Funciones

Curso 2025/26

Paco Segura

Funciones (repaso):

-Función: conjunto de instrucciones que hace algo en concreto.

-Permiten ejecutar el mismo código para distintas entradas sin que tengamos que reescribirlo.

-También nos sirven para esconder detalle (abstracción).

-Ofrecen un servicio que puede ser utilizado.

-Permiten crear código de más alto nivel.

Funciones (repaso):

```
function name(paramOne, paramTwo) {
    // Instrucciones
}
```

Funciones flecha (arrow functions)

Las funciones *flecha* o anónimas, evitan que el código crezca y se “empaste”. Ejemplo:

```
function powerOfTwo(x){
   return x * x;
}
```

Funciones flecha (arrow functions)

Lo que devuelve la función.

Parámetro de entrada.

Funciones flecha (arrow functions)

Si no tiene parámetros de entrada:

```
() => alert("Hello World");
```

Programación Funcional

Paradigma de programación. Uno de los más utilizados. Características:

- Se mantienen datos y funcionalidades separados. Pasamos los datos a las funciones sólo cuando queremos calcular algo.

- Las funciones devuelven nuevos valores que se usan en alguna otra parte del código.

Programación Funcional

Programación Funcional vs Programación Orientada a Objetos:

- PPO nos ayuda a modelar objetos de la vida real. Apropiada cuando la agrupación de propiedades y datos en un objeto tiene sentido lógico, es decir, las propiedades y los métodos '*van juntos*'. En POO datos y funcionalidades se agrupan en objetos '*significativos'*.

- La Programación Funcional mantiene separados datos y funcionalidades.

Programación Funcional

Hay muchos conceptos en programación funcional. Vamos a ver tres de los más importantes:

- Funciones puras y efectos secundarios.
- Funciones de primera clase.
- Funciones de orden superior.

Funciones puras

Una función pura es aquella que devuelve exactamente el mismo resultado siempre que se le den los mismos valores de entrada:

```
function add(paramOne, paramTwo) {
    return(paramOne + paramTwo);
}
```

Funciones puras

Una función pura no produce *‘efectos secundarios’*. Ejemplos:

- Cambiar valores de variables fuera de la propia función, o depender de variables externas.
- Llamar a una API del navegador o a la propia consola.
- Llamar a Math.random(). El valor no se puede repetir de forma fiable.

Funciones de primera clase

Se suele decir que en JavaScript las funciones son *'first class citizens'* porque con ellas se pueden realizar las mismas acciones que con el resto de datos:

- Guardar funciones en variables.
- Pasar funciones como parámetro de entrada.
- Una función puede devolver una llamada a otra función.

Funciones en variables

Las funciones son objetos en JavaScript (tipo *function*). Podemos almacenarlas en una variable: el nombre de la función es el nombre de la variable.

```
const result = function add(paramOne, paraTwo){
    return paramOne + paraTwo;
}

result(2, 3);
```

```
const result = (paramOne, paramTwo) => paramOne + paramTwo;
```

Funciones en variables

Podemos asignarla a otra variable.

```
function add(paramOne, paraTwo){
    return paramOne + paraTwo;
}

let result = add(2, 3);
console.log(result); // Muestra un 5 por pantalla
```

Funciones de orden superior

- Una **función es de orden superior** si cumple al menos una de estas dos condiciones:
  - Toma al menos una función como parámetro.
  - Devuelve una función como resultado.

- Permiten ocultar detalle (abstracción).

JavaScript ofrece varias funciones de orden superior integradas en el lenguaje. Veamos algunos ejemplos:

Funciones de orden superior

- .addEventListener(): se utiliza para agregar un evento a un elemento HTML. Toma como parámetros de entrada el tipo de evento y una función para asociar lógica a ese evento:

```
element.addEventListener('click', function(){
    // Lógica a implementar
});
```

Funciones de orden superior

- setTimeout(): toma como parámetros de entrada una función y un tiempo en milisegundos y programa la ejecución de la función después del tiempo especificado.

