

ÍNDICE

1. FUNCIONES

- a. Funciones predefinidas del lenguaje
- b. Funciones definidas por el usuario

2. ARRAYS

3. OBJETOS DEFINIDOS POR EL USUARIO

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	
<code>encodeURIComponent()</code>	<code>Number()</code>
<code>eval()</code>	<code>String()</code>
<code>isFinite()</code>	<code>parseInt()</code>
<code>isNaN()</code>	<code>parseFloat()</code>

Funciones predefinidas del lenguaje

Este método sustituye a `escape()`

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

```
<script type="text/javascript">
  var input = prompt("Introduce una cadena");
  var inputCodificado = encodeURIComponent(input);
  alert("Cadena codificada: " + inputCodificado);
</script>
```

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

`"my test.asp?name=ståle&car=saab"`

`my%20test.asp?name=st%C3%A5le&car=saab`

Funciones predefinidas del lenguaje

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

```
<script type="text/javascript">
  var input = prompt("Introduce una cadena");
  var inputCodificado = encodeURIComponent(input);
  alert("Cadena codificada: " + inputCodificado);
</script>
```

-- Existe también la función **decodeURIComponent()** que es la función opuesta a **encodeURIComponent()**.

Esta función decodifica los caracteres que estén codificados.

"https://w3schools.com/my test.asp?name=ståle&car=saab"

https%3A%2F%2Fw3schools.com%2Fmy%20test.asp%3Fname%3Dst%C3%A5le%26car%3Dsaab

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.

```
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);
```

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>
```

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.

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 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

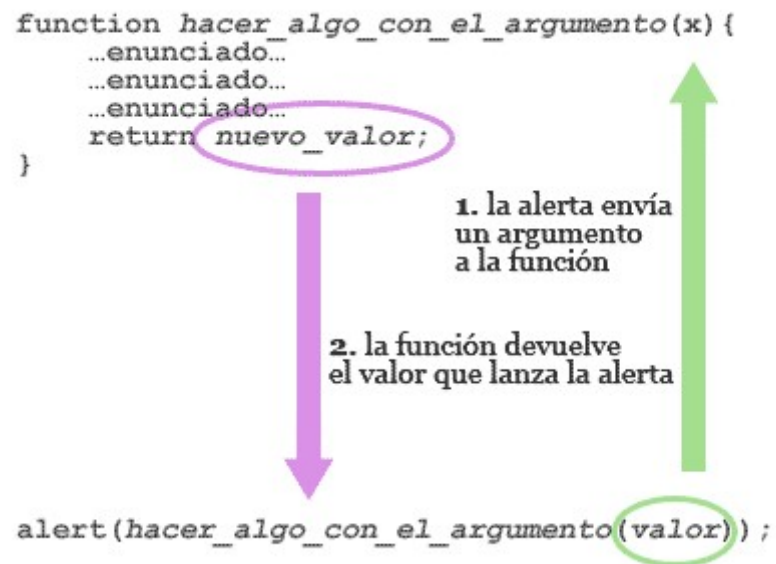
- Definición de funciones – Argumento:
 - Los argumentos se definen dentro del paréntesis situado después del nombre de la función.
 - No todas las funciones requieren argumentos, con lo cual el paréntesis se deja vacío.

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.



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 esto 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:
 - **Arguments** → pseudo-array que contiene todos los argumentos de la función (obligatorios y opcionales)
 - **nombreFuncion.length** → devuelve el número de parámetros obligatorios

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

→ Obligatorios

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

→ Optativos

Funciones del usuario

- Parámetros por defecto:

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

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

  return a*b;
}

multiply(5); // 5
```

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;
```

→ Contiene una referencia a la función *miFuncion*

```
f2 = 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 creadas como expresiones
 - Con este código estamos creando en realidad es una función anónima y asignarle un nombre inmediatamente, por lo cual es equivalente a **"function saluda(quien){ }"**.

