TP N°3 -TypeScript - OOP

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.

En el JavaScript orientado a objetos basado en prototipos, estas son las "leyes de la física" del lenguaje:

- **Los objetos heredan de otros objetos.** Esta es la regla más fundamental y la que define al paradigma. A diferencia de la herencia clásica (donde una clase hereda de otra clase), aquí un objeto concreto sirve de molde para otro objeto.
- **Funciones Constructoras y el operador** `new`. Una convención fundamental es usar una función (por convención, con mayúscula inicial, ej: `function Tarea()`) como un "constructor" de objetos. La regla es que al invocarla con el operador `new`, se crea un objeto vacío, se establece su prototipo y se le asigna a la palabra clave `this` dentro de la función.
- **La propiedad** `prototype`. Toda función en JavaScript tiene una propiedad pública y no enumerable llamada `prototype`. La regla es que este objeto `prototype` será el prototipo de todas las instancias creadas con esa función constructora a través de `new`. Es el lugar donde se definen los métodos y propiedades compartidos.
  - **Ejemplo de regla escrita:** `Tarea.prototype.setEstado = function() { ... };` Esta sintaxis es la forma canónica de añadir un método compartido a todos los objetos `Tarea`.
- **La Cadena de Prototipos (** `[[Prototype]]` **).** Esta es la regla que gobierna el comportamiento de la herencia. Cuando se intenta acceder a una propiedad de un objeto, el motor de JavaScript primero busca en el propio

objeto. Si no la encuentra, sigue un enlace interno (`[[Prototype]]` , a menudo expuesto como `__proto__`) hacia su objeto prototipo y busca allí. Si tampoco está, continúa subiendo por la cadena hasta llegar a `Object.prototype` o hasta encontrar `null` .

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

   Estas son las creencias compartidas por la comunidad sobre por qué el modelo prototipal es poderoso, efectivo o "mejor" para ciertos contextos que el modelo clásico de otros lenguajes (como Java o C#).

   - **Flexibilidad y Dinamismo Extremos**. Una creencia central es que el modelo prototipal es inherentemente más flexible. Los desarrolladores de JavaScript valoran la capacidad de modificar objetos en tiempo de ejecución.
     - **Por qué se cree que es "mejor":** A diferencia de las clases estáticas, en JavaScript puedes agregar o quitar métodos del prototipo de un objeto en cualquier momento, y todos los objetos que heredan de él reflejarán ese cambio instantáneamente. Esto permite técnicas como el "monkey-patching" y una adaptabilidad que es muy difícil de lograr en lenguajes de herencia clásica.
   - **Simplicidad Conceptual (Objetos a partir de Objetos)**. Muchos defensores del modelo creen que la idea de crear un objeto a partir de otro objeto es más simple y directa que el modelo dual de "clase-instancia".
     - **Por qué se cree que es "mejor":** Una clase es un "plano" abstracto, mientras que una instancia es una "cosa" concreta. En el modelo prototipal, siempre trabajas con "cosas" concretas. Un prototipo no es un plano, es un objeto funcional del que otros objetos pueden clonar o delegar su comportamiento. Se considera un modelo mental más directo. 🧠
   - **Menos "Ceremonia" y Código más Conciso**. El paradigma prototipal permite crear objetos y establecer relaciones de herencia con muy poco código.

- **Por qué se cree que es "mejor":** No necesitas una declaración formal de `class` para empezar. Puedes crear un objeto literal y luego usar `Object.create()` para que otro objeto herede de él. Esto promueve un estilo de codificación más rápido y a menudo más legible para estructuras de objetos simples y medianas, evitando el "boilerplate" de las definiciones de clase.

Ejercicio 4

▼ ⚠️ **Este ejercicio es mucho muy importante.**

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.

