Recuperación Trabajos Prácticos 1 al 3

> Plazo de entrega: Sabado 19 hasta las 23:59.

Dado que la asistencia es obligatoria y la presentación de los trabajos prácticos es la comprobación de dicha asistencia, se ha decidido brindar la oportunidad de recuperar los trabajos prácticos a aquellos estudiantes que no los hayan presentado a tiempo.

Cabe destacar que, en el caso del Trabajo Práctico 3, también deberán recuperarlo aquellos estudiantes que, habiéndolo presentado, no hayan logrado que funcione correctamente.

Los estudiantes que deban recuperar deberán realizar el trabajo y presentarlo de manera correcta para que sea aceptado.

Estos tres trabajos prácticos sirvieron, en parte, para poner en funcionamiento el sistema de corrección, que ya está automatizado. Si no se siguen estrictamente las reglas de presentación, el trabajo presentado no será tomado en cuenta por el sistema.

Cada trabajo práctico que deba recuperarse debe presentarse de forma independiente, asegurándose de subir únicamente el archivo `ejercicio.cs` con la solución funcionando como un `script` . Es decir, si deben recuperar dos trabajos prácticos, deberán realizar dos solicitudes de incorporación (`pull request`), una para cada TP.

Recuerden que para poder presentar el trabajo deberan primero pasa a la rama principal y actualizar el repositorio asi tiene los ultimos cambios realizados. Luego crear una rama nueva para implementar la solución y finalmente hacer la solicitud de incorporacion.

Este proceso debe repetirse para cada uno de los practicos que se deben recuperar.

En el caso del TP3, todos los tests deben estar incluidos en el código, y el programa debe ejecutarse sin errores.

Listado de alumnos que deben recuperar

Comisión C3

Legajo	Nombre Completo	TP1	TP2	TP3
61203	Acevedo Costello, Juan Ignacio		Recuperar	Recuperar

Legajo	Nombre Completo	TP1	TP2	TP3
62055	Ahumada, Aiquén			Corregir
61118	Barrios, Santiago Alexis			Recuperar
61214	Collazos Cortez, Máximo Alberto		Recuperar	Recuperar
61221	Duclós, Marcelo Ezequiel			Corregir
62093	Frías Silva, Juan Segundo			Recuperar
61139	Gallo, María Matilde			Corregir
61624	Godoy, Alan		Recuperar	
61595	González Patti, Valentín			Corregir
61562	Helguera, Agustina Elizabeth		Recuperar	Recuperar
61818	Lopez Tisera, Gaston			Recuperar
61579	Marti, Gonzalo		Recuperar	Recuperar
61676	Massey, Maximiliano	Recuperar		Recuperar
61113	Oriz Caballero, Juan José			Recuperar
61572	Parrado Navarrete, Alex Daniel			Corregir
61793	Trujillo, Facundo Nahuel		Recuperar	
61596	Zamora, Gonzalo Alejandro		Recuperar	Recuperar

Comisión C5

Legajo	Nombre Completo	TP1	TP2	TP3
62612	Campos, Julieta Antonella		Recuperar	Recuperar
61611	Coronel, Julieta Antonella		Recuperar	Recuperar
61535	Mainardi, Facundo			Recuperar
61905	Martinez, Augusto		Recuperar	Corregir
61588	Mussa, Agustín			Recuperar
61912	Nuñez, Arturo Valentin	Recuperar	Recuperar	
61644	Ortega, Fernando Nicolas			Recuperar

Legajo	Nombre Completo	TP1	TP2	TP3
61140	Perez, Fernando Alberto			Corregir 🚩
61962	Robledo, Rocio Micaela			Corregir 🚩
61061	Velardez, Leandro Ivan			Recuperar

> Nota: Si alguno de los estudiantes que deben recuperar ya ha presentado los trabajos, comuníquense conmigo para verificar la situación.
> Nota 2: Les adjuntos los enunciados de los trabajos practicos a este documento

