Índice

Ejercicios Unidad 19 - Colecciones

Descargar estos ejercicios

> **Antes de empezar**
>
> Para realizar estos ejercicios, deberás descargar los recursos del enlace de proyecto_poo. Como puedes ver, la solución está compuesta de varios proyectos. Cada uno de ellos corresponde con un ejercicio, deberás implementar todo el código, tanto de la Main como de los métodos que se piden en cada ejercicio. Cada proyecto contiene el test correspondiente, que deberás pasar para comprobar que has hecho el ejercicio correctamente.

Ejercicio 1. Gestión de lista de Platos

Implementa un sistema para gestionar una colección de platos utilizando `List<T>` y métodos para realizar operaciones básicas como añadir, eliminar, buscar y filtrar platos según diferentes criterios.

Requisitos

- Define una clase `Plato` con propiedades: `Nombre`, `Precio` (decimal), `Categoria` (enum: Entrante, Principal, Postre).
- Implementa un método `ToString()` que muestre la información del plato.
- Implementa los métodos `Equals(object obj)`, `GetHashCode()` y sobrecarga los operadores `==` y `!=` para que dos platos sean iguales si tienen el mismo nombre y precio.
- Crea la clase `Program` con los siguientes métodos estáticos:
  - Método `BuscaPlato` que utilice `IndexOf` para encontrar la posición de un plato en la lista.
  - Método `AñadePlato` que a partir de una lista y un plato, añadirá este a la lista.
  - `EliminaPlato` que eliminará el plato con la posición en la lista que coincida con el índice `i` que se haya pasado como argumento.
  - `MuestrarLista(List<Plato> lista)` muestra el contenido completo de la lista numerado.
- En el método `Main`, demuestra el uso de todos los métodos creando una lista inicial, realizando búsquedas con `IndexOf`, eliminaciones y adiciones.

Ejercicio 2. Gestión de lista de automóviles

Implementa un sistema para gestionar una colección de automóviles utilizando `List<T>` y métodos para realizar operaciones básicas como añadir, eliminar, buscar y filtrar automóviles según diferentes criterios.

```
Ejercicio 2: Gestión de lista de automóviles

=== LISTA INICIAL DE AUTOMÓVILES ===
Añadiendo automóviles a la lista...
Lista actual (5 automóviles):
1. Toyota Corolla - 1600cc - 2020 - Blanco
2. Honda Civic - 1800cc - 2019 - Negro
3. Ford Focus - 2000cc - 2020 - Rojo
4. Nissan Sentra - 1600cc - 2018 - Azul
5. Volkswagen Golf - 1400cc - 2020 - Blanco

=== BÚSQUEDA POR AÑO DE FABRICACIÓN ===
Automóviles del año 2020:
- Toyota Corolla - 1600cc - 2020 - Blanco
- Ford Focus - 2000cc - 2020 - Rojo
- Volkswagen Golf - 1400cc - 2020 - Blanco

=== BÚSQUEDA POR AÑO Y COLOR ===
Automóviles del año 2020 y color Blanco:
- Toyota Corolla - 1600cc - 2020 - Blanco
- Volkswagen Golf - 1400cc - 2020 - Blanco

=== ELIMINACIÓN DE AUTOMÓVIL ===
Eliminando automóvil en posición 2 (Honda Civic)...
Lista actualizada (4 automóviles):
1. Toyota Corolla - 1600cc - 2020 - Blanco
2. Ford Focus - 2000cc - 2020 - Rojo
3. Nissan Sentra - 1600cc - 2018 - Azul
4. Volkswagen Golf - 1400cc - 2020 - Blanco

=== AÑADIR NUEVO AUTOMÓVIL ===
Añadiendo: BMW Serie 3 - 2000cc - 2021 - Gris
Lista final (5 automóviles):
1. Toyota Corolla - 1600cc - 2020 - Blanco
2. Ford Focus - 2000cc - 2020 - Rojo
3. Nissan Sentra - 1600cc - 2018 - Azul
4. Volkswagen Golf - 1400cc - 2020 - Blanco
5. BMW Serie 3 - 2000cc - 2021 - Gris

Fin de la demostración.
```

