtiene una referencia a la misma

```
var f1, f2;
f1 = miFuncion;
f2 = miFuncion();
```

→ Contiene una referencia a la función *miFuncion*

→ Almacena el valor devuelto por la función *miFuncion*

Funciones del usuario

- Funciones utilizadas como datos
  - El hecho de definir una función con su propio nombre (en este caso "saluda") hace que, dentro de las variables globales se cree una llamada saluda, la cual podemos leer o pasar como parámetro a otra función como parámetro

```
function saluda(){
        console.log("hola");
}
function ejecuta(func){
        func();
}
ejecuta(saluda)
```

Funciones del usuario

- Funciones anónimas:
  - No tienen nombre y son una buena forma de poder reutilizar las funciones
  - Se pueden asociar a una variable
  - Aunque se puede definir sin que sea asignada a ninguna variable.

```
function(quien){
        console.log("hola " + quien);
}
```

  - Sin embargo, hacer esto es totalmente inútil e inservible ya que, al definir una función sin nombre hace que sea imposible ser ejecutada más adelante, pues sin un nombre con el cual acceder a ella es imposible encontrarla.

Funciones del usuario

- Funciones anónimas:
  - Como se ha comentado la verdadera potencia de las funciones anónimas es asociarlas a una variable**:**

```
var sayHello = function(name="world"){
      return `Hello ${name}`;
}
console.log(sayHello());  //Notar que se le ponen paréntesis al final de la variable
console.log(sayHello("Damian"));
```

  - De esta manera, vamos a poder reutilizar la función las veces que se consideren oportuna-

Funciones del usuario

- Funciones autoejecutables
  - Para ello debemos encerrar entre paréntesis la función que hemos creado y añadirle un par de paréntesis al final y esta se ejecutará en el instante.

```
(function() { console.log("hola mundo") }) ()
```

  - Pueden aceptar argumentos

```
( function(quien){
    console.log("hola " + quien);
})("mundo");
```

Funciones del usuario

- Funciones anónimas-autoejecutables
  - Se utilizan para aprovechar las propiedades de ámbito de las variables de JavaScript y el uso de clausuras para escribir **código más limpio que no interfiera con otro código JavaScript** que pudiera haber en la página. De hecho es la técnica que utilizan muchas de las bibliotecas importantes para inicializarse.

```
var v1 = 0;

function miFunc1(){
        v1 = 5;
        alert(v1);
}

function miFunc2(){
        v1 = 10;
        alert(v1);
}
```

```
(function (){
        var v1 = 0;

        function miFunc1(){
                v1 = 5;
                alert(v1);
        };

        function miFunc2(){
                v1 = 10;
                alert(v1);
        };
})();
```

Funciones del usuario

- ¿Una función puede devolver una función anónima?

```
function saludator(quien){
        return function(){ //    se crea la funcion anónima a retornar
                alert("hola " + quien);
        }
}

saludator("mundo")();
```

Funciones del usuario

- Funciones anidadas

```
function saludator(quien){
        function alertasaludo(){
                console.log("hola " +  quien);
        }

        return alertasaludo;
}

var saluda = saludator("mundo");
saluda();
```

```
function saludator(quien){
        function alertasaludo(){
                console.log("hola " +  quien);
        }

        return alertasaludo;
}

saludator("mundo")();
```

Funciones del usuario

- Resumen
  - Las funciones pueden o no llevar nombre, si no llevan nombre estas funciones son anónimas y para ser ejecutadas deben ser asignadas a una variable sino jamás se ejecutarán dentro del código.
  - Una función se puede manipular como si fuera un dato común y corriente, el cual puede ser enviado como parámetro a otra función.
  - Una función que es anónima puede ser auto ejecutable, sólo con ponerle encerrar la función en paréntesis y colocar un nuevo par de paréntesis afuera.
  - Las funciones se pueden anidar dando por resultado una función.