  Sintaxis:

```
setTimeout(functionRef, delay);
```

setTimeout

- Ejemplo: mostrar una alerta pasados 500ms.

```
setTimeout(function(){
    alert('Hello');
}, 500);
```

setTimeout

- El *delay* introducido **no impide** que el resto del código siga ejecutándose:

```
setTimeout(() => alert('First'), 500);

alert('Second');
```

Funciones de orden superior

- setInterval(): toma como parámetros de entrada una función y un intervalo de tiempo en milisegundos y programa la ejecución periódica de la función cada intervalo de tiempo especificado.

```
setInterval(functionRef, interval);
```

setInterval

```
let counter = 0;
setInterval(() => {
    console.log(counter);
    counter++;
}, 500);
```

setInterval

- Para cancelar el intervalo, guardamos la función en una variable y le aplicamos **clearInterval** a esa variable:

```
let counter = 0;
let interval = setInterval(() => {
    console.log(counter);
    if(counter === 10){
        clearInterval(interval);
    }
    counter++;
}, 500);
```

Funciones callback

- Nomenclatura utilizada para denominar a la función pasada como parámetro a otra función.

```
const powerOfTwo = (add) => {
    return add * add;
}

const add = (paramOne, paramTwo, callback) => {
    let sum = paramOne + paramTwo;
    return callback(sum);
}

console.log(add(2, 3, powerOfTwo));
```

Al hacer la llamada a la función ‘add’, le pasamos la función ‘powerOfTwo’ como argumento. De este modo ‘add’ hará la suma de 2 y 3 y luego llamará a la función que le hemos pasado por parámetro para que la ejecute con el resultado de esta suma. Como se ve en el ejemplo, también puedo devolver una llamada a una función desde otra función.

Funciones de orden superior sobre arrays

- Colección de funciones de orden superior predefinidas en que operan sobre arrays.
- Vamos a ver algunas de ellas:
  - map
  - forEach
  - filter
  - some
  - every
  - reduce
  - ...

map

- Similar a un ‘for’.
- Aplica una función sobre un ‘array’, ejecutando dicha función sobre cada posición del array.
- La función la define el usuario y puede tomar como parámetros de entrada el valor y el índice de cada posición del array.
- Devuelve un nuevo array.
- Veamos algunos ejemplos:

map

- Aplicamos una función flecha a un array para que muestre por consola el valor que tiene guardado en cada posición del array:

```
[0, 1, 2, 3].map(value => console.log(value));
```

- El parámetro de entrada 'value' es el valor guardado en cada posición del array. Puede tomar cualquier nombre:

```
[0, 1, 2, 3].map(element => console.log(element));
[0, 1, 2, 3].map(item => console.log(item));
```

map

- En la función podemos implementar cualquier lógica a aplicar sobre el array:

```
[0, 1, 2, 3].map(value => console.log(value + 1));
```

map

- También puede tomar como parámetro de entrada el índice del array:

```
[0, 1, 2, 3].map((value, index) => {
    console.log(`Valor: ${value + 1} / Índice: ${index}`);
});
```

map

- También puede no tomar parámetros de entrada:

```
[0, 1, 2, 3].map(() => console.log('Hello'));
```

map

- Devuelve un nuevo array:

```
let newArray = [0, 1, 2, 3].map(value => value + 1);

console.log(newArray);
```

map

- Si trabajamos con arrays más grandes, conviene guardarlos en variables:

```
let myArray = [0, 1, 2, 3, 5, 6, 7, 8, 9, 10];

myArray.map((value) => console.log(value));
```

map

- Podemos aplicarle directamente una función:

```
let myArray = [0, 1, 2, 3, 5, 6, 7, 8, 9, 10];

function squared(num){
    return console.log(num * num);
}

myArray.map(squared);
```

map

- Podemos trabajar con arrays de objetos:

```
let myArray = [
    { name: 'John', surname: 'Doe'},
    { name: 'Peter', surname: 'Collins'},
    { name: 'Susan', surname: 'Black'}
];

myArray.map(element => {
    console.log(`Nombre: ${element.name} / Apellido: ${element.surname}`);
});
```

forEach

- Similar a 'map', también aplica una función a cada posición de un array pero no devuelve un nuevo array.

```
let myArray = [
    { name: 'John', surname: 'Doe'},
    { name: 'Peter', surname: 'Collins'},
    { name: 'Susan', surname: 'Black'},
];

myArray.forEach(element => {
    console.log(`Nombre: ${element.name} / Apellido: ${element.surname}`);
});
```

forEach

- No devuelve un nuevo array.

```
let myArray = [0, 1, 2, 3, 4];

let myNewArray = myArray.forEach(element => element = element + 1);

console.log(myArray);
console.log(myNewArray); // Undefined!
```

filter

- Realiza un ‘filtrado’.
- Para ello aplica una función que define una condición a comprobar sobre el ‘array’ de entrada.
- Devuelve un nuevo array que contiene sólo aquellos elementos del array de entrada que cumplen con la condición determinada por la función.