TP1 : Agenda de Contactos

> El pull request se debe llamar "TP1 - Legajo - NombreCopleto"

Diseña un programa de consola que administre una **agenda de contactos** cumpliendo los siguientes requisitos:

1. **Identificación por ID**
   - Cada contacto tendrá un **ID único** (entero).
   - El programa debe asignar automáticamente el ID de forma incremental a cada contacto nuevo.
2. **Estructura de Almacenamiento**
   - Debes utilizar un `struct` (por ejemplo, `struct Contacto`) con los campos `Id`, `Nombre`, `Telefono` y `Email`.
   - Se recomienda usar un **array** para guardar los contactos.
   - No se permite el uso de clases personalizadas, LINQ, `foreach` ni enumeradores. Solo **struct**, **arrays**, `if` y `for`.
3. **Funciones Principales**
   - **Agregar contacto**: Se pide al usuario el nombre, teléfono y email. El programa asigna un nuevo ID y registra el contacto en el array.
   - **Modificar contacto**: Se solicita el **ID** del contacto a cambiar. Luego, se permiten actualizar individualmente (o dejar sin cambios) el nombre, teléfono y email.
   - **Borrar contacto**: El usuario ingresa el ID del contacto para eliminarlo de la agenda (ajusta el array según sea necesario).
   - **Listar contactos**: Muestra todos los contactos en **formato tabular**, con columnas alineadas para ID, nombre, teléfono y email.
   - **Buscar contacto**: Pide un **término de búsqueda** y debe mostrar solo aquellos contactos cuyo nombre, teléfono o email contengan dicho término (ignorando

mayúsculas/minúsculas).

4. **Interfaz de Usuario**
    - El programa deberá funcionar mediante un menú textual en la consola:
        a. Agregar contacto
        b. Modificar contacto
        c. Borrar contacto
        d. Listar contactos
        e. Buscar contacto
        f. Salir
5. **Manejo de Archivos**
    - Al iniciar la aplicación, se deben leer los contactos almacenados en el archivo **`agenda.csv`** (si existe) y cargarlos en el array.
    - Antes de que finalice la ejecución (opción “Salir”), el programa debe guardar todos los contactos de vuelta en **`agenda.csv`** para que los cambios persistan.
6. **Consideraciones Especiales**
    - El listado de contactos debe mostrarse usando un **formato de columnas alineadas** para mejorar la lectura (por ejemplo, especificadores de formato en `Console.WriteLine`).
    - Para recorrer el array y acceder a cada contacto, utiliza únicamente bucles `for`.
    - El programa debe validar que no se agreguen contactos más allá del límite de tu array.
    - Al finalizar, se debe poder **agregar, modificar, borrar, listar y buscar** contactos de manera adecuada.
    - El programa continúa ofreciendo el menú en un bucle hasta que el usuario elija la opción “Salir”.

Objetivo
Consolidar el uso de:

- `struct` en C#
- Manejo de datos utilizando solamente `if`, `for` y asignaciones en arrays
- Generación de un menú en consola
- Formateo de salida para mostrar tablas
- Lectura y escritura de datos mediante un archivo CSV