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

Funciones del usuario

- Funciones anónimas

- 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 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.**

```
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

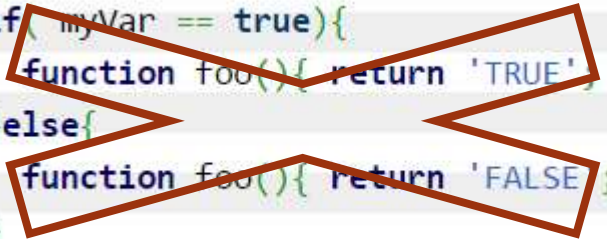
var add = function( x, y ){
    return x + y;
}
```

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

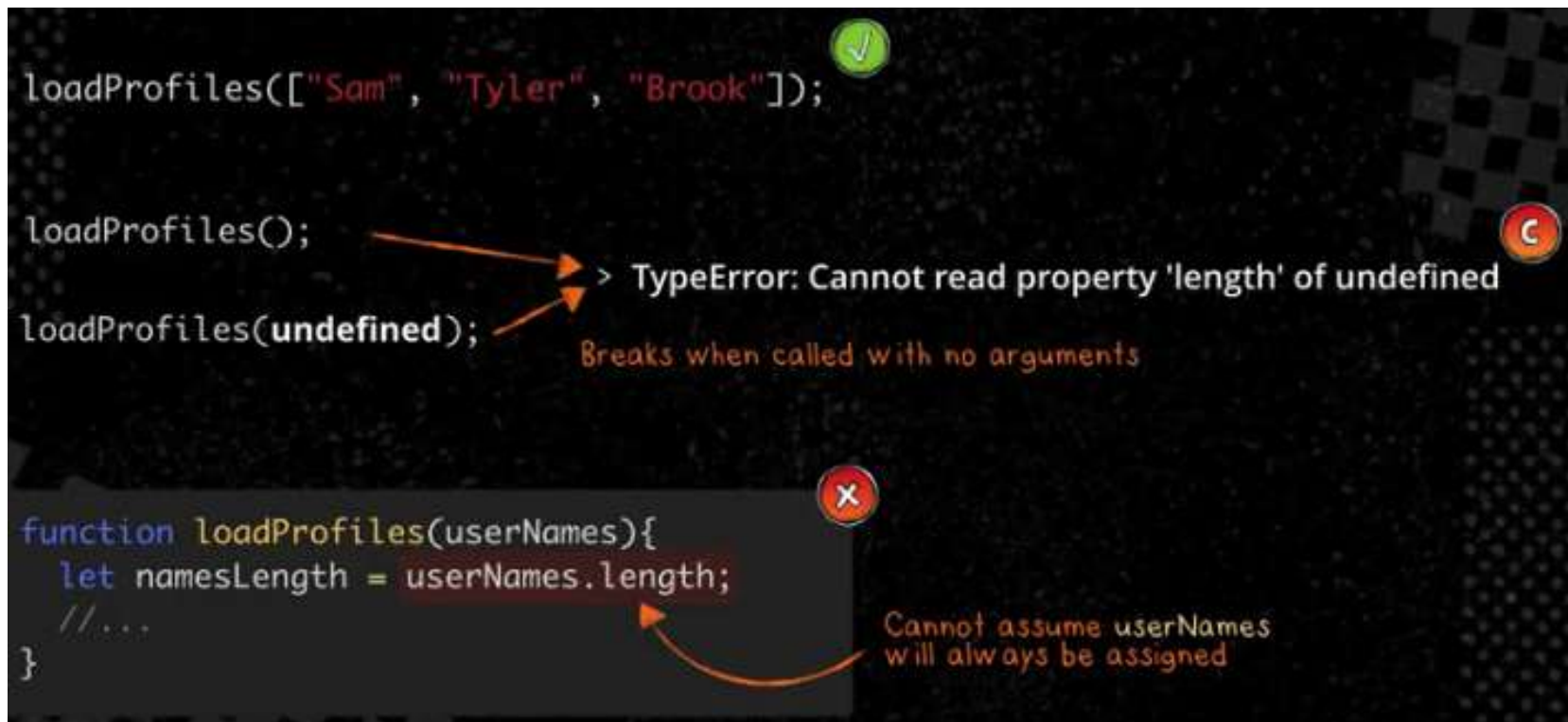
```
var foo = function bar() {  
    // statements go here  
};
```

