JavaScript – *function*

En JavaScript, hay dos formas principales de crear funciones: expresión de funciones y declaración de funciones. Aunque ambas permiten definir funciones en tu código, difieren en cómo se crean y cómo se comportan en términos de elevación (hoisting) y disponibilidad en el ámbito.

Declaración de Funciones:

La declaración de funciones es una forma más tradicional de crear funciones en JavaScript. Se utiliza la palabra clave `function` seguida por un nombre de función y un conjunto de paréntesis que pueden contener parámetros. Aquí hay un ejemplo:

```
function sumar(){
    console.log(2+2);
}
```

Las declaraciones de funciones son elevadas (hoisted) en el ámbito en el que se encuentran, lo que significa que se pueden llamar antes de la definición real en el código. Por ejemplo, puedes llamar a la función `sum` antes de su declaración y funcionará sin problemas.

Expresión de Funciones:

La expresión de funciones implica asignar una función anónima (o con nombre) a una variable. Esto se hace utilizando la sintaxis `var/let/const nombreFuncion = function(parámetros) { ... }` o mediante la notación de funciones flecha (`() => {...}`). Aquí hay un ejemplo de una expresión de función con notación de función flecha:

```
const sumar2 = function(){
    console.log(2+2);
}
```

Las expresiones de funciones no son elevadas en el ámbito en el que se encuentran. Deben declararse antes de usarse. Intentar llamar a una expresión de función antes de su definición resultará en un error.

Diferencias Clave:

1. Elevación (Hoisting): Las declaraciones de funciones son **elevadas en su ámbito**, lo que significa que pueden llamarse antes de su definición en el código. Las expresiones de funciones no son elevadas y deben declararse antes de ser utilizadas.

2. Asignación a Variables: Las expresiones de funciones se pueden asignar a variables, lo que permite una mayor flexibilidad y composición en tu código.

Diferencia entre funciones y métodos

En programación, los conceptos de función y método están estrechamente relacionados, pero se utilizan en contextos ligeramente diferentes según el paradigma de programación que estés utilizando, como la programación orientada a objetos (POO) o la programación funcional.

Función:

Una función es un bloque de código que realiza una tarea específica y puede aceptar entradas (argumentos) y devolver una salida (valor de retorno). Las funciones son componentes fundamentales de la programación y se utilizan para modularizar el código, hacerlo más legible y reutilizable. En la programación funcional, las funciones son tratadas como ciudadanos de primera clase y pueden ser pasadas como argumentos a otras funciones y devueltas como valores.

Método:

Un método es una función que está asociada con un objeto o una clase en el contexto de la programación orientada a objetos (POO). Los métodos son acciones o comportamientos específicos que un objeto puede realizar. Están definidos en la definición de una clase y se invocan en una instancia (objeto) de esa clase. Los métodos pueden acceder a los datos internos (propiedades) del objeto en el que se invocan.

Diferencias clave:

1. **Contexto**: Una función **puede existir de manera independiente** y no está necesariamente vinculada a un objeto o clase en particular. Un método, por otro lado, está vinculado a un objeto o una clase y opera en el contexto de ese objeto.

2. **Invocación**: Una función **se invoca directamente usando su nombre**, mientras que un método se invoca en el contexto de un objeto utilizando la notación de punto (objeto.metodo()).

3. **Acceso a datos**: Los métodos pueden acceder a las propiedades y datos internos de un objeto en el que se invocan. Las funciones pueden tener acceso a los datos pasados como argumentos, pero no necesariamente tienen acceso a los datos internos de otros objetos.

4. **Programación orientada a objetos**:

- El concepto de método es fundamental en la POO y se utiliza para encapsular comportamientos y acciones específicas de los objetos.
- Las funciones, en contraste, se utilizan en varios paradigmas de programación, incluida la programación funcional.

Una función es un bloque de código independiente que realiza una tarea, mientras que un método es una función asociada con un objeto o una clase en el contexto de la programación orientada a objetos. Ambos conceptos son esenciales en la programación y se utilizan en diferentes contextos para lograr diferentes objetivos.