Entrega

- Proporcionar el archivo .cs con todo el código de la aplicación.
- El programa debe poder compilarse y ejecutarse correctamente en la consola de C#.

```
===== AGENDA DE CONTACTOS =====
1) Agregar contacto
2) Modificar contacto
3) Borrar contacto
4) Listar contactos
5) Buscar contacto
0) Salir
Seleccione una opción: 1

=== Agregar Contacto ===
Nombre   : Juan Pérez
Teléfono : 123456
Email    : juan@example.com
Contacto agregado con ID = 1
Presione cualquier tecla para continuar...

===== AGENDA DE CONTACTOS =====
1) Agregar contacto
2) Modificar contacto
3) Borrar contacto
4) Listar contactos
5) Buscar contacto
0) Salir
Seleccione una opción: 1

=== Agregar Contacto ===
Nombre   : María López
Teléfono : 654321
Email    : mlopez@mail.com
Contacto agregado con ID = 2
Presione cualquier tecla para continuar...

===== AGENDA DE CONTACTOS =====
1) Agregar contacto
2) Modificar contacto
3) Borrar contacto
4) Listar contactos
5) Buscar contacto
0) Salir
Seleccione una opción: 4

=== Lista de Contactos ===
ID    NOMBRE               TELÉFONO       EMAIL
1     Juan Pérez           123456         juan@example.com
2     María López          654321         mlopez@mail.com
```

```
Presione cualquier tecla para continuar...

===== AGENDA DE CONTACTOS =====
1) Agregar contacto
2) Modificar contacto
3) Borrar contacto
4) Listar contactos
5) Buscar contacto
0) Salir
Seleccione una opción: 5

=== Buscar Contacto ===
Ingrese un término de búsqueda (nombre, teléfono o email): juan

Resultados de la búsqueda:
ID    NOMBRE               TELÉFONO       EMAIL
1     Juan Pérez           123456         juan@example.com

Presione cualquier tecla para continuar...

===== AGENDA DE CONTACTOS =====
1) Agregar contacto
2) Modificar contacto
3) Borrar contacto
4) Listar contactos
5) Buscar contacto
0) Salir
Seleccione una opción: 2

=== Modificar Contacto ===
Ingrese el ID del contacto a modificar: 1
Datos actuales => Nombre: Juan Pérez, Teléfono : 123456, Email: juan@example.com
(Deje el campo en blanco para no modificar)

Nombre    : Juan P. Domínguez
Teléfono  :
Email     :

Contacto modificado con éxito.
Presione cualquier tecla para continuar...

===== AGENDA DE CONTACTOS =====
1) Agregar contacto
2) Modificar contacto
3) Borrar contacto
4) Listar contactos
```

```
5) Buscar contacto
0) Salir
Seleccione una opción: 4

=== Lista de Contactos ===
ID    NOMBRE               TELÉFONO       EMAIL
1     Juan P. Domínguez    123456         juan@example.com
2     María López          654321         mlopez@mail.com

Presione cualquier tecla para continuar...

===== AGENDA DE CONTACTOS =====
1) Agregar contacto
2) Modificar contacto
3) Borrar contacto
4) Listar contactos
5) Buscar contacto
0) Salir
Seleccione una opción: 3

=== Borrar Contacto ===
Ingrese el ID del contacto a borrar: 2
Contacto con ID=2 eliminado con éxito.
Presione cualquier tecla para continuar...

===== AGENDA DE CONTACTOS =====
1) Agregar contacto
2) Modificar contacto
3) Borrar contacto
4) Listar contactos
5) Buscar contacto
0) Salir
Seleccione una opción: 4

=== Lista de Contactos ===
ID    NOMBRE               TELÉFONO       EMAIL
1     Juan P. Domínguez    123456         juan@example.com

Presione cualquier tecla para continuar...

===== AGENDA DE CONTACTOS =====
1) Agregar contacto
2) Modificar contacto
3) Borrar contacto
4) Listar contactos
5) Buscar contacto
```

```
0) Salir
Seleccione una opción: 6

Saliendo de la aplicación...
```

TP2: Sistema Bancario

> El pull request se debe llamar "TP2 - Legajo- NombreCopleto"

Objetivo:

Implementar un sistema bancario en C# utilizando programación orientada a objetos que permita gestionar clientes, cuentas y operaciones bancarias.

El sistema deberá registrar de forma individual cada operación (depósito, extracción, pago y transferencia) en clases separadas y mantener un historial global en el banco, así como un historial personal en cada cliente.