Requisitos

- Define un record `Automovil` con propiedades: `Marca`, `Modelo`, `Cilindrada` (int), `AñoFabricacion` (int), `Color`.
- Implementa un método `ToString()` que muestre la información del automóvil en formato como se muestra en la salida.
- Crea la clase `Program` con los siguientes métodos estáticos:

    - Método `AñadeAutomovil` que a partir de una lista y un automóvil, añadirá este a la lista.
    - `EliminaAutomovil` que eliminará el automóvil con la posición en la lista que coincida con el índice `i` que se haya pasado como argumento.
    - Crea un método `AutomovilesPorAñoFabricacion`, que te permita encontrar en la lista los coches con una determinada fecha de fabricación y que retorne una nueva lista con esos datos. ***Usa un bucle for para recorrer la lista y encontrar los automóviles que cumplan la condición.***
    - Otro método `AutomovilesPorAñoFabricacionYColor` que devuelva una sublista con los coches de la lista que sean de un determinado color y una fecha pasados ambos como parámetros. ***Usa un bucle for para recorrer la lista y encontrar los automóviles que cumplan la condición***.
    - `MostrarLista(List<Automovil> lista)` muestra el contenido completo de la lista numerado.
- Validaciones necesarias:
    - Verificar que el índice sea válido antes de eliminar.
    - Comprobar que la lista no sea null en todos los métodos.
    - Manejar casos donde no se encuentren automóviles que coincidan con los criterios de búsqueda.
- En el método `Main`, demuestra el uso de todos los métodos creando una lista inicial, realizando búsquedas, eliminaciones y adiciones.
- (Opcional) Añade métodos adicionales como `AutomovilesPorMarca(List<Automovil> lista, string marca)` o `AutomovilesOrdenadosPorAño(List<Automovil> lista)`.

Ejercicio 3. Comparación y Búsqueda en Colecciones

Vamos a trabajar con diferentes formas de comparación y búsqueda en colecciones.

Requisitos

1. Crea un tipo `record Usuario` con las propiedades: `UserName`, `NombreCompleto` y `FechaRegistro` (DateOnly).
2. Crea una clase `Publicacion` con las propiedades: `Id` (DateTime con Fecha, hora, minutos y segundos), `Autor` (Usuario), `Contenido`, `Likes` (int) y una lista de cadenas `Comentarios`.

Parte 1: Búsqueda lineal con `Contains`

Queremos comprobar si una publicación específica está en la lista usando el método . Crea un Método `EstaPublicación` al que le llega una lista de publicaciones y una publicación y usa el método `Contains` para devolver si está el elemento en la lista.

- **Estrategia (Strategy):** Intenta buscar la publicación usando una clase externa que implemente `IEqualityComparer<Publicacion>` comparando por el `Id`.
- **Natural:** Modifica la clase `Publicacion` para que implemente `IEquatable<Publicacion>` (comparando por `Id`) y prueba de nuevo el método `Contains` sin pasar el comparador externo.

Parte 2: Búsqueda de Usuarios
Crea un método `EstaUsuario` al que le llega una lista ade usuarios y un usuario y devuelve si el usuario pertenece a la lista.

- Crea una lista de usuarios y busca uno concreto usando `Contains`.
- **Reflexión:** Observa que al ser un tipo `record`, no es necesario implementar nada adicional para que la igualdad por valor funcione correctamente.

Parte 3: Búsqueda Binaria
Crea un método `BuscaPublicación` al que le llega una lista de publicaciones y una publicación y devuelve la posición del elemento encontrado. Para ellos se deberá:

- Realizar una `BinarySearch` sobre la lista de publicaciones basándonos en el `Id`. Para que esto sea posible, primero deberás **ordenar** la lista. Además de que la clase `Publicacion` deberá implementar la interfaz `IComparable<Publicacion>` comparando por la propiedad `Id`.