filter

- Ejemplo: obtener los números pares de un array.

```
let myArray = [0, 1, 2, 3, 4];

let myNewArray = myArray.filter(element => element % 2 === 0);

console.log(myNewArray);
```

some

- Realiza una ‘comprobación’.
- Para ello aplica una función que define una condición a comprobar sobre el ‘array’ de entrada.
- Devuelve un booleano. ‘True’ si hay al menos un elemento del array que cumple con la condición determinada por la función. ‘False’ si ningún elemento del array cumple con dicha condición.

some

- Ejemplo: comprobar si hay **algún** número par.

```
let myArray = [0, 1, 2, 3, 4];

let myNewArray = myArray.some(element => element % 2 === 0);

console.log(myNewArray);
```

every

- Realiza una ‘comprobación’.
- Para ello aplica una función que define una condición a comprobar sobre el ‘array’ de entrada.
- Devuelve un booleano. ‘True’ si hay todos los elementos del array cumplen con la condición determinada por la función. ‘False’ si ningún elemento del array cumple con dicha condición.

every

- Ejemplo: comprobar si **todos** los números son pares.

```
let myArray = [0, 1, 2, 3, 4];

let myNewArray = myArray.every(element => element % 2 === 0);

console.log(myNewArray);
```

reduce

- Realiza una ‘reducción’.
- Para ello aplica una función de entrada que toma como valores un acumulador, el valor actual y un valor inicial.
- Va actualizando el valor del acumulador a partir de los valores guardados en cada elemento del array.
- Devuelve como resultado final el valor del acumulador.

reduce

Ejemplos:

```
let myArray = [4, 5, 7, 12];

let myNewArray = myArray.reduce((accumulator, currentValue) => accumulator +
currentValue, 0);
console.log(myNewArray);
myNewArray = myArray.reduce((accumulator, currentValue) => accumulator -
currentValue, 0);
console.log(myNewArray);
myNewArray = myArray.reduce((accumulator, currentValue) => accumulator +
currentValue, 100);
console.log(myNewArray);
```

find y sus variantes

- Realiza una ‘búsqueda.
- Para ello aplica una función de entrada que define una condición.
- Busca en el array que elementos cumplen con la condición.
- Devuelve el primer valor que cumple con la condición o en caso de no encontrar ninguno, undefined.

find y sus variantes

- Ejemplo: buscar el primer elemento par del array.

```
[1, 2, 3, 4, 5].find(element => element % 2 === 0);
```

- Variantes de find(): **findLast** –último elemento-, **findIndex** –primer índice que cumple la condición-.

Operador Spread

Nos permite definir funciones que pueden recibir un número variable de parámetros de entrada.

```
const add = (...params) => {
    let result = 0;
    for (let value of params){
        result += value;
    }

    return result;
}

console.log(add(2, 3));
console.log(add(2, 3, 4));
console.log(add(2, 3, 4, 5, 6));
```

Operador Spread

```
const add = (...params) => {
    let result = 0;
    for (let value of params){
        result += value;
    }

    return result;
}

console.log(add(2, 3));
console.log(add(2, 3, 4));
console.log(add(2, 3, 4, 5, 6));
```

En la función tratamos a los parámetros variables como si vinieran en un objeto iterable cuyo nombre es el definido en la cabecera (args).

Operador Spread

```
const add = (...params) => {
    let result = 0;
    for (let value of params){
        result += value;
    }

    return result;
}

console.log(add(2, 3));
console.log(add(2, 3, 4));
console.log(add(2, 3, 4, 5, 6));
```

La función admite un número variable de parámetros. El usuario no tiene que poner los parámetros en una colección.