Para ejemplificar el principio de sustitución de Liskov en el contexto de JavaScript y arrays, necesitamos considerar cómo se heredan y se sobrescriben los métodos. Este principio establece que las instancias de una clase derivada deben poder reemplazar a las de la clase base sin alterar el comportamiento esperado.

Aunque el código proporcionado no involucra clases explícitamente, ciertas estructuras de datos o métodos relacionados con arrays pueden extenderse o modificarse sin violar este principio.

Ejemplo 1: Uso de Array.prototype.map() y Array.prototype.filter()
El método map() crea un nuevo array con los resultados de aplicar una función a cada elemento del array original, y el método filter() crea un nuevo array con los elementos que cumplan una condición.

Si extendemos o modificamos un array para incluir funcionalidad adicional, deberíamos poder sustituir el array original por el array extendido sin alterar el comportamiento esperado.

Justificación:
Si creamos una clase que extienda la funcionalidad de un array pero mantiene la misma interfaz de métodos (por ejemplo, map() o filter()), podríamos usar una instancia de esta clase en lugar de un array normal, y seguir obteniendo los mismos resultados.

Por ejemplo, si creamos una clase que extiende el comportamiento de los arrays pero mantiene los métodos estándar como map(), esto cumple con el principio de sustitución de Liskov, porque el código que usa el array puede funcionar con ambos tipos sin necesidad de conocer detalles sobre si estamos trabajando con un array extendido o con un array estándar.

```
class MyArray extends Array {
    customMethod() {
        return "Método personalizado";
    }
}

let myArray = new MyArray(1, 2, 3);
console.log(myArray.map(n => n * 2)); // [2, 4, 6]
console.log(myArray.filter(n => n > 1)); // [2, 3]
console.log(myArray.customMethod()); // "Método personalizado"
```

Aquí, MyArray es un array extendido con un método adicional customMethod, pero los métodos estándar como map() y filter() siguen funcionando como en un array regular. Esto respeta el principio de sustitución de Liskov porque podemos sustituir un array tradicional por una instancia de MyArray sin que cambie el comportamiento de los métodos que esperamos usar.

Ejemplo 2: Uso de Array.prototype.concat()

El método concat() puede combinar dos o más arrays en uno solo. Supongamos que tenemos una clase que extiende Array y sobrescribe el método concat para agregar una funcionalidad extra, como agregar un prefijo a los elementos del array concatenado.

Justificación:
Si el método concat() de la clase extendida sigue respetando la firma original (es decir, devuelve un nuevo array con los elementos combinados), los clientes que utilizan este método no se verían afectados por la extensión, ya que siguen obteniendo un array que contiene los elementos concatenados.

```
class MyArray extends Array {
   concat() {
      const result = super.concat(...arguments);
      result.push('Nuevo Elemento');
      return result;
   }
}

let array = new MyArray(1, 2, 3);
let result = array.concat([4, 5]);
console.log(result); // [1, 2, 3, 4, 5, "Nuevo Elemento"]
```

En este caso, la clase MyArray sobrescribe concat para agregar un "Nuevo Elemento" al array concatenado. Sin embargo, cualquier código que llame a concat() en una instancia de MyArray sigue recibiendo un array como resultado, lo que cumple con el principio de sustitución de Liskov.

Conclusión
En ambos ejemplos, podemos ver que la funcionalidad básica de los arrays no cambia al extenderlos o sobrescribir métodos. Esto es una aplicación del principio de sustitución de Liskov, donde las clases derivadas pueden sustituir a la clase base (en este caso, Array) sin que el comportamiento de los métodos estándares (como map(), filter(), concat()) se vea alterado. Los clientes de estas clases no necesitan saber si están trabajando con un array estándar o una versión extendida, lo que garantiza la transparencia y la consistencia del código.