Crea un método `BuscaPublicacionIComparer` que realizará la búsqueda binaria pero usando la sobrecarga a la que se le pasa un objeto de tipo `IComparer<T>`

Ejercicio 4. Diccionario contador de palabras

Utilizando la clase genérica `Dictionary<K, V>` definida en System.Collections.Generic,implementa un sencillo programa de consola que pida nombres por teclado hasta que introduzcamos la cadena "fin".

Requisitos

- Todo el ejercicio se realizará en un método **GestionPalabras** que tendrá el diccionario como variable local.
- Cada nombre se deberá guardar como clave, y las veces que se repite se guardará como valor.
- Al introducir "fin" se mostrarán los nombres introducidos y cuantas veces se ha introducido cada uno.

> **Idea**
>
> Para mostrar el resultado, deberás mostrar por un lado las claves y posteriormente el par clave valor. Para ello tendrás que recorre el diccionario 2 veces, una con un foreach para las claves y otra con un foreach para el diccionario obteniendo pares clave valor con la siguiente clase `KeyValuePair<K, V>`

Ejercicio 5. Diccionario con Clave Personalizada

Vamos a crear un inventario de productos donde la clave del diccionario será un objeto de tipo `Producto` y el valor será el stock disponible (un entero). Este ejercicio demostrará la importancia de implementar correctamente `Equals` y `GetHashCode` cuando se usan objetos propios como claves en un diccionario.

Requisitos

1. Crea una clase `Producto` con las propiedades: `Codigo` (string) y `Nombre` (string).
2. En un método `GestionInventario`, crea un `Dictionary<Producto, int>` llamado `inventario`.
3. **Prueba inicial (sin implementar Equals/GetHashCode):**
   - Crea dos instancias de `Producto` con el *mismo* código y nombre (por ejemplo, `p1` y `p2`).
   - Añade `p1` al diccionario con un stock de 10.
   - Intenta obtener el valor usando `p2` como clave (`inventario[p2]`) o verifica si existe con `inventario.ContainsKey(p2)`.
   - Observa y comenta qué sucede (debería fallar o no encontrarlo).
4. **Implementación correcta:**
   - Modifica la clase `Producto` para que implemente la interfaz `IEquatable<Producto>`.

- Sobrescribe el método `Equals(object obj)` y `GetHashCode()`. Dos productos deben considerarse iguales si tienen el mismo `Codigo`.

5. **Prueba final:**
   - Repite la prueba del paso 3. Ahora `inventario.ContainsKey(p2)` debería devolver `true` y permitirte acceder al stock, demostrando que el diccionario identifica correctamente la clave por su contenido y no por su referencia.

Ejercicio 6. Red Social con Diccionarios

Vamos a crear una estructura básica para una red social utilizando las clases `Usuario` y `Publicacion` del ejercicio anterior, pero gestionándolas con diccionarios para un acceso eficiente.

Requisitos

1. Reutiliza las clases `Usuario` y `Publicacion` definidas en el Ejercicio 3.
2. Crea una clase `RedSocial` que contenga las siguientes colecciones:
   - `Dictionary<string, Usuario> usuarios`: Donde la clave es el `UserName` y el valor es el objeto `Usuario`.
   - `SortedDictionary<DateTime, Publicacion> publicaciones`: Donde la clave es el `Id` de la publicación y el valor es el objeto `Publicacion`. Esto mantendrá las publicaciones ordenadas cronológicamente.
   - `Dictionary<string, List<int>> publicacionesPorUsuario`: Donde la clave es el `UserName` y el valor es una lista de enteros que representan los `Id` de las publicaciones de ese usuario.
3. Implementa los siguientes métodos en la clase `RedSocial`:
   - `RegistraUsuario(Usuario usuario)`: Añade un usuario al sistema. Debe verificar si el usuario ya existe.
   - `private void AñadePublicacionAUsuario(Usuario usuario, int idPublicacion)`: Método privado que añade el `Id` de la publicación a la lista de publicaciones del usuario correspondiente en el diccionario `publicacionesPorUsuario`. Deberá realizar las comprobaciones necesarias, para que no se repitan ID en el usuario.
   - `public void AñadePublicacion(Publicacion publicacion)`: Añade una publicación al `SortedDictionary` de publicaciones y llama al método `AñadePublicacionAUsuario` para vincularla con su autor.
   - `MostrarPublicacionesUsuario(string userName)`: Muestra todas las publicaciones de un usuario específico, recuperándolas a partir de sus IDs.

