¿Qué es debounce?

Debounce es una técnica que retrasa la ejecución de una función hasta que haya pasado un tiempo determinado sin que se vuelva a llamar. Es como un "temporizador que se reinicia" cada vez que intentas ejecutar la función.

Análisis paso a paso

1. Definición de la función

```
const debounce = (func: (e: ChangeEvent<HTMLInputElement>) => void,
delay: number) =>
```

- Recibe dos parámetros:
  - `func`: La función que queremos "debouncear" (retrasar)
  - `delay`: El tiempo en milisegundos que debe esperar antes de ejecutar la función

2. Variable para almacenar el timeout

```
let timeoutId: ReturnType<typeof setTimeout>
```

- Declara una variable que almacenará el ID del timeout
- `ReturnType<typeof setTimeout>` typeof setTimeout es el tipo de la función `setTimeout`, mientras que `ReturnType` obtiene el tipo de retorno de esa función, que es un número en JavaScript. Esto asegura que `timeoutId` sea del tipo correcto para almacenar el ID del timeout. Esta variable persiste entre llamadas gracias al closure.

3. Función que se retorna

```
return (e: ChangeEvent<HTMLInputElement>) => {
```

- Retorna una nueva función que será la que realmente uses
- Esta función recibe el evento del input como parámetro

4. Cancelar timeout anterior

```
clearTimeout(timeoutId)
```

- **Clave del debounce**: Cancela cualquier timeout pendiente

- Esto significa que si la función se llama antes de que termine el delay, el timeout anterior se cancela

5. Crear nuevo timeout

```
timeoutId = setTimeout(() => {
    func(e)
}, delay)
```

- Crea un nuevo timeout que ejecutará la función original después del delay
- Guarda el ID del timeout en la variable para poder cancelarlo después

Cómo funciona en la práctica?

1. **Usuario escribe "H"** → Se programa ejecutar `handleSearch` en 500ms
2. **Usuario escribe "e" (antes de 500ms)** → Se cancela el timeout anterior y se programa uno nuevo
3. **Usuario escribe "l" (antes de 500ms)** → Se cancela el timeout anterior y se programa uno nuevo
4. **Usuario escribe "l" (antes de 500ms)** → Se cancela el timeout anterior y se programa uno nuevo
5. **Usuario escribe "o" (antes de 500ms)** → Se cancela el timeout anterior y se programa uno nuevo
6. **Usuario para de escribir** → Después de 500ms se ejecuta `handleSearch` con "Hello"

Beneficios

- **Performance**: Evita hacer muchas llamadas innecesarias (ej: a APIs)
- **UX**: Reduce el "ruido" de ejecuciones mientras el usuario está escribiendo
- **Recursos**: Ahorra ancho de banda y procesamiento

Concepto clave: `Closure`

Gracias al **`closure`** de JavaScript, la variable `timeoutId` queda "capturada" en el scope de la función `debounce`, permitiendo que persista entre múltiples llamadas a la función retornada.