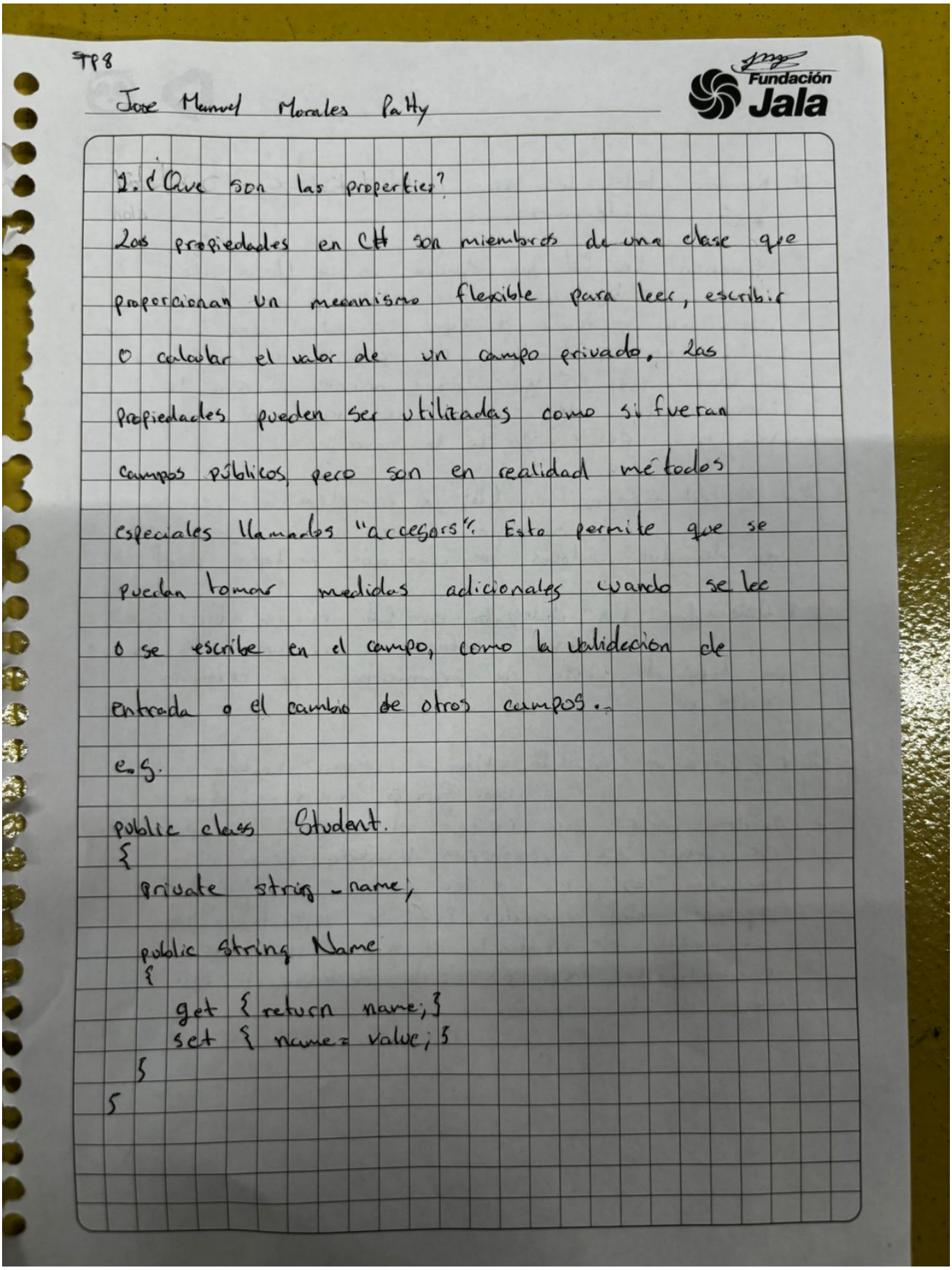
# TP - 8 - C Sharp | Manuel Morales

## Que son los “properties”?




## Implementar una estructura de datos que represente una coleccion de items (Génerico)

- Implementar `IEnumerator<T>`

- Implementar `IList<T>`
- Implementar `IEnumerable<T>`
- No utilizer ningun tipo de EDD como campo de su implementacion
- A base de nodos, ejemplo

## Nodo

```
namespace DataStructure
{
    public class Node<T>
    {
        public T Value { get; set; }
        public Node<T>? Next { get; set; }

        public Node(T value)
        {
            Value = value;
            Next = null;
        }
    }
}
```