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);
```

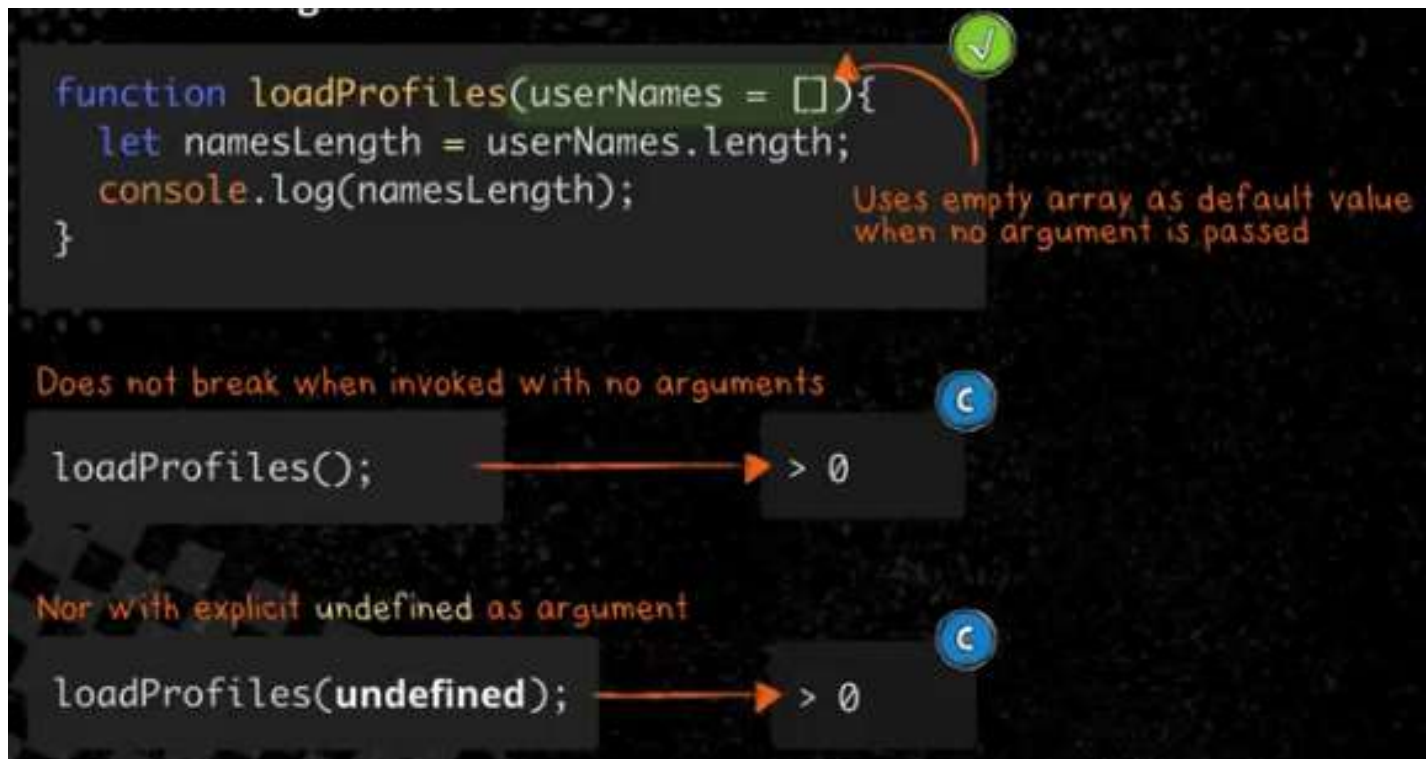

Novedades ES6 - funciones

- Valores por defecto



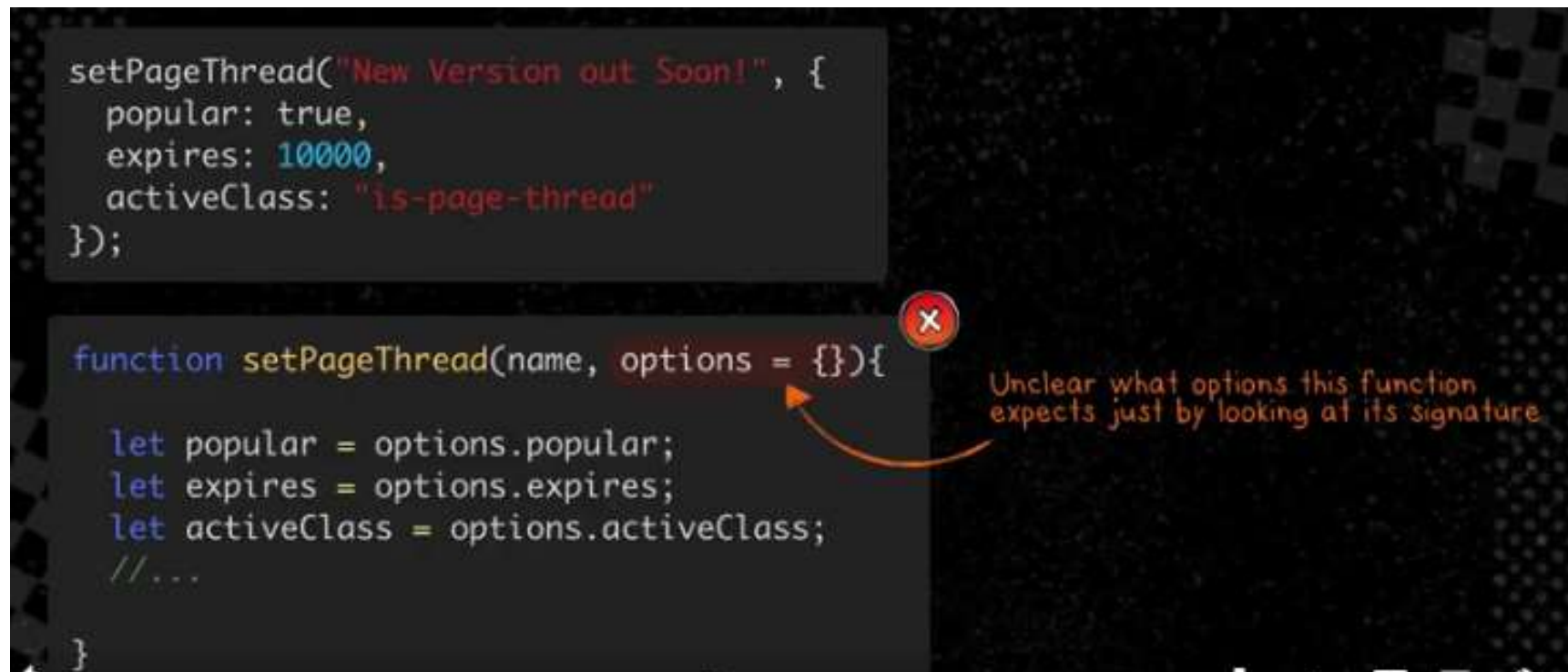
Novedades ES6 - funciones

- Valores por defecto



Novedades ES6 - funciones

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



Novedades ES6 - funciones

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

