Práctica Evaluable Tema 7.

Gestión dinámica de memoria

Objetivos.

- Repasar los conceptos estudiados hasta ahora.

Consideraciones iniciales.

- La práctica consiste en un único proyecto a realizar en Visual Studio o similar, que se evaluará sobre 10 puntos
- Cada clase debe estar en un fichero fuente propio, con el nombre de la clase y extensión “.cs”, como en los ejercicios que habéis hecho durante el tema.
- El programa principal (la clase con el método "Main") podrá guardarse en el archivo que se crea por defecto, llamado "Program.cs".
- Como en otras prácticas evaluables anteriores, el proyecto debe compilar, ya que de lo contrario se evaluará con un 0.

Código implementado

Para cada archivo fuente entregado se deberá incluir como comentario en las primeras líneas del archivo el nombre del autor, y una breve descripción de en qué consiste el archivo o clase.

Además, en la clase principal (la que tenga el método Main) se incluirá un listado de todos los apartados, indicando si han sido implementados totalmente, parcialmente o no ha sido realizado.

Por ejemplo:

```
/*
Perez Gomez, Andres
Practica Evaluable Tema 7
Apartado 1 si / no / parcialmente
Apartado 2 si / no / parcialmente
Apartado 3 si / no / parcialmente
...
*/
```

Entrega.

Se debe entregar un archivo comprimido ZIP con el proyecto Visual Studio completo.

- Nombre del archivo: **Apellidos_Nombre_PracT7.zip**

  Por ejemplo, si te llamas Andrés Pérez Gómez el archivo debe llamarse *Perez_Gomez_Andres_PracT7.zip*

Desarrollo.

Ejercicio "Tienda de Informática".

Nombre del proyecto: "TiendaInformatica"

Puntuación máxima: 10 puntos

La práctica de este tema va a consistir en la realización de un proyecto en Visual Studio (o algún entorno equivalente, como Xamarin o SharpDevelop), en el que implementemos varias clases con sus relaciones entre ellas, y gestionemos colecciones de las mismas.

Se simulará el comportamiento de una tienda de informática que ofrece un catálogo de productos. Almacenaremos los productos en un diccionario para poder acceder a cualquiera de ellos en cualquier momento.

1. Estructura de clases para el catálogo (2 puntos)

(1 punto) Partiremos de una clase base, llamada **ProductoInformatico,** que almacenará los datos generales de cualquier producto informático. Dichos datos serán un código alfanumérico, la marca, el modelo y el precio (con decimales). Se deberán definir todos mediante un constructor,. También, se deberán definir las propiedades get/set públicas para acceder a los atributos de la clase. Se añadirá también un método *Mostrar* que muestre por pantalla los datos del producto.

Por simplicidad, solo se considerarán tres subtipos de productos: periféricos, portátiles y otros componentes.

- De los periféricos nos interesa saber su nombre (por ejemplo, "Ratón Bluetooth"), y el tipo de conexión (por ejemplo, "USB", "USB-C", "RJ45"...).
- De los portátiles nos interesa el tamaño de la pantalla en pulgadas (con decimales), la cantidad de RAM instalada (en GB) y el tamaño de disco (también en GB)
- De los otros componentes nos interesa su nombre (por ejemplo, "Conector PLC"), y una breve descripción de su uso.

(1 punto) Se creará una clase para cada uno de estos tres subtipos, heredando de la clase padre *ProductoInformatico*, definiendo los correspondientes constructores, atributos privados y propiedades get/set públicas. Se redefinirá también el método *Mostrar* del padre para añadir la información de los subtipos.

2. El catálogo (4 puntos)

Para gestionar el catálogo de productos, crearemos una clase **Catalogo**.

(0,5 puntos) Internamente, tendrá como atributo un diccionario genérico (*Dictionary*), cuya clave será una cadena, y cuyo valor será un *ProductoInformatico*, del tipo que sea. En este diccionario almacenaremos cada producto, identificado por su clave (el código del producto). El catálogo será un atributo privado, y se inicializará (*new*) en el constructor de la clase.

Se proporcionarán los siguientes métodos públicos para gestionar el catálogo:

- **(1 punto)** *NuevoProducto:* recibirá como parámetro un objeto de tipo *ProductoInformatico* ya creado (NO se debe crear dentro de este método), y deberá validar que sus datos son correctos, y añadirlo al diccionario. Las comprobaciones que debe hacer son las siguientes:
  - Ningún atributo de tipo texto puede estar vacío.

- Todos los datos numéricos (precio, y si procede, memoria RAM o tamaño de disco) deben ser positivos (mayores que 0).
- El código del producto no debe existir ya en las claves del diccionario del catálogo.

En el caso de que se cumpla todo esto, se añadirá una nueva entrada en el diccionario, con el código del producto como clave, y el producto entero como valor. Si algo no es correcto, no se añadirá el producto. Se devolverá un booleano indicando si se ha podido añadir o no el producto al catálogo.

- **(0,5 puntos)** *EliminarProducto*: recibirá como parámetro el código del producto, y lo eliminará del catálogo. Devolverá un booleano indicando si se pudo eliminar o no. En el caso de que el producto no exista en el catálogo, se devolverá *false*, al no haberse podido eliminar.
- **(0,5 puntos)** *ObtenerProducto*: recibirá como parámetro el código del producto, y devolverá el producto (objeto *ProductoInformático*) asociado, o *null* si no se encuentra el producto en el catálogo.
- **(0,5 puntos)** *ListarCatalogo*: no recibirá parámetros, y mostrará por pantalla un listado de los productos del catálogo. Para ello, deberéis recorrer el diccionario con un enumerador, y mostrar por pantalla toda la información de cada producto, llamando a su método *Mostrar*. El listado debe tener un formato más o menos ordenado para poderse leer con claridad.
- **(1 punto)** *CalcularTotal*: recibirá como parámetro una lista de códigos de productos, y devolverá como resultado el precio total de todos ellos, buscándolos en el catálogo. En el caso de que alguno de los productos de la lista no se encuentre en el catálogo, simplemente no se sumará su precio al total.