- `MostrarTodasPublicaciones()` : Muestra todas las publicaciones ordenadas por fecha (gracias al `SortedDictionary`).

4. En el método `Main` , instancia la `RedSocial` , registra varios usuarios, crea y añade publicaciones, y prueba los métodos de visualización.

Ejercicio 7. Lista de Reproducción Musical (LinkedList)

Vamos a simular una lista de reproducción de música donde el orden de las canciones es importante y queremos poder navegar entre ellas (siguiente, anterior) así como insertar canciones en posiciones específicas de forma eficiente. Para ello utilizaremos `LinkedList<T>` .

Requisitos

1. Crea una clase `Cancion` con las propiedades: `Titulo` , `Artista` y `Duracion` (TimeSpan).
2. Crea una clase `ReproductorMusica` que gestione una `LinkedList<Cancion>` llamada `listaReproduccion` .
3. Implementa los siguientes métodos en `ReproductorMusica` :
   - `AgregaCancionAlFinal(Cancion cancion)` : Añade una canción al final de la lista (`AddLast`).
   - `AgregaCancionAlPrincipio(Cancion cancion)` : Añade una canción al principio de la lista (`AddFirst`).
   - `InsertaDespuesDe(string tituloCancionExistente, Cancion nuevaCancion)` : Busca una canción por su título y, si existe, inserta la nueva canción justo después (`AddAfter`). Necesitarás usar `Find` para obtener el nodo.

> ⚠ **Aviso**
>
> Como `Find` busca por valor, necesitarás implementar `Equals` en `Cancion` .

   - `EliminaCancion(string titulo)` : Busca y elimina la primera canción que coincida con el título (`Remove`).
   - `Reproduce()` : Simula la reproducción recorriendo la lista desde el principio hasta el final. Muestra por consola "Reproduciendo: [Titulo] - [Artista]".

- `ReproduceInverso()`: Simula la reproducción en orden inverso (desde la última hasta la primera).

4. En el método `Main`, crea una instancia del reproductor, añade varias canciones, prueba a insertar una entre dos existentes, elimina una y reproduce la lista en ambos sentidos.

Ejercicio 8. Sistema de pedidos de restaurante con colecciones especializadas

Desarrolla un sistema para gestionar pedidos de un restaurante usando colecciones apropiadas para cada tipo de operación: menú ordenado, ingredientes únicos, cola de pedidos y histórico de clientes.

```
Ejercicio 2: Sistema de pedidos de restaurante

=== MENÚ DEL DÍA (por categorías) ===
ENTRANTES:
  - Ensalada César ($8.50)
  - Bruschetta ($6.00)
PRINCIPALES:
  - Paella Valenciana ($15.20)
  - Salmón a la plancha ($18.50)
POSTRES:
  - Tiramisu ($7.80)
  - Flan casero ($5.50)

=== GESTIÓN DE INGREDIENTES ===
Ingredientes disponibles: Lechuga, Tomate, Queso, Arroz, Azafrán, Salmón, Huevos, Café
Platos vegetarianos disponibles: Ensalada César, Flan casero
Alerta: El ingrediente 'Azafrán' está en varios platos (verificar stock)

=== COLA DE PEDIDOS ===
Cola actual (3 pedidos esperando):
1. Mesa 5: Paella Valenciana, Tiramisu - Total: $23.00
2. Mesa 2: Ensalada César, Salmón a la plancha - Total: $27.00
3. Mesa 8: Bruschetta, Flan casero - Total: $11.50

Cocinando pedido de Mesa 5...
Pedido completado. Cola actualizada (2 pedidos).

=== CLIENTES FRECUENTES ===
Top 3 clientes por número de visitas:
1. Juan Martínez - 15 visitas - Gasto promedio: $25.30
2. Ana López - 12 visitas - Gasto promedio: $31.50
3. Carlos García - 8 visitas - Gasto promedio: $19.80

Cliente Juan Martínez - Historial últimas 3 visitas:
- 2025-10-05: $28.50 (Paella + Tiramisu)
- 2025-10-02: $22.10 (Ensalada + Salmón)
- 2025-09-28: $15.20 (Solo Paella)

Fin de la demostración.
```