```
function setPageThread(name, { popular, expires, activeClass }) {  
  console.log("Name: ", name);  
  console.log("Popular: ", popular);  
  console.log("Expires: ", expires);  
  console.log("Active: ", activeClass);  
}
```

Now we know which arguments are available

Local variables

Novedades ES6 - funciones

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

The image shows a code editor with a dark background. At the top, a function is defined with named parameters: `function setPageThread(name, { popular, expires, activeClass }) {`. The parameter object `{ popular, expires, activeClass }` is highlighted with a green box. Below the function definition, four `console.log` statements are shown, each with a corresponding variable from the parameter object: `console.log("Name: ", name);`, `console.log("Popular: ", popular);`, `console.log("Expires: ", expires);`, and `console.log("Active: ", activeClass);`. A dashed orange box encloses these four log statements, with an arrow pointing to it from the text "Local variables". Another arrow points from the text "Now we know which arguments are available" to the parameter object. Below the function definition, a function call is shown: `setPageThread("New Version out Soon!", { popular: true, expires: 10000, activeClass: "is-page-thread" });`. To the right of the function call, a console output window shows the results of the logs: `> Name: New Version out Soon!`, `> Popular: true`, `> Expires: 10000`, and `> Active: is-page-thread`. A blue circle with a 'C' icon is in the bottom right corner of the console output area.