Funciones

```
// Definición de una función que suma dos números
function sum(a, b) {
    return a + b;
}

// Invocación de la función
const result = sum(3, 5); // Resultado: 8
```

Métodos

```
// Definición de una clase con un método
class Rectangle {
    constructor(width, height) {
        this.width = width;
        this.height = height;
    }
    // Método para calcular el área del rectángulo
    calculateArea() {
        return this.width * this.height;
    }
}

// Creación de una instancia de la clase Rectangle
const rect = new Rectangle(4, 6);

// Invocación del método en la instancia
const area = rect.calculateArea(); // Resultado: 24
```

Arrow function

Las funciones de flecha (arrow functions) en JavaScript tienen varias ventajas que las hacen populares y útiles en comparación con las funciones tradicionales definidas con la sintaxis convencional de `function`. Aquí hay algunas de las principales ventajas de las funciones de flecha:

1. **Sintaxis más concisa**: Las funciones de flecha permiten una sintaxis más compacta y legible, lo que resulta en un código más limpio y fácil de entender.

2. **Ligadura del valor `this` simplificada**: En las funciones tradicionales, el valor de `this` puede variar dependiendo de cómo se llama a la función. Las funciones de flecha mantienen el valor de `this` del contexto en el que se definen, lo que puede evitar confusiones y errores.

3. **No tienen su propio `this`, `arguments`, `super` o `new.target`**: Esto las hace especialmente útiles en situaciones donde es deseable que el valor de `this` sea el mismo que en el ámbito circundante.

4. **No necesitan la palabra clave `return` para retornar valores**: Si la función de flecha contiene solo una expresión, esa expresión se devuelve implícitamente como resultado de la función.

5. **Ideal para funciones anónimas y devoluciones de llamada**: Las funciones de flecha son ideales para funciones cortas y funciones pasadas como argumentos a otras funciones, como en métodos como `map`, `filter`, `reduce`, entre otros.

6. **No tienen asociado el objeto `arguments`**: Esto puede simplificar el manejo de argumentos en funciones, ya que las funciones de flecha no tienen un objeto `arguments` propio.

7. **Buena para contextos de programación funcional**: Debido a su sintaxis y comportamiento simplificado, las funciones de flecha son populares en programación funcional y en la creación de flujos de datos.

8. **Mejor rendimiento en algunas situaciones**: Debido a su sintaxis más simple y la falta de propiedades dinámicas, en algunos casos las funciones de flecha pueden ser más eficientes en términos de rendimiento que las funciones tradicionales.

Aunque las funciones de flecha tienen muchas ventajas, también es importante tener en cuenta sus limitaciones. Por ejemplo, debido a la falta de su propio `this`, no son adecuadas para ser utilizadas como métodos en objetos, y pueden no ser la elección correcta en situaciones donde se requiera el uso de `this` dinámico o el acceso a propiedades del objeto `arguments`.

Ejemplo Objeto Constructor

```
function producto (descripcion, precio)
{
    this.descripcion = descripcion;
    this.precio = precio;

    this.mostrarProducto = function () {
        console.log(`El producto: ${this.descripcion} tiene un costo de:
${this.precio}`)
    }

    this.total = () => {
        return this.precio;
    }
}

const prod = new producto('Lenovo 110', 1000);
console.log(prod.total())
console.log(prod.mostrarProducto())
```

Ejemplo Objeto Literal

```
const persona = {
    nombre:``,
    saludar: function(){
        console.log(`Hola ${this.nombre}`);
    },
    despedirse: () => {
        console.log(`Chau amigos...`)
    }
}
```

En las clases

```
class Rectangulo{
    constructor(ancho, alto) {
        this.alto = alto;
        this.ancho = ancho;
    }
    calcularArea(){
        return this.alto*this.ancho;
    }
    get altoRect(){
        return this.alto;
    }

    mostrar = () => {
        console.log('Este es un rectangulo de ' + this.alto);
    }
}

const r = new Rectangulo(10,10);
console.log(r.calcularArea());
console.log(r.altoRect)
r.mostrar()
```