Aplicando lo Aprendido 3

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

Respuestas:

1.
   - Todo (casi) es un objeto: La regla fundamental. Los primitivos (números, strings) se comportan como objetos cuando se accede a sus propiedades (gracias al "auto-boxing"), y las funciones son objetos de primera clase.
   - Objetos como contenedores dinámicos: Los objetos son colecciones de pares clave-valor (propiedades). Estas propiedades pueden ser agregadas, modificadas o eliminadas en cualquier momento (en tiempo de ejecución).
   - El enlace [[Prototype]]: Cada objeto tiene una propiedad interna, denominada [[Prototype]], que es un enlace (o referencia) a otro objeto. Este enlace puede ser null.
   - La cadena de prototipos (Prototype Chain): Cuando se intenta acceder a una propiedad en un objeto, el motor de JS primero busca en el propio objeto. Si no la encuentra, sigue el enlace [[Prototype]] y busca en ese objeto. Este proceso continúa recursivamente hasta que se encuentra la propiedad o se llega al final de la cadena (donde [[Prototype]] es null). Esto es delegación, no copia.
   - Object.create(): La forma "pura" de crear un objeto con un prototipo específico. let obj = Object.create(unPrototipo); crea obj cuyo [[Prototype]] es unPrototipo.
   - Funciones constructoras y new: Cualquier función puede ser usada como un "constructor" con la palabra clave new. Cuando se usa new MiFuncion(), JS crea un objeto nuevo y automáticamente establece el [[Prototype]] de ese nuevo objeto para que apunte al objeto que se encuentra en la propiedad prototype de la función constructora (es decir, MiFuncion.prototype).
   - class es azúcar sintáctico: La sintaxis de class introducida en ES6 (2015) no cambió este modelo. Sigue siendo el mismo mecanismo de prototipos

subyacente; es solo una sintaxis más familiar para programadores de lenguajes clásicos (como Java o C#) sobre el mismo sistema de delegación.
- El contexto dinámico de this: El valor de this no está ligado estáticamente a una clase, sino que se determina en tiempo de ejecución según cómo se invoca la función (por ejemplo, si se llama como obj.metodo(), this es obj).

2.

- La creencia más arraigada es que el dinamismo de JS es una ventaja. La capacidad de crear objetos sobre la marcha (literales {}) sin necesidad de definir una class primero, y la capacidad de modificar cualquier objeto (incluso los prototipos) en tiempo de ejecución, se considera una forma de "meta-programación" ágil y potente.
- Se cree que el modelo de prototipos (delegación) es conceptualmente más simple. En lugar de la "taxonomía" rígida de la herencia clásica (donde una *Clase* define un *molde* y la *Instancia* es una *copia*), en JS simplemente tienes objetos que *delegan* el comportamiento a otros objetos. No hay distinción artificial entre "clase" e "instancia"; todo es solo un objeto.
- Relacionado con lo anterior, los desarrolladores (especialmente los puristas de JS) creen que el modelo de prototipos evita la "ceremonia" de la OOP clásica. No necesitas abstract, interface, final, protected, etc. Si quieres que un objeto herede de otro, simplemente los vinculas (con Object.create()).
- Aunque los prototipos son una forma de herencia, la flexibilidad de JS (objetos dinámicos, funciones de primera clase) hace que los patrones de composición (como *mixins* o simplemente combinar objetos) se sientan más naturales y preferibles a las cadenas de herencia profundas y frágiles que a menudo se ven en la OOP clásica.
- Existe una creencia (especialmente entre desarrolladores experimentados) de que la sintaxis class de ES6 es perjudicial porque *oculta* la verdadera naturaleza del lenguaje (prototipos). Creen que es "mejor" entender y usar los prototipos directamente, ya que class da una falsa sensación de OOP clásica que gotea (genera problemas) cuando el desarrollador inevitablemente se topa con el comportamiento de this o la naturaleza dinámica de los prototipos.

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.

Respuesta:

Abstracción y Encapsulamiento

Estos principios funcionan de manera conjunta en el desarrollo del proyecto. La abstracción consiste en ocultar los detalles complejos y mostrar únicamente lo que es esencial de un objeto, mientras que el encapsulamiento agrupa tanto los datos como los métodos que los manipulan en una sola unidad.

Implementación en el proyecto: 1. Objetos Tarea y GestorTareas : Cada prototipo se comporta como una entidad encapsulada. Por ejemplo, Tarea contiene todos sus atributos (id, titulo, estado, etc.) y los métodos que permiten operar sobre ellos (mostrarResumen , mostrarDetalle). Quien utiliza un objeto Tarea no necesita saber cómo se genera el resumen; simplemente llama a mostrarResumen().

2. Interfaces (ItfTarea , ItfGestorTareas): Las interfaces definen un contrato que determina qué propiedades y funciones debe tener un objeto para considerarse una Tarea o un GestorTareas . Esto permite enfocarse en la funcionalidad que ofrece el objeto, sin exponer la implementación interna.

Ejemplo práctico: En interfaz.ts , al listar las tareas se ejecuta t.mostrarResumen() para cada objeto. No es necesario acceder manualmente a id, titulo o procesamiento está protegido dentro del método del objeto.

Herencia mediante Prototipos

La herencia permite crear nuevos objetos que reutilizan y amplían el comportamiento de otros. En este caso, se aplica la herencia basada en prototipos, donde los objetos heredan directamente de otros objetos preexistentes.

Cómo se aplicó: Los métodos comunes se asignan al prototype de las funciones constructoras Tarea y GestorTareas . Así, cuando se instancia un objeto con una referencia a new Tarea(...), este mantiene Tarea.prototype . Si se invoca un método que no existe en la instancia, JavaScript lo busca en el prototipo.

Ejemplo práctico: Métodos como agregarTarea , listarTareas o el prototipo de buscarTareas se colocan en GestorTareas . Esto asegura que solo exista una copia de cada función en memoria, compartida por todas las instancias (aunque en este caso se use solo una).

Polimorfismo

El polimorfismo permite que distintos tipos de objetos respondan de manera particular a la misma acción o mensaje.

Implementación en este proyecto: Aunque discreto, el polimorfismo aparece gracias a las interfaces. Por ejemplo, la función mostrarListadoTareas en interfaz.ts recibe un arreglo de sin importar si los elementos son instancias de ItfTarea Tarea o, en un futuro, de tipos como TareaUrgente o TareaConSubtareas .

Característica no utilizada

No se empleó la herencia basada en clases con class

Razones:

1. Modelo sencillo: El gestor maneja un único tipo de tarea, por lo que no se requería una jerarquía de clases compleja ni subtipos.

2. Diseño con prototipos: El uso directo de prototipos simplifica el código y puede ser más eficiente para este tipo de proyectos.

3. Ausencia de relaciones "es un": No existía la necesidad de modelar subtipos de tareas que compartieran propiedades pero que tuvieran comportamientos diferentes. El proyecto se mantuvo funcional con un solo tipo de objeto.