## Linked List

```
using System.Collections;

namespace DataStructure
{
    public class LinkedList<T> : IEnumerator<T>, IEnumerable<T>, IList<T>
    {
        private Node<T>? _head;
        private Node<T>? _current;

        public T Current
        {
            get
            {
                if (_current == null)
                {
                    throw new IndexOutOfRangeException();
                }

                return _current.Value;
            }
        }

        object IEnumerator.Current
        {
            get
```

```
            {
                if (Current == null)
                {
                    throw new IndexOutOfRangeException();
                }

                return Current;
            }
        }

        public bool MoveNext()
        {
            if (_current == null)
            {
                _current = _head;
            }
            else
            {
                _current = _current.Next;
            }

            return _current != null;
        }

        public void Reset()
        {
            _current = null;
        }

        public void Dispose() { }

        public IEnumerator<T> GetEnumerator()
        {
            return this;
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }

        private Node<T>? GetNodeByIndex(int index)
        {
            int currentIndex = 0;
            Node<T>? currentNode = _head;

            while (currentNode != null)
            {
                if (currentIndex == index)
                {
                    return currentNode;
                }

                currentIndex++;
```

```
                currentNode = currentNode?.Next;
            }

            return null;
        }

        public T this[int index]
        {
            get
            {
                Node<T>? node = GetNodeByIndex(index);

                if (node == null)
                {
                    throw new IndexOutOfRangeException();
                }

                return node.Value;
            }
            set
            {
                Node<T>? node = GetNodeByIndex(index);

                if (node == null)
                {
                    throw new IndexOutOfRangeException();
                }

                node.Value = value;
            }
        }

        public int Count
        {
            get
            {
                int count = 0;
                Node<T>? currentNode = _head;

                while (currentNode != null)
                {
                    count++;
                    currentNode = currentNode.Next;
                }

                return count;
            }
        }

        public bool IsReadOnly => false;

        public void Add(T item)
        {
            if (_head == null)
```

```
            {
                _head = new Node<T>(item);
                return;
            }

            Node<T>? currentNode = _head;
            while (currentNode.Next != null)
            {
                currentNode = currentNode.Next;
            }

            currentNode.Next = new Node<T>(item);
        }

        public void Clear()
        {
            _head = null;
        }

        public bool Contains(T item)
        {
            Node<T>? currentNode = _head;

            while (currentNode != null)
            {
                if (EqualityComparer<T>.Default.Equals(currentNode.Value,
item))
                {
                    return true;
                }

                currentNode = currentNode.Next;
            }

            return false;
        }

        public void CopyTo(T[] array, int arrayIndex)
        {
            Node<T>? currentNode = _head;

            if (arrayIndex < 0 || arrayIndex >= array.Length)
            {
                throw new IndexOutOfRangeException();
            }

            while (currentNode != null)
            {
                array[arrayIndex++] = currentNode.Value;
                currentNode = currentNode.Next;
            }
        }

        public int IndexOf(T item)
```

```
        {
            int index = 0;
            Node<T>? currentNode = _head;

            while (currentNode != null)
            {
                if (EqualityComparer<T>.Default.Equals(currentNode.Value,
item))
                {
                    return index;
                }

                index++;
                currentNode = currentNode.Next;
            }

            return -1;
        }

        public void Insert(int index, T item)
        {
            if (index == 0)
            {
                Node<T> newHead = new Node<T>(item) { Next = _head };
                _head = newHead;
                return;
            }

            Node<T>? currentNode = GetNodeByIndex(index - 1);

            if (currentNode == null)
            {
                throw new IndexOutOfRangeException();
            }

            Node<T> newNode = new Node<T>(item) { Next = currentNode.Next
};

            currentNode.Next = newNode;
        }

        public bool Remove(T item)
        {
            if (_head == null)
            {
                return false;
            }

            if (EqualityComparer<T>.Default.Equals(_head.Value, item))
            {
                _head = _head.Next;
                return true;
            }
```

```
            Node<T>? currentNode = _head;

            while (currentNode.Next != null)
            {
                if
(EqualityComparer<T>.Default.Equals(currentNode.Next.Value, item))
                {
                    currentNode.Next = currentNode.Next.Next;
                    return true;
                }

                currentNode = currentNode.Next;
            }

            return false;
        }

        public void RemoveAt(int index)
        {
            if (index == 0)
            {
                if (_head == null)
                {
                    throw new IndexOutOfRangeException();
                }

                _head = _head.Next;
                return;
            }

            Node<T>? currentNode = GetNodeByIndex(index - 1);

            if (currentNode == null || currentNode.Next == null)
            {
                throw new IndexOutOfRangeException();
            }

            currentNode.Next = currentNode.Next.Next;
        }
    }
}
```

