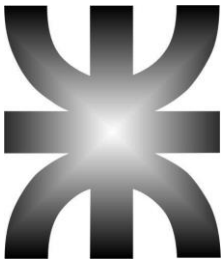

ARROW FUNCTIONS – FUNCIONES FLECHA

UNIDAD #5 - JS

TUP - Laboratorio de Computación II

MATERIAL COMPLEMENTARIO

Funciones Flecha

Una **expresión de función flecha** es una alternativa compacta a una `expresión de función` tradicional, pero es limitada y no se puede utilizar en todas las situaciones.

Diferencias y limitaciones:

- No tiene sus propios enlaces a `this` o `super` y no se debe usar como métodos.
- No tiene `argumentos` o palabras clave `new.target`.
- No apta para los métodos `call`, `apply` y `bind`, que generalmente se basan en establecer un ámbito o alcance
- No se puede utilizar como constructor.
- No se puede utilizar `yield` dentro de su cuerpo.

Comparación de funciones tradicionales con funciones flecha

Observa, paso a paso, la descomposición de una "función tradicional" hasta la "función flecha" más simple: **Nota**: Cada paso a lo largo del camino es una "función flecha" válida

```
// Función tradicional
function (a) {
    return a + 100;
}

// Desglose de la función flecha

// 1. Elimina la palabra "function" y coloca la flecha entre el argumento y el
corchete de apertura.
(a) => {
    return a + 100;
```

```
}

// 2. Quita los corchetes del cuerpo y la palabra "return" — el return está
implícito.
(a) => a + 100;

// 3. Suprime los paréntesis de los argumentos
a => a + 100;
```

Nota: Como se muestra arriba, los { corchetes }, (paréntesis) y "return" son opcionales, pero pueden ser obligatorios.

Por ejemplo, si tienes **varios argumentos** o **ningún argumento**, deberás volver a introducir paréntesis alrededor de los argumentos:

```
// Función tradicional
function (a, b) {
    return a + b + 100;
}

// Función flecha
(a, b) => a + b + 100;

// Función tradicional (sin argumentos)
let a = 4;
let b = 2;
function () {
    return a + b + 100;
}

// Función flecha (sin argumentos)
let a = 4;
let b = 2;
() => a + b + 100;
```

Del mismo modo, si el cuerpo requiere **líneas de procesamiento adicionales**, deberás volver a introducir los corchetes **Más el "return"** (las funciones flecha no adivinan mágicamente qué o cuándo quieres "volver"):

```
// Función tradicional
function (a, b) {
    let chuck = 42;
    return a + b + chuck;
}

// Función flecha
(a, b) => {
    let chuck = 42;
```

```
    return a + b + chuck;
}
```

Y finalmente, en las **funciones con nombre** tratamos las expresiones de flecha como variables

```
// Función tradicional
function bob(a) {
    return a + 100;
}

// Función flecha
let bob = a => a + 100;
```

Sintaxis

Sintaxis básica

Un parámetro. Con una expresión simple no se necesita `return`:

```
param => expression
```

Varios parámetros requieren paréntesis. Con una expresión simple no se necesita `return`:

```
(param1, paramN) => expression
```

Las declaraciones de varias líneas requieren corchetes y `return`:

```
param => {
    let a = 1;
    return a + b;
}
```

Varios parámetros requieren paréntesis. Las declaraciones de varias líneas requieren corchetes y `return`:

```
(param1, paramN) => {
    let a = 1;
    return a + b;
}
```

Sintaxis avanzada

Para devolver una expresión de objeto literal, se requieren paréntesis alrededor de la expresión:

```
params => ({foo: "a"}) // devuelve el objeto {foo: "a"}
```

Los `parámetros rest` son compatibles:

```
(a, b, ...r) => expression
```

Se admiten los `parámetros predeterminados`:

```
(a=400, b=20, c) => expression
```

`Desestructuración` dentro de los parámetros admitidos:

```
([a, b] = [10, 20]) => a + b;  // el resultado es 30
({ a, b } = { a: 10, b: 20 }) => a + b; // resultado es 30
```

Descripción

"`this`" y funciones flecha

Una de las razones por las que se introdujeron las funciones flecha fue para eliminar complejidades del ámbito (`this`) y hacer que la ejecución de funciones sea mucho más intuitiva.