```
function setPageThread(name, { popular, expires, activeClass }) {  
  console.log("Name: ", name);  
  console.log("Popular: ", popular);  
  console.log("Expires: ", expires);  
  console.log("Active: ", activeClass);  
}
```

Now we know which arguments are available

Local variables

```
setPageThread("New Version out Soon!", {  
  popular: true,  
  expires: 10000,  
  activeClass: "is-page-thread"  
});
```

```
> Name: New Version out Soon!  
> Popular: true  
> Expires: 10000  
> Active: is-page-thread
```


Novedades ES6 - funciones

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

The screenshot shows a code editor with a function definition and a function call. The function `setPageThread` has four parameters: `name`, `popular`, `expires`, and `activeClass`. The function body logs each parameter. The function is called with `setPageThread("New Version out Soon!", { popular: true });`. A red dotted arrow points from the `popular` property in the object to the `popular` parameter in the function definition, with the text "popular is the only named argument being passed". The console output shows the function's execution: `> Name: New Version out Soon!`, `> Popular: true`, `> Expires: undefined`, and `> Active: undefined`. The last two lines are enclosed in a red dotted box. A red arrow points from the text "No value assigned to remaining parameters" to the `expires` and `activeClass` parameters in the function definition.

```
function setPageThread(name, { popular, expires, activeClass }){  
  console.log("Name: ", name);  
  console.log("Popular: ", popular);  
  console.log("Expires: ", expires);  
  console.log("Active: ", activeClass);  
}  
  
setPageThread("New Version out Soon!", {  
  popular: true  
});
```

popular is the only named argument being passed

> Name: New Version out Soon!
> Popular: true
> Expires: undefined
> Active: undefined

No value assigned to remaining parameters

Novedades ES6 - funciones

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