Funciones del usuario

- Funciones declaradas vs funciones expresadas

Las funciones *declaradas* **son evaluadas antes que cualquier otra expresión (hoisting de funciones).**

```
alert( add( 3, 5 ) ); // 8

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

Las funciones *expresadas*, éstas **solo son evaluadas cuando el flujo natural de ejecución las alcanza.**

```
alert( add( 3, 5 ) ); // ErrorType: add is not defined

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

Funciones del usuario

- Funciones declaradas vs funciones expresadas

  Para crear una función en base a un condicional, **nunca** debemos utilizar las **funciones declarativas**

```
if( myVar == true){
  function foo(){ return 'TRUE'; }
}else{
  function foo(){ return 'FALSE'; }
}
```

```
if( myVar == true){
  var foo = function(){ return 'TRUE'; }
}else{
  var foo = function(){ return 'FALSE'; }
}
```

Funciones del usuario

- Resumen
  - La forma correcta de definir una función varía según el comportamiento que esperemos de la misma:
    - Con las **funciones declaradas**, tenemos la seguridad de que siempre estarán disponibles en tiempo de ejecución.
    - Con las **funciones expresadas**, éstas no son evaluadas hasta que el intérprete no alcance su posición en el código, lo cual puede generar errores en arquitecturas muy anidadas.
  - Por otro lado, el hecho de que las funciones declarativas se evalúen antes que las expresiones, pueden producir comportamientos no deseados cuando forman parte de condicionales. Para estos casos, el uso de las expresiones garantiza que éstas formarán parte del flujo general del programa, lo cual puede evitarnos sorpresa en determinados entornos.

Funciones del usuario

- Recursividad
  - Una función puede referirse y llamarse a sí misma. Hay tres formas de una función para referirse a sí mismo:
    - El nombre de la función
    - ~~arguments.callee~~
    - Una variable en el ámbito que se refiere a la función

```
var foo = function bar() {
   // statements go here
};
```

```
1. bar()
2. arguments.callee()
3. foo()
```

```
function loop(x) {
  if (x >= 10) // "x >= 10" is the exit condition (equivalent to "!(x < 10)")
    return;
  // do stuff
  loop(x + 1); // the recursive call
}
loop(0);
```

Funciones del usuario

- Funciones callbacks
  - Cuando se usa una función como parámetro de otra

```
// function expression catSays
var catSays = function(max) {
  var catMessage = "";
  for (var i = 0; i < max; i++) {
    catMessage += "meow ";
  }
  return catMessage;
};

// function declaration helloCat accepting a callback
function helloCat(callbackFunc) {
  return "Hello " + callbackFunc(3);
}

// pass in catSays as a callback function
console.log(helloCat(catSays));
```

Funciones del usuario

- Funciones callbacks inline
  - Cuando no se va a usar más la función

```
// function declaration emotions accepting a callback
function emotions(myString, myFunc) {
    console.log("I am " + myString + ", " + myFunc(2));
}

// Call the emotions() function with two arguments
// Argument 1 - "happy" string
// Argument 2 - an inline function expression
emotions("happy", function(max){
    var str="";
    for (i=0;i<max;i++){
        str+="ha";
    }
    str+="!";
    return str;

})
```

Novedades ES6 - funciones

- **Funciones arrow**
  - Se tratan de funciones sin nombre o anónimas con una sintaxis más compacta
  - No son adecuadas para ser utilizadas como métodos.
  - No pueden ser utilizadas como constructores

```
// Sintaxis básica:
(param1, param2, paramN) => { statements }
(param1, param2, paramN) => expression
   // equivalente a:  => { return expression; }

// Los paréntesis son opcionales cuando solo dispone de un argumento:
singleParam => { statements }
singleParam => expression

// Una función sin argumentos requiere paréntesis:
() => { statements }
```

Si la función solo es un return no necesita llaves ni poner return

```
// ES5
var miFuncion = function(num) {
    return num + num;
}
```

```
// ES6
var miFuncion = (num) => num + num;
```

Novedades ES6 - funciones

- **Funciones arrow**
  - ¿Cómo podríamos escribir esta función utilizado las funciones arrow?

```
let printName = function(value){
  console.log( value );
}
```

```
let printName => (value) {
  console.log( value );
}
```

```
printName(value) => {
  console.log( value );
}
```

```
let printName = (value) => {
  console.log( value );
}
```

Novedades ES6 - funciones

- **Funciones arrow**
  - Contextualiza el valor del objeto this al momento de la declaración.

```
function Person() {
  // El constructor Person() define `this` como el mismo.
  this.age = 0;

  setInterval(function growUp() {
    // En modo no estricto, la funcion growUp() define `this`
    como el objeto global(window), el cual es diferente de el
    `this` definido por el constructor Person().
    alert(this.age++);
  }, 1000);
}

var p = new Person();
```