Requisitos

- Usa la clase `Plato` del ejercicio 1.
- Clase `Pedido`: `NumeroMesa`, `Platos` (List), `FechaHora` (DateTime), `Total` (calculado).
- Clase `Cliente`: `Nombre`, `HistorialPedidos` (List), propiedades calculadas: `NumeroVisitas`, `GastoPromedio`.
- Clase `Restaurante` que use:
  - `SortedDictionary<Categoria, List<Plato>>` para menú organizado.

    - `Queue<Pedido>` para cola de cocina.
    - `Dictionary<string, Cliente>` para clientes frecuentes.
- Métodos principales:
    - `AgregaPlato(Plato plato)` - organiza por categoría.
    - `EncolaPedido(Pedido pedido)` - añade a cola de cocina.
    - `ProcesaPedido()` - saca de la cola y procesa.
    - `ActualizaCliente(string nombre, Pedido pedido)` - actualiza historial.
    - `TopClientesFrecuentes(int cantidad)` - ordena por número de visitas.

Ejercicio 9. Implementación de una Pila Genérica (Wrapper)

Vamos a crear nuestra propia implementación de una pila (Stack) genérica, pero en lugar de usar un array interno (como hace `Stack<T>` de .NET), utilizaremos una `List<T>` privada para almacenar los elementos. Esto nos permitirá "capar" la funcionalidad de la lista y exponer solo las operaciones LIFO (Last In, First Out).

Requisitos

1. Crea una clase genérica `Pila<T>`.
2. Define un campo privado `List<T> _elementos` donde se almacenarán los datos.
3. Implementa los siguientes constructores:
    - **Constructor vacío**: Inicializa la lista interna.
    - **Constructor con** `IEnumerable<T>`: Permite inicializar la pila con los elementos de cualquier colección existente (Array, List, etc.).
4. Implementa los métodos típicos de una pila:
    - `Push(T elemento)`: Añade un elemento a la cima de la pila (final de la lista).
    - `Pop()`: Devuelve y elimina el elemento de la cima. Debe lanzar una excepción `InvalidOperationException` si la pila está vacía.
    - `Peek()`: Devuelve el elemento de la cima sin eliminarlo. También debe lanzar excepción si está vacía.
    - `EstaVacia()`: Devuelve `true` si no hay elementos.
5. Implementa una propiedad `Count` que devuelva el número de elementos.
6. Sobrescribe `ToString()` para mostrar el contenido de la pila.
7. En el método `Main`, crea una pila de enteros, apila varios números, desapila uno, y prueba a crear otra pila de cadenas a partir de un array de strings usando el segundo constructor.

Ejercicio 10. Patrón Iterator en la Pila Genérica

Vamos a ampliar la clase `Pila<T>` del ejercicio anterior para que sea posible recorrer sus elementos utilizando un bucle `foreach`.

Requisitos

1. **Intento inicial:** En el `Main`, intenta recorrer una instancia de tu `Pila<T>` con un bucle `foreach`. Observarás que el compilador da un error porque tu clase no implementa `IEnumerable` ni tiene un método `GetEnumerator`.
2. **Implementación del Iterador:**
   - Modifica la clase `Pila<T>` para que implemente la interfaz `IEnumerable<T>`.
   - Implementa el método `GetEnumerator()` utilizando la palabra clave `yield return`.
   - *Nota:* El recorrido debe hacerse desde la cima (último elemento añadido) hacia el fondo, respetando la naturaleza LIFO de la estructura.
3. **Prueba final:**
   - Vuelve a probar el bucle `foreach` en el `Main`. Ahora debería funcionar correctamente y mostrar los elementos de la pila.