Práctica 2: JavaScript (front-end)

Operaciones con productos y carrito de compras

Objetivo: Crear un conjunto de funciones en JavaScript que permitan simular operaciones de altas, bajas, cambios y consultas dentro de una aplicación de e-commerce.

Descripción de la aplicación

Se desea tener una colección de funciones desarrolladas haciendo uso de JavaScript que permitirán la emulación de operaciones como altas, bajas, cambios y consultas para la parte de productos y carrito de compras de una aplicación de comercio digital.

La descripción de la actividad presente cuenta con lo siguiente:

1. Consideraciones
2. Configuración inicial
3. Funciones de productos
4. Funciones del carrito de compras
5. Funciones para manejo de datos
6. Búsqueda de productos por distintos criterios
7. Ejemplos de pruebas
8. Entregables
9. Evaluación
10. Consideraciones adicionales

Consideraciones

Para la realización de la práctica se considerará lo siguiente:

1. Se utilizará el archivo **home.html** de la **práctica 1**
2. Se mostrará la funcionalidad de la colección de funciones desde la ***consola del navegador*** y opcionalmente de ***manera visual***
3. En esta práctica *no se espera **aún** la interacción con la página web* (solo se mostrará información); no se habilitarán botones, modales, paginación, etc.

Configuración inicial

Crea los siguientes archivos de JavaScript:

- product.js
- shopping_cart.js
- utils.js
- data_handler.js
- index.js

Se puede hacer uso de estos archivos de múltiples formas:

1. Importando todos los archivos en **home.html**
    a. Poniendo los scripts en el encabezado (head) y utilizando el atributo ***defer***
    b. Poniendo los scripts *justo antes de cerrar el contenido del documento* (body)
2. Importando únicamente **index.js** y desde este archivo importar los scripts restantes (Pregunta cómo se haría esto)

Para el primer caso, el último archivo que debería importarse sería **index.js,** ya que este depende de los demás.

Funciones de productos

Para poder utilizar correctamente los productos en la aplicación, se necesita una manera de representarlos a través de objetos de JavaScript. Para atender este propósito, serán necesarias 2 clases dentro de **product.js**: ***Product*** y ***ProductException***. La segunda será utilizada para levantar excepciones personalizadas.

La clase **Product** debe ser capaz de guardar los siguientes datos:

```
[{
    “uid”: “eg3119b6-2d51-5ee2-0ec8-9bbd14bf3gc”,
    “title”: “Plátano”,
    “description”: “Plátano Chiapas 8.90 la pieza bien fresca y tersa”,
    “imageUrl”: “https://images.freeimages.com/images/large-
    previews/4ec/banana-s-1326714.jpg”,
    “unit”: “pieza”,
    “stock”: 15,
    “pricePerUnit”: 3.6,
    “category”: “Fruta”
}]
```

Se necesitarán **getters** y **setters** que hagan las validaciones adecuadas (los cuales levantarán excepciones de tipo **ProductException**), como revisión de cadenas vacías, números negativos en los artículos, congruencia de datos, etc.

Para el UUID, se levantará una excepción desde el **setter** correspondiente para solo permitir el uso interno del dato, es decir, no se puede acceder a esta propiedad desde fuera de la clase (esto puede manejarse con la restricción de la propiedad desde el **setter** y adicionalmente se puede definir como atributo privado). También, se utilizará la siguiente función para “auto-generar” los **UUIDs** al crear los productos.

```
function generateUUID() {
    return 'xx3xxxxx-x8xx-xxxx-yxxx-xxxx5xxxxxxx'.replace(/[xy]/g, c => {
        let r = Math.random() * 16 | 0;
        let v = c == 'x' ? r : (r & 0x3 | 0x8);
        return v.toString(16);
    });
}
```

Este código irá en el archivo **util.js**.