3. El programa principal (2,25 puntos)

Desde el programa principal, inicialmente se creará un catálogo (objeto de tipo *Catalogo*), y se mostrará continuamente un menú con las siguientes opciones:

1. **(0,25 puntos)** Salir del programa.
2. **(1 punto)** Nuevo producto: se pedirá al usuario que especifique qué tipo de producto añadir (periférico, portátil u otro producto), y en función de eso, se le pedirán los datos, se creará el nuevo objeto y se llamará al método *NuevoProducto* del objeto *Catalogo* para que lo valide, lo añada al catálogo y devuelva el resultado, mostrando por pantalla un mensaje de si todo ha ido bien o ha habido algún error.
3. **(0,5 puntos)** Listar productos: se llamará al método *ListarCatalogo* del objeto *Catalogo* para que muestre por pantalla los productos del catálogo.
4. **(0,5 puntos)** Eliminar producto: se le pedirá al usuario que introduzca el código del producto, y se llamará al método *EliminarProducto* del objeto *Catalogo*, recogiendo el resultado y mostrando por pantalla si la operación se hizo correctamente o no.

4. El carro de la compra (1,75 puntos)

Para simular las compras en la tienda, desde el programa principal vamos a llevar un carro de la compra. Crearemos una lista, donde iremos almacenando los códigos de los productos que vayamos comprando. Será, por tanto, una lista de strings. Esta lista se inicializará al principio del programa, junto con el catálogo, y para gestionarla, añadiremos estas opciones al menú de la aplicación:

5. **(0,5 puntos)** Añadir producto al carro: se le pedirá al usuario que introduzca el código del producto, y si existe en el catálogo (podemos verificar si existe llamando al método *ObtenerProducto* del *Catalogo* y viendo si devuelve *null* o no), se añadirá dicho código a la lista del carro de la compra.
6. **(1 punto)** Ver total: Se mostrarán los datos de cada producto de nuestro carro de la compra, y se utilizará el método *CalcularTotal* del objeto *Catalogo*, pasándole nuestro carro de la compra, para visualizar el total de lo que llevamos comprado.
7. **(0,25 puntos)** Vaciar carro: Se limpiará el carro de la compra (se dejará la lista vacía).

En todos los casos, se valorará de forma positiva la sustitución de los métodos “Mostrar” por la sobrecarga del método ToString.

Observaciones generales de la práctica

- Se deberá controlar con TryParse o mediante un bloque *try-catch-finally* que cualquier dato numérico (entero, real) que se pida sea correcto, y esté entre los límites acordados según el dato en cuestión.
- Cualquier dato de tipo texto será válido siempre que no esté vacío. Si está vacío, se volverá a pedir de nuevo.
- En cada clase que se defina (salvo el programa principal), es OBLIGATORIO que los atributos sean privados o protegidos (esto último en caso de que la clase tenga subclases). Se deberá definir en cada una de estas clases, al menos, un **constructor** que reciba tantos parámetros como atributos tenga la clase, y asigne cada parámetro a su atributo correspondiente. Además, se deben definir las correspondientes **propiedades get/set** públicas para acceder a cada elemento privado.
- Se deberá también redefinir (*override*) el método *ToString* en las clases de las que se quiera sacar algo de información por pantalla, para indicar en dicho método el formato de salida de la información.
- Además de los elementos a añadir indicados en cada clase, puedes añadir otros atributos, constructores o métodos si lo consideras oportuno. Se valorará después si esos elementos añadidos son de utilidad o no.
- Se valorará negativamente:
    - La repetición innecesaria de código en cualquier parte del programa. Por ejemplo, volver a asignar manualmente en el constructor de una clase hija atributos que ya asigna la clase padre, o repetir el código para pedir al usuario los datos del transporte de una etapa, dependiendo del tipo de transporte
    - Las funciones o métodos excesivamente largos o complejos, como por ejemplo, el programa principal, que deberá dividir su funcionalidad adecuadamente en funciones auxiliares.
    - La suciedad de código, en lo referente a lo visto en el Tema 9 del bloque 1 del módulo de Entornos de Desarrollo. Fundamentalmente, se evaluaran las siguientes malas prácticas:
        - Nombres poco apropiados de variables, funciones o clases (ya se venía penalizando con anterioridad).
        - Espaciado y alineación vertical (separación entre funciones y entre bloques de código con propósito diferente).
        - Espaciado y alineación horizontal (incluyendo que las líneas de código no excedan del ancho recomendado).
    - En las funciones que devuelvan algún tipo de dato (*return*), se valorará negativamente que haya más de un punto de salida. Las buenas prácticas de programación indican que las funciones deben tener un único punto de salida (una única instrucción *return*).
    - La utilización incorrecta de las estructuras de control, estructuras repetitivas, las instrucciones,... Algunos casos típicos que :
        - Utilización incorrecta de la estructura de control “switch-case”.
        - La instrucción “goto” no debe utilizarse salvo en “switch-case”.
        - Los bucles “for” sólo deberían utilizarse cuando se conoce el principio y fin del bucle.