```
function setPageThread(name, { popular, expires, activeClass }) {  
  console.log("Name: ", name);  
  console.log("Popular: ", popular);  
  console.log("Expires: ", expires);  
  console.log("Active: ", activeClass);  
}
```

`setPageThread("New Version out Soon!");`

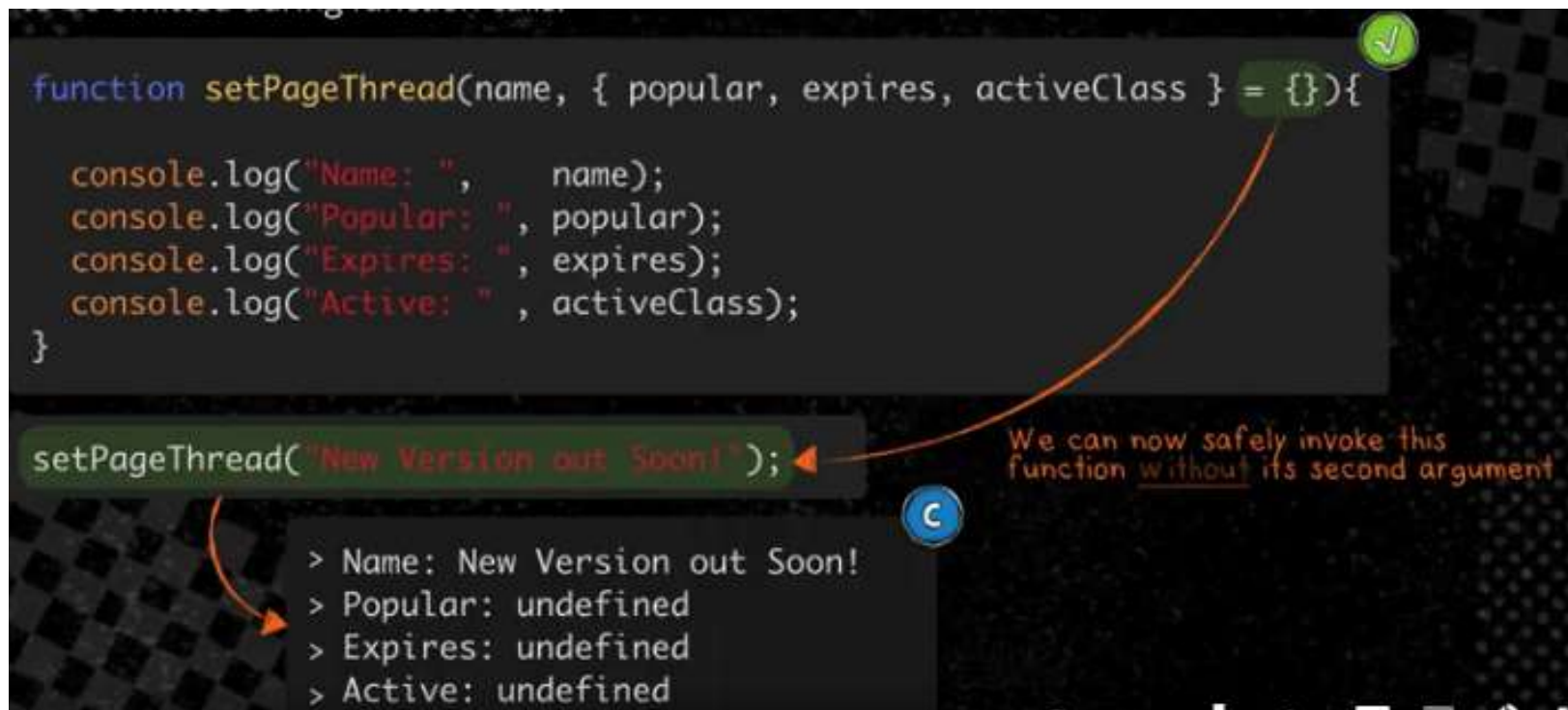
Invoking this function without its second argument breaks our code

> TypeError: Cannot read property 'popular' of undefined

The screenshot shows a code editor with a dark background. The function definition is in a light blue box. The function call is in a light red box. An orange arrow points from the call to the error message. A red 'X' icon is in the top right corner of the code editor area, and a red 'C' icon is in the bottom right corner.

Novedades ES6 - funciones

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



```
function setPageThread(name, { popular, expires, activeClass } = {}){  
  console.log("Name: ", name);  
  console.log("Popular: ", popular);  
  console.log("Expires: ", expires);  
  console.log("Active: ", activeClass);  
}
```

`setPageThread("New Version out Soon!");`

We can now safely invoke this function without its second argument