Para poder interactuar con el servidor posteriormente, la funcionalidad deberá ser capaz de interpretar *objetos* externos como productos, para lo cual se crearán las siguientes funciones ***estáticas***:

- **createFromJson**(jsonValue): Esta función debe convertir la representación de un **JSON** contenida dentro de una **String** en una nueva instancia de producto (utilizando la clase **Product**)
- **createFromObject**(obj): Esta función debe convertir el **objeto** recibido en una nueva instancia de producto (utilizando la clase **Product**) y *debe ser capaz de ignorar todos aquellos valores que no pertenezcan a la clase **Product***
- **cleanObject**(obj): Esta función debe limpiar el **objeto** recibido de todos aquellos valores que no pertenezcan a la clase **Product**.

Funciones del carrito de compras

Dentro del archivo **shopping_cart.js**, se agregarán 3 clases: **ShoppingCart, ProductProxy** y **ShoppingCartException**.

Para referenciar los productos que serán agregarán al carrito de compras de la aplicación, se utilizará la clase **ProductProxy**, en donde se guardará *únicamente* el **UUID** del producto y la cantidad a comprar.

Dentro de la clase **ShoppingCart**, se guardarán 2 arreglos: uno para los **proxies** y otro de **productos**, en donde se guardará una copia de los verdaderos productos a comprar. Se debe validar que el usuario *no pueda modificar directamente el arreglo de **proxies***, para lograr esto, se agregarán los siguientes métodos:

- **addItem**(productUuid, amount): Esta función debe agregar al carrito un nuevo **ProductProxy**, o en caso de que ya exista el producto, actualizar la cantidad a la suma de ambos valores (cantidad previa + cantidad nueva)
- **updateItem**(productUuid, newAmount): Esta función debe actualizar el producto correspondiente a la nueva cantidad. Si el nuevo valor es negativo, se levantará una excepción; si es igual a 0, se eliminará el producto del carrito; si es mayor a 0, entonces se actualizará al nuevo valor.
- **removeItem**(productUuid): Esta función debe eliminar el **producto** correspondiente

- **calculateTotal**(): Esta función debe calcular el valor total de la compra, utilizando la cantidad correspondiente a cada producto y el valor unitario del mismo.

Funciones para manejo de datos

Para esta parte se utilizará el archivo **data_handler.js** y dentro del mismo se definirá un arreglo *constante* de productos, el cual será la lista oficial de productos a la venta.

Se deben agregar los métodos correspondientes para leer (**getProducts**(), **getProductById**(**uuid**)), para crear (**createProduct**(**product**)), para actualizar (**updateProduct**(**uuid**, **updatedProduct**)), y para eliminar (**deleteProduct**(**uuid**)) productos.

Además de estos métodos básicos conocidos como *CRUD*, de manera opcional, es posible agregar un método de búsqueda por categoría o por nombre del producto. Para esto se crearía una función **findProduct**(**query**), en donde *query* es un String, en el cual se pondría la categoría y el nombre a buscar en el siguiente formato **“<category>:<title>”**.

En caso de que el usuario pase solo la categoría (**“<category>:”**), se realizará la búsqueda de los productos que contengan esa cadena dentro de su *categoría*. De manera similar, si el usuario solo pasa el nombre (**“<title>”**) se hará la búsqueda de los productos que contengan esa cadena dentro de su *nombre*. Para el caso combinado, se deberán consultar ambas partes y aplicar ambos filtros para la lista de productos.

Ejemplos de pruebas

Dentro de **index.js** se mandarán a llamar a las diferentes funciones creadas en los pasos anteriores para validar que todo se ejecute según lo descrito.

Para las pruebas de la clase de **productos**, se poblará la lista de productos encontrada en **data_handler.js** agregando valores similares a los que se tenían en la **práctica 1**; deben agregarse por lo menos *4 elementos*, después se debe actualizar alguno de estos, posteriormente se debe eliminar alguno, y por último, *en caso de haber implementado la funcionalidad de búsqueda*, hacer 3 búsquedas: una solo por categoría, otra solo por nombre, y otra combinada.