Requerimientos:

1. Clientes y Cuentas:
    - El banco debe poder gestionar múltiples clientes.
    - Cada cliente tiene un nombre y puede poseer varias cuentas.
    - Cada cuenta cuenta con un número (formato XXXXX) y un saldo.
    - El número de cuenta debe ser único.
    - Se deben definir tres tipos de cuentas:
        - Oro: Acumula un 5% de Puntos sobre los pagos realizados para montos mayores a 1000 y un 3% para montos menores.
        - Plata: Acumula un 2% de Puntos sobre los pagos realizados.
        - Bronce: Acumula un 1% de Puntos sobre los pagos realizados.
2. Operaciones Bancarias:
    - El sistema debe soportar las siguientes operaciones:
        - Depositar: Incrementa el saldo de la cuenta.
        - Extraer: Disminuye el saldo de la cuenta si hay fondos suficientes.
        - Pagar: Disminuye el saldo de la cuenta y acumula Puntos según el porcentaje definido para el tipo de cuenta.
        - Transferir: Permite mover fondos de una cuenta a otra.

- Cada operación se debe implementar en su propia clase derivada de una clase abstracta base llamada Operacion.
- Cada operación debe registrar el monto y las cuentas involucradas (para transferencias se deben registrar ambas: origen y destino).
- El banco debe verificar que la cuenta origen de la operacion es suya.

3. Registro y Reportes:
    - El banco debe mantener un registro global de todas las operaciones realizadas.
    - Además, cada cliente debe mantener un historial (lista) de las operaciones realizadas en cualquiera de sus cuentas.
    - Al finalizar las operaciones, el sistema deberá generar un informe completo que muestre:
        - El detalle global de las operaciones (tipo, monto y cuentas involucradas).
        - El estado final de cada cuenta (saldo y Puntos acumulado).
        - El historial individual de operaciones para cada cliente.

Clases:

El programa debe incluir las siguientes clases:

- `Banco` : Gestiona clientes, cuentas y operaciones.
- `Cliente` : Representa a un cliente con un nombre y una lista de cuentas.
- `Cuenta` (abstracta): Clase base para las cuentas bancarias.
    - `CuentaOro` : Implementa la acumulación de puntos específica para cuentas Oro.
    - `CuentaPlata` : Implementa la acumulación de puntos específica para cuentas Plata.
    - `CuentaBronce` : Implementa la acumulación de puntos específica para cuentas Bronce.
- `Operacion` (abstracta): Clase base para las operaciones bancarias.
    - `Deposito` : Representa una operación de depósito.
    - `Retiro` : Representa una operación de extracción.
    - `Pago` : Representa una operación de pago.
    - `Transferencia` : Representa una operación de transferencia entre cuentas.

Consideraciones Adicionales:

- Utiliza principios de programación orientada a objetos (herencia, encapsulamiento y polimorfismo).
- Asegúrate de manejar adecuadamente los posibles errores (por ejemplo, fondos insuficientes para extracción o pago).

Ejemplo de salida

Dato el siguiente ejemplo de uso:

```
// Definiciones

var raul = new Cliente("Raul Perez");
raul.Agregar(new CuentaOro("10001", 1000));
raul.Agregar(new CuentaPlata("10002", 2000));

var sara = new Cliente("Sara Lopez");
sara.Agregar(new CuentaPlata("10003", 3000));
sara.Agregar(new CuentaPlata("10004", 4000));

var luis = new Cliente("Luis Gomez");
luis.Agregar(new CuentaBronce("10005", 5000));

var nac = new Banco("Banco Nac");
nac.Agregar(raul);
nac.Agregar(sara);

var tup = new Banco("Banco TUP");
tup.Agregar(luis);


// Registrar Operaciones
nac.Registrar(new Deposito("10001", 100));
nac.Registrar(new Retiro("10002", 200));
nac.Registrar(new Transferencia("10001", "10002", 300));
nac.Registrar(new Transferencia("10003", "10004", 500));
nac.Registrar(new Pago("10002", 400));

tup.Registrar(new Deposito("10005", 100));
tup.Registrar(new Retiro("10005", 200));
tup.Registrar(new Transferencia("10005", "10002", 300));
tup.Registrar(new Pago("10005", 400));


// Informe final
nac.Informe();
tup.Informe();
```

