Trabajo Práctico 3

Alumno: Tula E. Fernando

Carrera: Ing. de Sistemas de Información

Ejercicio 1

Considera el lenguaje JavaScript acotado al paradigma de programación orientada a objetos basado en prototipos y analízalo en términos de los cuatro componentes de un paradigma mencionados por Kuhn.

1. Generalización simbólica: ¿Cuáles son las reglas escritas del lenguaje?

Son las **reglas formales y sintácticas** que definen cómo funciona el lenguaje.
En JavaScript, acotado al paradigma orientado a objetos prototípico, estas son:

- Todo objeto puede actuar como **prototipo** de otro.
- Los objetos se crean a partir de **funciones constructoras** y se vinculan mediante la propiedad interna `[[Prototype]]` o `__proto__`.
- Los métodos y propiedades compartidas se definen en el **objeto prototipo**.
- La **herencia** se realiza por **delegación de prototipos**, no por clases (en su forma original).
- La palabra clave `this` referencia al objeto que ejecuta el método.
- Se pueden crear objetos mediante **literales** (`{}`), **Object.create()**, o **funciones constructoras**.

2. Creencias de los profesionales: ¿Qué características particulares del lenguaje se cree que sean "mejores" que en otros lenguajes?

Son las **ideas compartidas por la comunidad** sobre lo que hace bueno al lenguaje:

- **Flexibilidad extrema:** todo es objeto, y se puede modificar o extender dinámicamente.
- **Modelo simple de herencia:** más liviano y natural que la herencia clásica (no requiere jerarquías rígidas).
- **Reutilización por composición:** se valora combinar objetos en lugar de heredar.
- **Tipado dinámico:** permite programar más rápido y adaptar estructuras en tiempo de ejecución.
- **Funciones de primera clase:** los métodos y constructores pueden manipularse como cualquier otro valor.

Estas creencias defienden que JavaScript es **más expresivo y adaptable** que lenguajes con orientación a clases estática como Java o C++.

3. ¿Qué pensamiento o estilo de programación consideraron mejor los creadores?

Reflejan la **filosofía o estilo de pensamiento** promovido por sus creadores (Brendan Eich y la comunidad inicial de Netscape):

- **Simplicidad y dinamismo sobre rigidez.**
- **Programación incremental:** crear y modificar objetos en tiempo real.
- **Tolerancia a errores y flexibilidad sintáctica.**
- **Enfoque en la interoperabilidad:** integración fácil con otros lenguajes y entornos (navegadores, servidores, APIs).
- **Delegación y composición** en lugar de herencia estricta.

El valor central es que el **comportamiento de los objetos puede definirse y modificarse dinámicamente**, reflejando un pensamiento práctico y experimental, más cercano al modelo biológico de "prototipos que evolucionan".

4. Ejemplares: ¿Qué clase de problemas pueden resolverse más fácilmente en el lenguaje?

Son los **problemas modelo** que se resuelven especialmente bien en este paradigma:

- **Interfaces y componentes reutilizables** (widgets, módulos, prototipos visuales).
- **Simulación de objetos reales** donde las instancias comparten estructura pero pueden modificarse individualmente.
- **Aplicaciones web dinámicas**, donde los objetos representan elementos interactivos del DOM.
- **Sistemas extensibles** (plugins, frameworks) donde nuevos objetos se agregan sin redefinir clases.
- **Prototipado rápido**: ideal para construir versiones preliminares o experimentales.

Ejercicio 4

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.

Para realizar el ejercicio 3 implementé la programación orientada a prototipos. Considero que esta forma es más natural para javacript, ya que el lenguaje está basado en objetos y prototipos desde su diseño original: todo en JavaScript es un objeto que puede heredar de otro mediante la delegación prototípica.

En mi código utilicé funciones constructoras para crear los objetos base y el operador new para instanciar nuevas tareas. También definí los métodos en el objeto prototipo, de modo que todas las instancias compartan las mismas funciones sin duplicar código.

```
export function constructorTarea(
  this: interfazTarea,
  titulo : string,
```

```
  descripcion : string,
  dificultad: Dificultad,
  vencimiento: Vencimiento,
  ultimaModificacion: Date,
  estado: Estado)
{
  Object.defineProperty(this, "id", {
    value: uuidv4(),
    writable: false, // impide modificarlo luego
    enumerable: true,
  });

  Object.defineProperty(this, "fechaCreacion", {
    value: new Date(),
    writable: false,
    enumerable: true,
  });
  this.titulo = titulo;
  this.descripcion = descripcion;
  this.dificultad = dificultad;
  this.vencimiento = vencimiento;
  this.ultimaModificacion = ultimaModificacion;
  this.estado = estado;
}
```

