Paradigmas de Programación

Alumno: Lázaro León Sánchez

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?

1) Las reglas escritas del paradigma de programación orientada a objetos basado en prototipos en JavaScript incluyen:

   Herencia prototípica:

   En lugar de clases, JS utiliza **prototipos** para compartir propiedades y métodos entre objetos. Cada objeto puede tener un vínculo interno a un prototipo (su prototipo padre), que se accede a través de la cadena de prototipos.

   Creación dinámica de objetos:

   Los objetos se pueden crear y extender dinámicamente en tiempo de ejecución.

   Propiedades y métodos mutables:

   Se pueden modificar o añadir nuevas propiedades/métodos a los objetos o sus prototipos dinámicamente.

   Métodos integrados para gestionar prototipos:

   Métodos como Object.create(), Object.getPrototypeOf(), Object.setPrototypeOf(), y el acceso al prototipo mediante __proto__ permiten un control explícito de la herencia.

   Constructor Functions y ES6+ Classes:

   Antes de ES6, la creación de "clases" se realizaba mediante funciones constructoras y la palabra clave new.

   A partir de ES6, la sintaxis de class es un azúcar sintáctico que sigue basándose en la herencia prototípica.

2) En la comunidad de desarrolladores, ciertas características de JavaScript dentro del paradigma prototípico son percibidas como ventajas:

Flexibilidad y dinamismo:

La capacidad de modificar objetos y prototipos en tiempo de ejecución es vista como una ventaja significativa en entornos que requieren adaptabilidad.

Simplicidad inicial:

La herencia prototípica es conceptualmente más simple que la herencia basada en clases tradicionales porque solo se trabaja con objetos. Esto facilita la creación de prototipos rápidos y la comprensión inicial.

Compatibilidad con paradigmas múltiples:

Aunque orientado a prototipos, JS permite aplicar principios de programación funcional y basada en clases. Esta flexibilidad hace que sea más versátil y atractivo para distintos casos de uso.

Evolución y adopción moderna:

La introducción de class en ES6 hizo más accesible la herencia prototípica para quienes venían de lenguajes basados en clases (como Java o C#), facilitando la transición a JS.

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.

En el caso del ejercicio 2, la calculadora, las características que utilice fueron:

Encapsulamiento: Ciertas propiedades y métodos están encapsuladas, agrupadas en clases creadas en base a ciertos conceptos específicos, en este caso las clases calculadora y menú, que tienen límites claros definidos por su concepto y su propia lógica independiente.

Abstracción: En la clase Calculadora, las operaciones aritméticas que es capaz de realizar están abstraídas en métodos, específicamente sumar(), restar(), multiplicar() y dividir(), las cuales ocultan su funcionamiento especifico detrás de dicha abstracción.

Por otro lado, las que no utilice fueron:

Herencia: Para este caso particular no creí necesario usar herencia ya que cada clase tiene una responsabilidad bastante clara y no se requiere extender las funcionalidades ni de Calculadora ni de Menus utilizando herencias.

Cada clase tiene funciones específicas y están aisladas correctamente, por lo tanto intentar aplicar herencia solo lograría complejizar y complicar más innecesariamente la solución.

Polimorfismo: Debido a que no se usó herencia, no he utilizado polimorfismo debido a la falta de métodos que sobrescribir de una clase a la otra.

Además, debido a la complejidad baja de las operaciones matemáticas y de la interacción que se tiene con el usuario, no hay mucho espacio para variabilidad en el funcionamiento de las mismas, por lo que no es necesario sobrescribir dichos métodos para cambiar esto último.

En el caso del ejercicio 3, la lista de tareas, las características que use fueron:

Encapsulamiento: Un ejemplo de encapsulamiento en el código está en el prototipo de Tarea, este prototipo encapsula todas las propiedades y métodos referidas a la modificación de valores en una tarea dada, por lo que se evita el acceso directo a las propiedades de cada tarea en pos de utilizar métodos de estas que modifiquen dichas propiedades.

Abstracción: Nuevamente, en este ejercicio aplique abstracción al convertir ciertas operaciones complejas que actúan sobre los prototipos en métodos que

faciliten su uso. Un ejemplo es en el prototipo de ListaDeTareas, en donde la funcionalidad de buscar una tarea esta abstraída en el método buscarTarea:

```
ListaDeTareas.prototype.buscarTarea = function (nombre) {
    let coincidencias = [];
```

```
        for(let i=0; i<this.tareas.length; i++){


    if((this.tareas[i].getTitulo()).toLowerCase().includes(no
    mbre.toLowerCase())){

                coincidencias.push(i);

            }

        }

        return coincidencias;

    }
```

Esta lógica y su complejidad es abstraída al ser almacenada en el método buscarTarea, mediante el cual se puede realizar dicha acción fácilmente.

Por otro lado, las características que no utilice fueron:

Herencia: Nuevamente no utilice herencia debido a la falta de distintos tipos de tarea en el ejercicio para usarla. Al tener solo dos entidades que manejar en este ejercicio conceptualmente, estas entidades siendo la de una tarea y la de una lista de tareas, y al no haber distintas variaciones de tareas o listas de tareas con características únicas la una de la otra, no había necesidad de utilizar herencia en este caso.

Polimorfismo: Como este ejercicio se trataba de una lista de tareas simple, y por las mismas razones expuestas en herencia, no hay necesidad de variaciones de distintos métodos en los prototipos, tanto en la lista de tareas como en las tareas en sí mismas.

En ambos prototipos, el de la lista de tareas y el de las tareas sus métodos tienen una funcionalidad especifica y la cual no requiere de variar su lógica para poder funcionar en el contexto del código desarrollado, por lo que no hay necesidad de aplicar polimorfismo.