```
> Name: New Version out Soon!  
> Popular: undefined  
> Expires: undefined  
> Active: undefined
```


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");`

Novedades ES6 - funciones

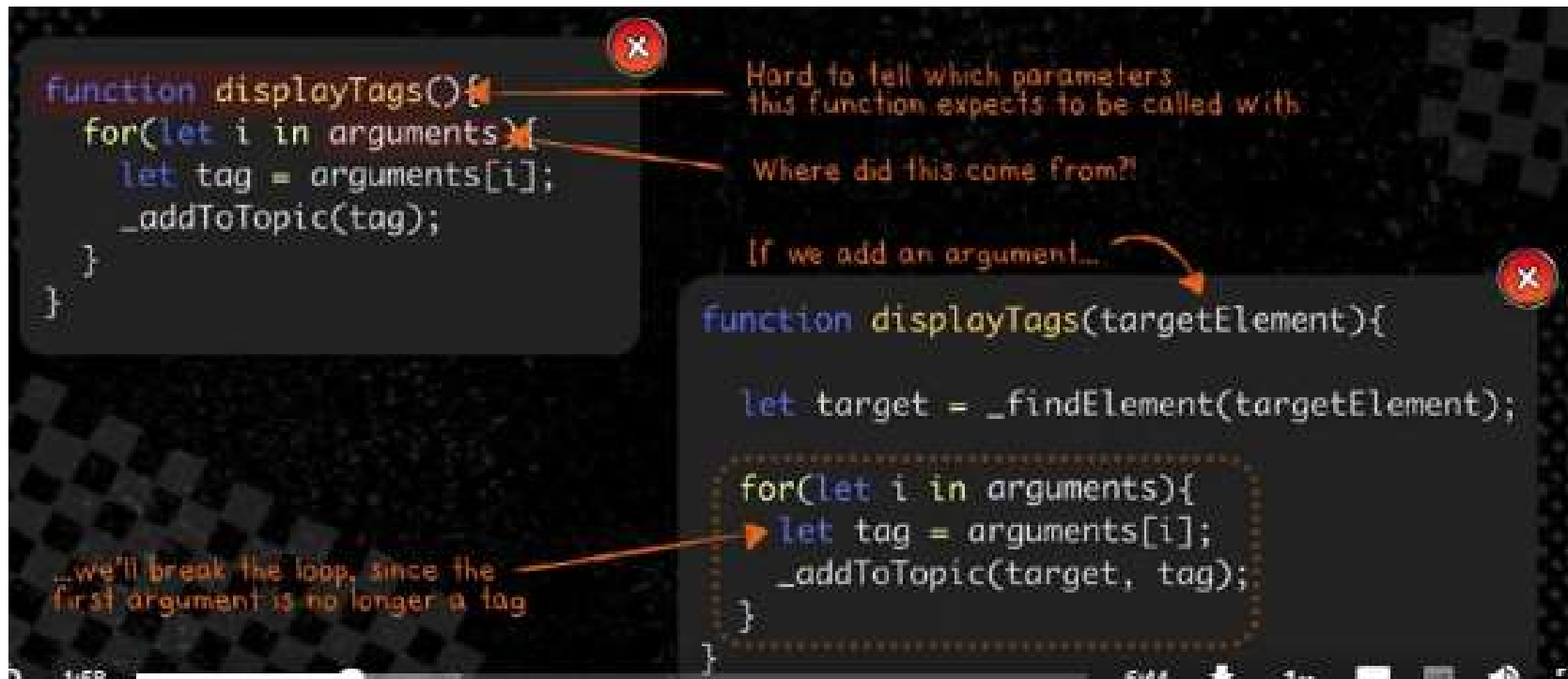
- **Funciones con número de parámetros variable**