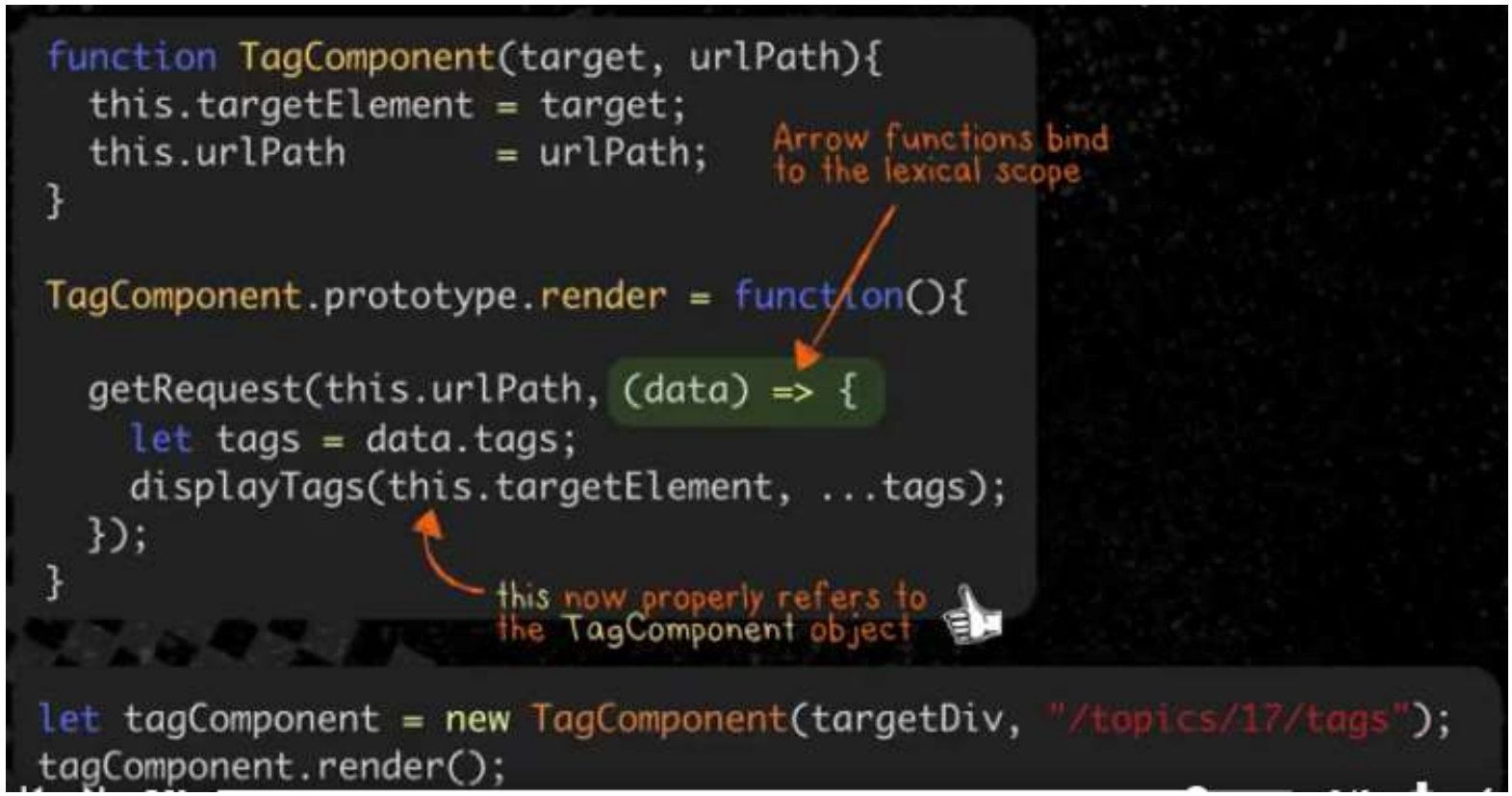
```
function Person(){
  this.age = 0;
   // |this| se refiere apropiadamente al objeto instancia de Person
  setInterval( () => { alert(this.age++) }, 1000);
}

var p = new Person();
```

Novedades ES6 - funciones

- **Funciones arrow**
  - Esto nos facilita con la encapsulación y organización del código

Novedades ES6 - funciones

- **Funciones arrow**
  - Esto nos facilita con la encapsulación y organización del código



Novedades ES8 - funciones

- **Comas al final de parámetros**
  - A partir de ES8, JavaScript no se quejará por esas comas, las ignorará.

```
function foo(a,b,c,) {console.log('Las comas al final ahora no fallan')}

foo ('x','y','z',) //tampoco en la llamada
```