Ejemplo de instanciación del objeto tarea:

```
const tarea = new (constructorTarea as any) ( titulo, descripcion, dificultad, vencimiento, ultimaModificacion, estado);
```

Ejemplo de definicion de metodo del prototipo:

```
constructorListaTareas.prototype.agregarTarea = function (): void {
    //Las funciones auxiliares deben ser métodos del prototipo e invocarse como this.metodo().
    const titulo = this.agregarTitulo();
    const descripcion = this.agregarDescripcion();
    const estado: Estado = "pendiente"; // Estado inicial siempre es "pendiente"
    const dificultad= this.agregarDificultad();
    const vencimiento= this.agregarVencimiento();
    const ultimaModificacion = new Date() ;

      const tarea = new (constructorTarea as any) ( titulo, descripcion, dificultad, vencimiento, ultimaModificacion, estado);
    this.listaTareas.push(tarea);

    console.log(`Tarea "${tarea.titulo}" agregada.`);
    this.ordenarTareas(this.listaTareas);
}
```

Apliqué conceptos de la orientación a objetos, como la encapsulación (mediante getters y setters)

```
constructorTarea.prototype.getTitulo = function() : string {
  return this.titulo;
}

constructorTarea.prototype.setTitulo = function(nuevoTitulo : string ) : void{
  this.titulo = nuevoTitulo;
  this.ultimaModificacion = new Date();
}
```

y la modularidad (separando los constructores Tarea y ListaTareas en diferentes archivos). No implementé herencia ni polimorfismo porque el problema no requería jerarquías de clases ni distintos tipos de tareas: era suficiente con instancias del mismo tipo que comparten comportamiento común.

Hice uso de ES Modules para importar y exportar tipos y funciones entre archivos.

```
import { constructorListaTareas } from './ListaTareas';
import { constructorMenuPrincipal } from './MenuPrincipal';
//creo un arreglo de objetos de tipo Tarea
const objetoListaTareas = new (constructorListaTareas as any)();
const menuPrincipal = new (constructorMenuPrincipal as any) ();

menuPrincipal.menu(objetoListaTareas);
```

Me pareció super interesante el uso de las interfaces como plantilla de los objetos, ya que permite a su vez definir tipos que sirven para indicar a las funciones que tipo de parámetros recibir y que tipo de datos produce como salida, a su vez que le da uniformidad al codigo, de modo que se debe respetar un contrato, tal cual establece la interfaz.

```
export interface interfazTarea {
    readonly id: string; //el id es un uuid
    titulo: string;
    descripcion: string;
    dificultad: Dificultad;
    vencimiento: Vencimiento;
    readonly fechaCreacion: Date;
    ultimaModificacion: Date;
    estado: Estado;
    getId(): string;
    getFechaCreacion(): Date;
    getTitulo(): string;
    setTitulo(nuevoTitulo : string): void;
    getDescripcion(): string;
    setDescripcion(nuevaDescripcion : string) : void;
    getVencimiento() : Vencimiento;
    setVencimiento(nuevoVencimiento : Vencimiento) : void;
    getDificultad() : Dificultad;
    setDificultad(nuevaDificultad : Dificultad) : void;
    getEstado(): Estado;
    setEstado(nuevoEstado : Estado) : void;
}
```

Considero que la implementación de esta forma de programar, la programación orientada a prototipos es más dinámica, ya que consiste en la clonación de objetos existentes, va de la mano con javascript, ya que está diseñado en base a objetos. A su vez no me gusta que se pueda aplicar el encapsulamiento de forma camuflada, ya que todos los objetos son públicos y no existen los modificadores de acceso: public, private, static.

Es importante agregar que la programación orientada a prototipos tiene gran flexibilidad, ya que adquiere todas las características de javascript, pero también es más difícil de usar, es fácil perderse en lo que realmente hacen las instrucciones, por ser tan permisivo el lenguaje.

Particularmente me gusta mucho más la programación orientada a objetos, donde se definen clases con métodos y atributos como plantillas para la creación de objetos, es mucho más estructurada y controlable al basarse en los cuatro pilares de la POO: abstracción, polimorfismo, herencia y encapsulamiento.

En otras palabras, creo que la libertad en exceso se deforma en libertinaje, y el libertinaje lleva al descontrol, que en este caso lleva a dificultades de mantenimiento y la reingeniería de sistemas.