**Informe sobre Programación Orientada a Objetos en el Código**

En el código proporcionado, se implementan varias características de la Programación Orientada a Objetos (OOP), incluyendo encapsulamiento, polimorfismo, herencia, modularidad y abstracción. A continuación, se explican cada una de estas características con ejemplos específicos de tu implementación.

**1. Encapsulamiento**

**Definición:** El encapsulamiento es la práctica de ocultar los detalles internos de un objeto y exponer solo lo necesario a través de una interfaz. Esto se logra utilizando métodos para acceder y modificar los atributos de un objeto.

**Ejemplo en el código:**

javascript

Copiar código

function Task(title, description = '', difficulty, expireDate) {

this.title = title;

this.description = description;

this.status = "pendiente"; // Valor predeterminado

this.creationDate = new Date();

this.lastEditDate = new Date();

this.expireDate = expireDate;

this.difficulty = difficulty;

}

Task.prototype.changeStatus = function(newStatus) {

this.status = newStatus;

};

En este fragmento, la clase Task encapsula los atributos como title, description, status, etc. Los detalles de cómo se almacenan estos atributos están ocultos, y los métodos públicos como changeStatus permiten interactuar con el objeto de manera controlada.

**2. Polimorfismo**

**Definición:** El polimorfismo permite que diferentes objetos respondan a la misma operación de maneras diferentes, a menudo a través de métodos que comparten el mismo nombre.

**Ejemplo en el código:**

javascript

Copiar código

Task.prototype.getDifficultyStars = function() {

switch (this.difficulty.toLowerCase()) {

case 'baja':

return '★☆☆';

case 'media':

return '★★☆';

case 'alta':

return '★★★';

default:

return 'Dificultad inválida';

}

};

En este caso, el método getDifficultyStars se puede considerar un ejemplo de polimorfismo en el sentido de que el comportamiento del método cambia dependiendo del valor de this.difficulty. Dependiendo de la dificultad, se devuelve un número diferente de estrellas.  
  
actuando en la función de crear una tarea , en la de editarla y mostrarla

**3. Herencia Prototípica (implícita)**

Aunque en el código proporcionado no hay una implementación explícita de herencia, el sistema de prototipos en JavaScript permite que un objeto herede propiedades y métodos de otro.   
No se implementa una herencia explicita ya que todo el tiempo hablamos de un solo tipo y todo gira en torno a una lista de tareas;  
  
En un sistema con varios tipos seria mas útil ej Empleados y tipos gerente,obrero,etc

**4. Modularidad**

**Definición:** La modularidad se refiere a la práctica de dividir un programa en módulos independientes y reutilizables que pueden ser desarrollados y probados por separado.

**Ejemplo en el código:**

javascript

Copiar código

function isValidDay(day) {

return day >= 1 && day <= 31;

}

Aquí, se han creado varias funciones auxiliares (como isValidDay, isValidMonth, isValidDifficulty, etc.) que gestionan tareas específicas. Esta estructura modular permite que el código sea más organizado y fácil de mantener.  
  
si bien se implemento la modularidad utilizando corchete y separando funciones y procedimientos del main ;

No se implemento una modularidad extricta a través de varios archivos archivo.js y uso de (import,export).

**5. Abstracción**

**Definición:** La abstracción implica representar conceptos complejos a través de modelos simplificados. Esto permite que el programador se concentre en los aspectos relevantes de un objeto sin preocuparse por los detalles complejos.

**Ejemplo en el código:**

javascript

Copiar código

Task.prototype.showTask = function() {

console.log("\nTitulo:", this.title);

// ...

};

El método showTask proporciona una forma simple de representar la información de una tarea sin exponer los detalles de su implementación. El usuario de este método solo necesita conocer que al llamar a showTask, obtendrá una representación de la tarea.

**Conclusiones**

En resumen, el código implementa adecuadamente las características de OOP que permiten un desarrollo más organizado y eficiente. Sin embargo, la herencia estricta no se aplicó en este ejercicio específico, ya que las tareas se gestionan adecuadamente dentro de la clase Task sin la necesidad de extenderla en este contexto a otro tipo . La inclusión de herencia podría considerarse en el futuro si se requiere una funcionalidad más compleja o si se introducen diferentes tipos de tareas.