Ejercicio 2: Calculadora

En el desarrollo de la clase `Calculadora`, me enfoqué en crear una estructura utilizando la sintaxis de `class`, aplicando los siguientes principios de la POO:

Características Utilizadas

- Abstracción 🧩

  La abstracción, en este caso nos permite ocultar la complejidad interna de la calculadora y exponer únicamente las funcionalidades esenciales. Se creó la clase Calculadora que presenta una interfaz simple con métodos claros como suma(), resta(), división() y multiplicación().

  - **Ejemplo y Fundamentación:** Para realizar una operación, simplemente se instancia un objeto y se llama al método correspondiente. El consumidor de la clase no necesita conocer la lógica interna de la suma ni la validación que realiza el método `division()` para evitar la división por cero. Esto simplifica su uso y hace el código más legible.

    Ejemplo:

```
let suma = new Calculadora(num1, num2);
suma.suma();
```

- Encapsulamiento 📦

  El encapsulamiento nos permite proteger la integridad de los datos del objeto y asegurar que solo se puedan manipular de la forma prevista.

  - **Ejemplo y Fundamentación:** Agrupé los datos (los atributos `num1` y `num2`) junto con las operaciones que los manipulan (los métodos) dentro de la misma clase `Calculadora`. Declaré los atributos como `private`, lo que impide que se pueda modificar su estado de forma directa desde el exterior. Para acceder o cambiar sus valores, implementé métodos públicos (`getters` y `setters`) como `getNum1()` y `setNum1()`. De esta manera, mantengo el control sobre el estado del objeto, garantizando su consistencia.

Características No Utilizadas

- Herencia y Polimorfismo 🧬🎭

  En este ejercicio, no implementé herencia ni polimorfismo, ya que no fueron necesarios para cumplir con los requisitos del problema.

  - **Fundamentación:** La razón es que la funcionalidad de la calculadora no requería de una jerarquía de clases. No existía la necesidad de crear tipos especializados (como una `CalculadoraCientifica` que heredara

comportamientos de la `Calculadora` base). Como consecuencia directa de no usar herencia, el polimorfismo tampoco fue aplicable.

Ejercicio 3: Gestor de Tareas

Características Utilizadas

- Abstracción 🧩

  El objeto Tarea encapsula toda la información y la lógica para su manipulación, exponiendo métodos sencillos para modificar sus propiedades.

  - **Ejemplo y Fundamentación:** En el archivo `modificarTarea.ts`, cuando se necesita cambiar el estado de una tarea, simplemente se llama al método `tareaEditada.setEstado()`. El código que llama a este método no necesita saber cómo se le pregunta al usuario por el nuevo estado ni cómo se valida; esa complejidad está oculta (abstraída) dentro de la implementación del método.

- Herencia (Prototipal) 🧬

  En este caso, utilicé la herencia prototipal, que es el mecanismo de herencia nativo de JavaScript. En lugar de usar la sintaxis class, definí los métodos (setDescripcion, setEstado, etc.) directamente en el prototype de la función constructora Tarea.

  - **Ejemplo y Fundamentación:** Al definir `Tarea.prototype.setDescripcion = function() {...}`, logro que todas las instancias creadas con `new Tarea(...)` compartan la misma implementación de este método. No se crea una copia de la función para cada objeto de tarea, lo cual es más eficiente en memoria. Cada objeto `Tarea` *hereda* estos métodos de su prototipo, demostrando un pilar clave de la POO.

Características Parcialmente No Utilizadas

- Encapsulamiento 📦

  Si bien el código agrupa los datos (titulo, descripción, etc.) y los métodos en un concepto único (Tarea), no implementa el ocultamiento de datos de forma estricta.

- Polimorfismo 🎭

  Esta característica no fue utilizada en el proyecto.

  - **Fundamentación:** El polimorfismo requiere la existencia de una jerarquía de herencia donde distintas clases hijas responden de manera diferente a un mismo mensaje. Dado que en mi implementación solo existe un tipo de objeto (`Tarea`) y no hay clases que hereden de ella para especializar su comportamiento, no hubo necesidad de aplicar el polimorfismo.