```
displayTags("songs");  
or  
displayTags("songs", "lyrics");  
or  
displayTags("songs", "lyrics", "bands");
```

variadic functions can accept any number of arguments

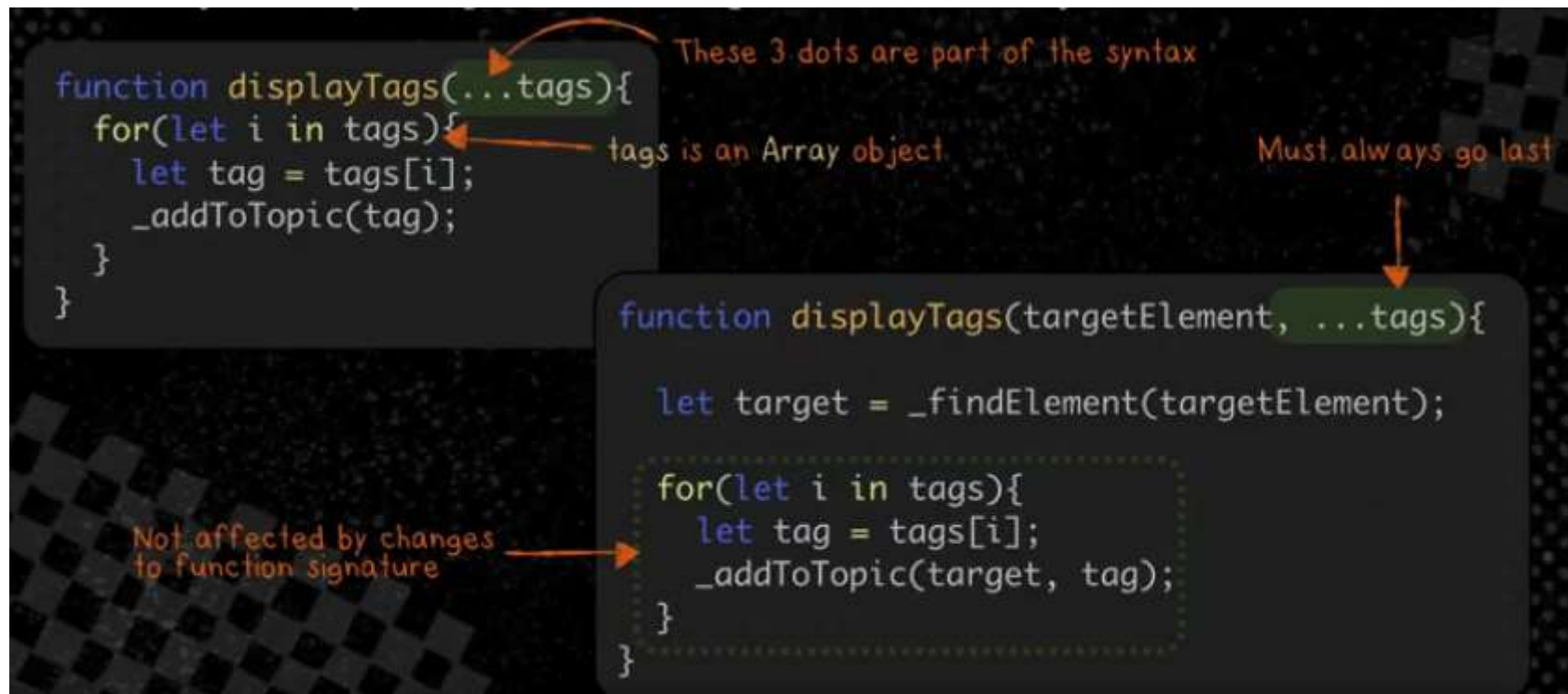
Novedades ES6 - funciones

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



Novedades ES6 - funciones

- Funciones con número de parámetros variable (parámetros rest – parámetros agrupados)



Novedades ES6 - funciones

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



Novedades ES6 - funciones

- Funciones con número de parámetros variable (operador spread vs parámetros rest)

The diagram illustrates the relationship between Rest Parameters and the Spread Operator. It is divided into two main sections: 'Rest Parameters' and 'Spread Operator', separated by a 'vs.' label.

Rest Parameters: This section shows a function definition: `function displaytags(...tags){`. An orange arrow points from the text 'Function definition' to the opening curly brace of the function.

Spread Operator: This section shows a function invocation: `getRequest("/topics/17/tags", function(data){ let tags = data.tags; displayTags(...tags); })`. An orange arrow points from the text 'Function invocation' to the spread operator `...` used to pass the `tags` array as arguments to `displayTags`.

The 'vs.' label is positioned between the two code snippets, indicating a comparison or relationship between the two concepts.

Novedades ES6 - funciones

- **Funciones arrow**

- Tienen una sintaxis más corta comparada con las expresiones de función

```
// 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 }
```

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

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


Novedades ES6 - funciones

- **Funciones arrow**

- ¿Cómo podríamos escribir esta función utilizando 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

```
function TagComponent(target, urlPath){
  this.targetElement = target;
  this.urlPath      = urlPath;
}

TagComponent.prototype.render = function(){
  getRequest(this.urlPath, (data) => {
    let tags = data.tags;
    displayTags(this.targetElement, ...tags);
  });
}

let tagComponent = new TagComponent(targetDiv, "/topics/17/tags");
tagComponent.render();
```

Arrow functions bind to the lexical scope

this now properly refers to the TagComponent object

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
```