Para las pruebas de la clase del carrito de compras, se agregarán 3 elementos de la lista de productos a un *nuevo* objeto de tipo **ShoppingCart**, después se actualizará alguno de ellos y también se eliminará algún otro. Finalmente, se deberá calcular el total del carrito, para lo cual será necesario llenar con valores reales los productos del carrito de compras.

Manejo del DOM: De manera *opcional*, es posible cargar los elementos directamente en el contenedor principal de **home.html** usando la propiedad ***innerHTML***. Esto requiere crear un método que "mapee" los valores de algún producto a una cadena con una plantilla de HTML correspondiente a la estructura base de una tarjeta (card) como las que se debieron utilizar en la **práctica 1**. Los conocimientos necesarios para lograr esto sin ayuda se verán más adelante en la clase, sin embargo, puedes preguntar al profesor para orientarte en esta implementación para la práctica actual.

Entregables

1. Archivo .zip (no .rar ni otros formatos) con los archivos solicitados
    a. Los archivos de código que se solicitan son 6 (un solo html, cinco js), no más, no menos
2. Vídeo *narrado* demostrando la práctica y todas las funcionalidades solicitadas en las secciones anteriores; se recomienda que te apoyes de este documento para demostrar cada parte de la misma

Evaluación

Sección	Puntuación
Clase **Product**	15 puntos
Clase **ShoppingCart**	15 puntos
Uso de excepciones y validaciones	10 puntos
Funciones estáticas de **Product**	20 puntos
CRUD del carrito de compras	20 puntos
CRUD en **data_handler.js**	15 puntos
Pruebas y ejemplos	5 puntos
Extra: Búsqueda por categoría/nombre	10 puntos
Extra: Manejo del DOM	5 puntos

Máxima calificación de la práctica: 100 puntos.

Consideraciones adicionales

- El alcance de la práctica llega hasta lo descrito únicamente, por ende, si se desean realizar implementaciones adicionales y no funcionan, habrá una penalización por ello
- El vídeo debe ser conciso; puedes explayarte tanto como desees, siempre y cuando lo que expliques sea relevante para la práctica. Procura no producir algo extenso
- Todo el material necesario para la realización de la práctica (a excepción de métodos estáticos y DOM) fue revisado en clase. De esta aseveración se desprenden dos cuestiones:
    - En teoría no tendría que ser necesaria la investigación de métodos adicionales para la implementación, sin embargo, no está prohibido la búsqueda de referencia para métodos adicionales, *solo asegúrate de explicar por qué los usas y cómo los implementas*
    - No pierdas tiempo tratando de reinventar la rueda, es decir, es posible que trates de implementar alguna función cuando ya existe un método para resolver el problema inicial
- Si no estás seguro de algo o te parece ambigua alguna instrucción, ***PREGUNTA***
    - No hagas asunciones, es posible que tu entendimiento de algún punto sea distinto a lo que se pide
- **Para aquellos que desarrollaron vistas personalizadas en la práctica pasada:** La práctica pasada brindó la flexibilidad de implementación en cuanto a la temática y diseño de los entregables, sin embargo, se propuso como base una tienda en línea ya que la práctica presente y posteriores van de la mano en cuestión de contenido. La simulación de los productos y el carrito de compras mantienen la línea del tema manejado hasta el momento, ***no obstante***, si tú implementaste una temática distinta, debes ajustar las instrucciones de este documento para que se alineen con tu práctica anterior
    - Ejemplo: temática de videojuegos
        - Los atributos de producto **unit** y **pricePerUnit** no podrían ser trasladados a esto *a menos que se considere un almacén/emporio de videojuegos*.
        - El atributo **category** *sí* podría usarse (haría referencia al género del juego)

El propósito es adecuar tu caso (si aplica) para mostrar un carrito similar. *Tu implementación no puede tener menos atributos (8) que los mostrados y los únicos que se podrían modificar serían **unit, stock, pricePerUnit,** y **category***.