Aplicando lo aprendido 3.

Nombre y apellido: Maia Belén Bulacio.

Ejercicio 4:

1) Explica en un texto, con ejemplos y fundamentación qué características de la OOP utilizaste para resolver los programas de los Ejercicios 2 y 3. Si hay alguna que no utilizaste o no implementaste, indica cuál y por qué crees que no fue necesario.

RESPUESTA:

En ambos ejercicios implemente algunas de las características de la OOP como clases, objetos, encapsulamiento, abstracción, modularidad, herencia e interfaces.

- ✓ En la lista de tareas (JavaScript) solo use clases para definir la estructura de la tarea, por lo cual no fue necesario hacer uso de herencia. Además, cada vez que se agrega una tarea se crea un objeto donde cada una de ellas tenga su propio estado y comportamiento:

```
const nuevaTarea = new Tarea(titulo, descripcion, dificultad, vencimiento);
```

- ✓ Encapsulamiento: Cada método encapsula comportamientos específicos, por ejemplo, en mostrarDetalles se imprimen todos los datos guardados (en caso de haber guardado algo) de una tarea:

```
gestionTarea.prototype.mostrarDetalles = function (t) {
    console.log("\n --- Detalles de la tarea ---");
    console.log("Título: ", t.titulo || "sin datos");
    console.log("Descripción: ", t.descripcion || "sin datos");
    console.log("Estado: ", t.estado);
    console.log("Dificultad: ", t.mostrarDificultad());
    console.log("Vencimiento: ", t.vencimiento || "sin datos");
    console.log("Creación: ", t.creacion);
    console.log("Última edición: ", t.ultimaEdicion || "sin datos");
    console.log("-------------------------");
};
```

- ✓ Abstracción: se abstrae en dos archivos la estructura de tareas y el gestor donde van a estar todos los métodos para administrar las tareas (agregar, editar, buscar, ver)
- ✓ Modularidad: dividí el programa en varios archivos (tarea.js, main.js y gestionTarea.js) para mayor claridad en el código y para que pueda ser reutilizado después.

- ✓ En la calculadora (TypeScript) use clases para cada operación (suma, resta, producto y división) y una interfaz que define el método ejecutar(a, b)
- ✓ Polimorfismo: la clase de cada operación va a implementar la interfaz Operación, teniendo que usar en cada método ejecutar() con su respectivo comportamiento. Ej:

```ts
export class Suma implements Operacion {
    ejecutar(a: number, b: number): number {
        return a + b;
    }
}
```

- ✓ Encapsulamiento: usa encapsulamiento al declarar private el atributo operación:

```ts
export class Calculadora {
    private operacion: Operacion;

    constructor(operacion: Operacion) {
        this.operacion = operacion;
    }

    setOperacion(operacion: Operacion) {
        this.operacion = operacion;
    }

    calcular(a:number, b:number):  number {
        return this.operacion.ejecutar(a,b);
    }
}
```

- ✓ Modularidad: el código está separado en distintos archivos (operacion.ts, operaciones.ts, main.ts) para mejorar la organización.