Debe producir la siguiente salida

```
Banco: Banco Nac | Clientes: 2

  Cliente: Raul Perez | Saldo Total: $ 2.800,00 | Puntos Total: $ 8,00

    Cuenta: 10001 | Saldo: $ 800,00 | Puntos: $ 0,00
     -  Deposito $ 100,00 a [10001/Raul Perez]
     -  Transferencia $ 300,00 de [10001/Raul Perez] a [10002/Raul Perez]

    Cuenta: 10002 | Saldo: $ 2.000,00 | Puntos: $ 8,00
     -  Retiro $ 200,00 de [10002/Raul Perez]
     -  Transferencia $ 300,00 de [10001/Raul Perez] a [10002/Raul Perez]
     -  Pago $ 400,00 con [10002/Raul Perez]
     -  Transferencia $ 300,00 de [10005/Luis Gomez] a [10002/Raul Perez]

  Cliente: Sara Lopez | Saldo Total: $ 7.000,00 | Puntos Total: $ 0,00

    Cuenta: 10003 | Saldo: $ 2.500,00 | Puntos: $ 0,00
     -  Transferencia $ 500,00 de [10003/Sara Lopez] a [10004/Sara Lopez]

    Cuenta: 10004 | Saldo: $ 4.500,00 | Puntos: $ 0,00
     -  Transferencia $ 500,00 de [10003/Sara Lopez] a [10004/Sara Lopez]


Banco : Banco TUP | Clientes: 1

  Cliente: Luis Gomez | Saldo Total: $ 4.200,00 | Puntos Total: $ 4,00

    Cuenta: 10005 | Saldo: $ 4.200,00 | Puntos: $ 4,00
     -  Deposito $ 100,00 a [10005/Luis Gomez]
     -  Retiro $ 200,00 de [10005/Luis Gomez]
     -  Transferencia $ 300,00 de [10005/Luis Gomez] a [10002/Raul Perez]
     -  Pago $ 400,00 con [10005/Luis Gomez]
```

Entrega:

Desarrolla y documenta el código en C#. El programa debe compilar y ejecutarse mostrando en consola el informe global de operaciones y el historial individual de cada cliente, junto con el estado final de sus cuentas.

TP3: Crear Lista Ordenada Genérica

> El pull request se debe llamar "TP3 - Legajo - NombreCopleto"

Objetivo

Aprender a desarrollar estructuras de datos reutilizables.

Tarea

Se debe crear una clase `ListaOrdenada` que permita agregar un elemento, verificar si contiene un elemento y eliminar un elemento.

Además, se deben poder leer los elementos según su posición ([]) y determinar cuántos elementos componen la lista.

Por último, se debe poder filtrar todos los elementos que cumplan una condición.

Funciones:

- `Contiene(elemento)` : Indica si el elemento existe en la lista
- `Agregar(elemento)` : Agrega un elemento manteniendo la lista ordenada (si esta repetido ignorar)
- `Eliminar(elemento)` : Elimina un elmento de la lista (si no existe ignorar)
- `Cantidad` : Indica la cantidad de elementos que hay en la lista.
- `Lista[indice]` : Me retornar el elemento que esta en la posicion indicada por el indice.
- `Filtrar(condicion)` : Me da una nueva lista ordenada con todos los elemento que cumpla la condicion.

Ademas debe crear la clase `Contacto` con `Nombre` y `Telefono` que debe mantenerse ordenanda alfabeticamente.

Presentación

El trabajo consiste en implementar las clases `ListaOrdenada` y `Contacto` para que se ejecuten todas las pruebas sin errores.

No se debe crear ninguna interfaz de usuario; únicamente se deberán pasar los test exitosamente.

Solo debe modificar el archivo `ejercicio.cs` de la carpeta `tp3` y solo debe subir dicho archivo.