En las **funciones tradicionales** de manera predeterminada `this` está en el ámbito de `window`:

```
window.age = 10; // <-- ¿me notas?
function Person() {
    this.age = 42; // <-- ¿me notas?
    setTimeout(function () {// <-- La función tradicional se está ejecutando
en el ámbito de window
        console.log("this.age", this.age); // genera "10" porque la función se
ejecuta en el ámbito de window
    }, 100);
}

var p = new Person();
```

Las **funciones flecha no** predeterminan `this` al ámbito o alcance de `window`, más bien se ejecutan en el ámbito o alcance en que se crean:

```
window.age = 10; // <-- ¿me notas?
function Person() {
    this.age = 42; // <-- ¿me notas?
    setTimeout(() => {// <-- Función flecha ejecutándose en el ámbito de "p"
(una instancia de Person)
        console.log("this.age", this.age); // genera "42" porque la función se
ejecuta en el ámbito de Person
    }, 100);
}

var p = new Person();
```

En el ejemplo anterior, la función flecha no tiene su propio `this`. Se utiliza el valor `this` del ámbito léxico adjunto; las funciones flecha siguen las reglas normales de búsqueda de variables. Entonces, mientras busca `this` que no está presente en el ámbito actual, una función flecha termina encontrando el `this` de su ámbito adjunto.

Relación con el modo estricto

Dado que `this` proviene del contexto léxico circundante, en el `modo estricto` se ignoran las reglas con respecto a `this`.

```
var f = () => {
    'use strict';
    return this;
};

f() === window; // o el objeto global
```

Todas las demás reglas del `modo estricto` se aplican normalmente.

Funciones flecha utilizadas como métodos

Como se indicó anteriormente, las expresiones de función flecha son más adecuadas para funciones que no son métodos. Observa qué sucede cuando intentas usarlas como métodos:

```
'use strict';

var obj = { // no crea un nuevo ámbito
```

```
    i: 10,
    b: () => console.log(this.i, this),
    c: function () {
        console.log(this.i, this);
    }
}

obj.b(); // imprime indefinido, Window {...} (o el objeto global)
obj.c(); // imprime 10, Object {...}
```

Las funciones flecha no tienen su propio `this`. Otro ejemplo que involucra `Object.defineProperty()`:

```
'use strict';

var obj = {
    a: 10
};

Object.defineProperty(obj, 'b', {
    get: () => {
        console.log(this.a, typeof this.a, this); // indefinida 'undefined'
Window {...} (o el objeto global)
        return this.a + 10; // representa el objeto global 'Window', por lo
tanto 'this.a' devuelve 'undefined'
    }
});
```

`call`, `apply` y `bind`

Los métodos `call`, `apply` y `bind` **NO son adecuados** para las funciones flecha, ya que fueron diseñados para permitir que los métodos se ejecuten dentro de diferentes ámbitos, porque **las funciones flecha establecen "`this`" según el ámbito dentro del cual se define la función flecha.**

Por ejemplo, `call`, `apply` y `bind` funcionan como se esperaba con las funciones tradicionales, porque establecen el ámbito para cada uno de los métodos:

```
// ----------------------
// Ejemplo tradicional
// ----------------------
// Un objeto simplista con su propio "this".
var obj = {
    num: 100
}
```

```
// Establece "num" en window para mostrar cómo NO se usa.
window.num = 2020; // ¡Ay!

// Una función tradicional simple para operar en "this"
var add = function (a, b, c) {
    return this.num + a + b + c;
}

// call
var result = add.call(obj, 1, 2, 3) // establece el ámbito como "obj"
console.log(result) // resultado 106

// apply
const arr = [1, 2, 3]
var result = add.apply(obj, arr) // establece el ámbito como "obj"
console.log(result) // resultado 106

// bind
var result = add.bind(obj) // estable el ámbito como "obj"
console.log(result(1, 2, 3)) // resultado 106
```

Con las funciones flecha, dado que la función `add` esencialmente se crea en el ámbito del `window` (global), asumirá que `this` es window.

```
// ----------------------
// Ejemplo de flecha
// ----------------------

// Un objeto simplista con su propio "this".
var obj = {
    num: 100
}

// Establecer "num" en window para mostrar cómo se recoge.
window.num = 2020; // ¡Ay!

// Función flecha
var add = (a, b, c) => this.num + a + b + c;

// call
console.log(add.call(obj, 1, 2, 3)) // resultado 2026

// apply
const arr = [1, 2, 3]
console.log(add.apply(obj, arr)) // resultado 2026

// bind
```

```
const bound = add.bind(obj)
console.log(bound(1, 2, 3)) // resultado 2026
```

Quizás el mayor beneficio de usar las funciones flecha es con los métodos a nivel del DOM (`setTimeout`, `setInterval`, `addEventListener`) que generalmente requieren algún tipo de cierre, llamada, aplicación o vinculación para garantizar que la función se ejecute en el ámbito adecuado.

