Algoritmos: 8

Programación funcional

La **programación funcional** es un paradigma de programación que se basa en el uso de funciones matemáticas para construir software. Se enfoca en el manejo de funciones puras, la inmutabilidad y la evaluación declarativa de datos, promoviendo un estilo de codificación más predecible y menos propenso a errores.

Ventajas de la programación funcional

- **Código más predecible**: Las funciones puras garantizan que el mismo input siempre produzca el mismo output, lo que facilita el razonamiento y depuración del código.

- **Facilidad para probar y depurar:** Al evitar efectos secundarios y mantener funciones puras, las pruebas unitarias son más simples de implementar y más confiables.

- **Reutilización de código:** Las funciones son modulares y pueden combinarse para resolver problemas más complejos, promoviendo la reutilización.

- **Mejor manejo de concurrencia:** La inmutabilidad reduce los riesgos de condiciones de carrera (race conditions) y conflictos en entornos concurrentes o paralelos.

Ventajas de la programación funcional

- **Menos errores:** La eliminación de efectos secundarios y la adopción de inmutabilidad ayudan a prevenir errores difíciles de detectar, como la mutación inesperada de datos.

- **Código más declarativo:** En lugar de describir cómo se realiza una tarea (imperativo), el enfoque funcional describe qué se desea lograr, lo que puede hacer el código más claro y legible.

- **Facilidad para razonamiento matemático:** Basado en principios matemáticos, como funciones puras y composición, lo que simplifica el análisis lógico del programa.

- **Optimización automática:** Muchas implementaciones de lenguajes funcionales soportan técnicas como la **evaluación perezosa (lazy evaluation)**, que permite optimizar el rendimiento ejecutando cálculos solo cuando son necesarios.

Funciones de orden superior

Las **funciones de orden superior** son funciones que cumplen al menos una de las siguientes características:

1. **Reciben otras funciones como parámetros.**
2. **Devuelven una función como resultado.**

Este concepto es fundamental en paradigmas como la programación funcional y permite que el código sea más modular, reutilizable y expresivo.

Función que recibe otra función como parámetro

Una función de orden superior puede tomar otra función como argumento y aplicarla.

```
const aplicarOperacion = (operacion, a, b) => operacion(a, b);

const sumar = (x, y) => x + y;

console.log(aplicarOperacion(sumar, 5, 3)); // 8
```

aplicarOperacion es una función de orden superior porque toma operacion (otra función) como argumento.

sumar es una función que se pasa a aplicarOperacion.

Función que recibe otra función como parámetro

Así mismo como lo hicimos con la suma podemos hacerlo con las demás operaciones básicas, se veria algo así:

```
const ejecutarOperacion = (operacion, a, b) => operacion(a, b);

const sumar = (x, y) => x + y;
const restar = (x, y) => x - y;
const multiplicar = (x, y) => x * y;
const dividir = (x, y) => x / y;


console.log(ejecutarOperacion(sumar, 5, 3));       // 8
console.log(ejecutarOperacion(restar, 5, 3));     // 2
console.log(ejecutarOperacion(multiplicar, 5, 3)); // 15
console.log(ejecutarOperacion(dividir, 6, 3));     // 2
```

Función que devuelve otra función

Una función de orden superior también puede generar y devolver una nueva función.

```
const crearMultiplicador = multiplicador => numero => numero * multiplicador;


const multiplicarPor2 = crearMultiplicador(2);
const multiplicarPor5 = crearMultiplicador(5);

console.log(multiplicarPor2(4)); // 8
console.log(multiplicarPor5(3)); // 15
```

crearMultiplicador devuelve una nueva función que multiplica un número por el valor proporcionado al llamarla.

Función que devuelve otra función

En este ejemplo creamos una función de orden superior la cual realizara un saludo dependiendo del idioma que se especifique.

```
const crearSaludo = idioma => {
    if (idioma === 'es') {
        return nombre => `Hola, ${nombre}!`;
    } else if (idioma === 'en') {
        return nombre => `Hello, ${nombre}!`;
    } else {
        return nombre => `Salut, ${nombre}!`;
    }
};


const saludoEnEspanol = crearSaludo('es');
const saludoEnIngles = crearSaludo('en');


console.log(saludoEnEspanol('Carlos')); // Hola, Carlos!
console.log(saludoEnIngles('John'));   // Hello, John!
```

¿Qué es un Callback?

Un **callback** es una función que se pasa como argumento a otra función y que se ejecuta después de que esta última haya terminado su tarea. Los callbacks permiten personalizar y extender el comportamiento de una función al permitir que otra función específica sea ejecutada en un momento determinado, como respuesta a un evento o tras la finalización de un proceso.

Ejemplo básico de un callback:

```
function procesarUsuario(nombre, callback) {
    console.log(`Procesando al usuario: ${nombre}`);
    callback(nombre);
}

function saludo(nombre) {
    console.log(`¡Hola, ${nombre}!`);
}

procesarUsuario('Carlos', saludo);
```

En este ejemplo realizamos la función procesarUsuario la cual recibe como parámetros un nombre y un callback, el callback se ejecutará una vez realizado el bloque de código de la función procesarUsuario

Ventajas de los callbacks

- **Permiten manejar tareas asíncronas:** Los callbacks son fundamentales para manejar operaciones asíncronas, como solicitudes a APIs, acceso a bases de datos o temporizadores, sin bloquear la ejecución del programa.
- **Modularidad y flexibilidad:** Separan la lógica principal de una función de las acciones que se ejecutarán al completar la tarea, lo que mejora la claridad y reutilización del código.
- **Fomentan la reutilización de código:** Puedes usar la misma función con diferentes callbacks para personalizar su comportamiento sin duplicar código.

Algoritmos: 8

VankVersity