Ejemplo tradicional:

```
var obj = {
    count: 10,
    doSomethingLater: function () {
        setTimeout(function () { // la función se ejecuta en el ámbito de
window
            this.count++;
            console.log(this.count);
        }, 300);
    }
}

obj.doSomethingLater(); // la consola imprime "NaN", porque la propiedad
"count" no está en el ámbito de window.
```

Ejemplo de flecha:

```
var obj = {
    count: 10,
    doSomethingLater: function () {  // por supuesto, las funciones flecha no
son adecuadas para métodos
        setTimeout(() => { // dado que la función flecha se creó dentro del
"obj", asume el "this" del objeto
            this.count++;
            console.log(this.count);
        }, 300);
    }
}

obj.doSomethingLater();
```

Sin enlace de `arguments`

Las funciones flecha no tienen su propio objeto `arguments`. Por tanto, en este ejemplo, `arguments` simplemente es una referencia a los argumentos del ámbito adjunto:

```
var arguments = [1, 2, 3];
var arr = () => arguments[0];

arr(); // 1

function foo(n) {
    var f = () => arguments[0] + n; // Los argumentos implícitos de foo son
vinculantes. arguments[0] es n
    return f();
}

foo(3); // 6
```

En la mayoría de los casos, usar `parámetros rest` es una buena alternativa a usar un objeto `arguments`.

```
function foo(n) {
    var f = (...args) => args[0] + n;
    return f(10);
}

foo(1); // 11
```

Uso del operador `new`

Las funciones flecha no se pueden usar como constructores y arrojarán un error cuando se usen con `new`.

```
var Foo = () => {};
var foo = new Foo(); // TypeError: Foo no es un constructor
```

Uso de la propiedad `prototype`

Las funciones flecha no tienen una propiedad `prototype`.

```
var Foo = () => {};
console.log(Foo.prototype); // undefined
```

Uso de la palabra clave `yield`

La palabra clave `yield` no se puede utilizar en el cuerpo de una función flecha (excepto cuando está permitido dentro de las funciones anidadas dentro de ella). Como consecuencia, las funciones flecha no se pueden utilizar como generadores.

Cuerpo de función

Las funciones flecha pueden tener un "cuerpo conciso" o el "cuerpo de bloque" habitual.

En un cuerpo conciso, solo se especifica una expresión, que se convierte en el valor de retorno implícito. En el cuerpo de un bloque, debes utilizar una instrucción `return` explícita.

```
var func = x => x * x;
// sintaxis de cuerpo conciso, "return" implícito

var func = (x, y) => { return x + y; };
// con cuerpo de bloque, se necesita un "return" explícito
```

Devolver objetos literales

Ten en cuenta que devolver objetos literales utilizando la sintaxis de cuerpo conciso `params => {object: literal}` no funcionará como se esperaba.

```
var func = () => { foo: 1 };
// ¡Llamar a func() devuelve undefined!

var func = () => { foo: function() {} };
// SyntaxError: la declaración function requiere un nombre
```

Esto se debe a que el código entre llaves ({}) se procesa como una secuencia de declaraciones (es decir, `foo` se trata como una etiqueta, no como una clave en un objeto literal).

Debes envolver el objeto literal entre paréntesis:

```
var func = () => ({ foo: 1 });
```

Saltos de línea

Una función flecha no puede contener un salto de línea entre sus parámetros y su flecha.

```
var func = (a, b, c)
    => 1;
// SyntaxError: expresión esperada, obtuve '=>'
```

Sin embargo, esto se puede modificar colocando el salto de línea después de la flecha o usando paréntesis/llaves como se ve a continuación para garantizar que el código se mantenga bonito y esponjoso. También puedes poner saltos de línea entre argumentos.

```
var func = (a, b, c) =>
    1;

var func = (a, b, c) => (
    1
);

var func = (a, b, c) => {
    return 1
};

var func = (
    a,
    b,
    c
) => 1;

// no se lanza SyntaxError
```

Orden de procesamiento

Aunque la flecha en una función flecha no es un operador, las funciones flecha tienen reglas de procesamiento especiales que interactúan de manera diferente con `prioridad de operadores` en comparación con las funciones regulares.

```
let callback;

callback = callback || function() {}; // ok
```

```
callback = callback || () => {};
// SyntaxError: argumentos de función flecha no válidos

callback = callback || (() => {});    // bien
```

Ejemplos

Uso básico

```
// Una función flecha vacía devuelve undefinided
let empty = () => { };

(() => 'foobar')();
// Devuelve "foobar"
// (esta es una expresión de función invocada inmediatamente)

var simple = a => a > 15 ? 15 : a;
simple(16); // 15
simple(10); // 10

let max = (a, b) => a > b ? a : b;

// Fácil filtrado de arreglos, mapeo, ...

var arr = [5, 6, 13, 0, 1, 18, 23];

var sum = arr.reduce((a, b) => a + b);
// 66

var even = arr.filter(v => v % 2 == 0);
// [6, 0, 18]

var double = arr.map(v => v * 2);
// [10, 12, 26, 0, 2, 36, 46]

// Cadenas de promesas más concisas
promise.then(a => {
    // ...
}).then(b => {
    // ...
});

// Funciones flecha sin parámetros que son visualmente más fáciles de procesar
setTimeout(() => {
    console.log('sucederá antes');
```

```
    setTimeout(() => {
        // código más profundo
        console.log('Sucederá más tarde');
    }, 1);
}, 1);
```

Referencias

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/Arrow_functions#comparaci%C3%B3n_de_funciones_tradicionales_con_funciones_flecha