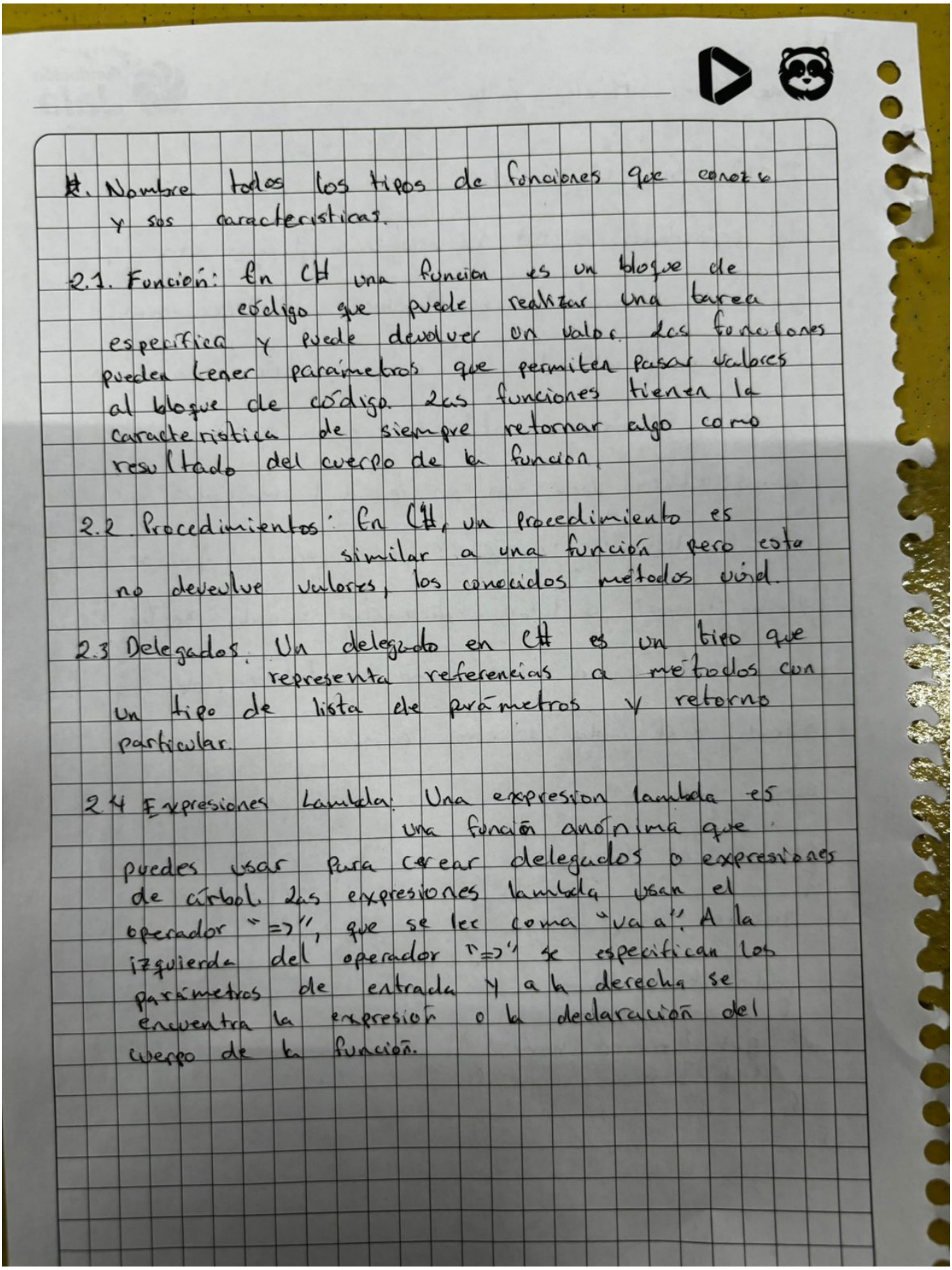
## Tests

```
using NUnit.Framework;

namespace DataStructure
{
    [TestFixture]
    public class LinkedListTests
    {
```

```
        [Test]
        public void Test_Add()
        {
            var list = new LinkedList<int>();
            list.Add(1);
            list.Add(2);
            list.Add(3);
            Assert.That(3, Is.EqualTo(list.Count));
            Assert.That(1, Is.EqualTo(list[0]));
            Assert.That(2, Is.EqualTo(list[1]));
            Assert.That(3, Is.EqualTo(list[2]));
        }

        [Test]
        public void Test_Clear()
        {
            var list = new LinkedList<int>();
            list.Add(1);
            list.Add(2);
            list.Clear();
            Assert.That(0, Is.EqualTo(list.Count));
        }

        [Test]
        public void Test_Contains()
        {
            var list = new LinkedList<int>();
            list.Add(1);
            list.Add(2);
            Assert.That(list.Contains(1), Is.True);
            Assert.That(list.Contains(3), Is.False);
        }

        [Test]
        public void Test_CopyTo()
        {
            var list = new LinkedList<int>();
            list.Add(1);
            list.Add(2);
            var array = new int[2];
            list.CopyTo(array, 0);
            Assert.That(1, Is.EqualTo(array[0]));
            Assert.That(2, Is.EqualTo(array[1]));
        }

        [Test]
        public void Test_IndexOf()
        {
            var list = new LinkedList<int>();
            list.Add(1);
            list.Add(2);
            Assert.That(0, Is.EqualTo(list.IndexOf(1)));
            Assert.That(1, Is.EqualTo(list.IndexOf(2)));
            Assert.That(-1, Is.EqualTo(list.IndexOf(3)));
```

```
        }

        [Test]
        public void Test_Insert()
        {
            var list = new LinkedList<int>();
            list.Add(1);
            list.Add(3);
            list.Insert(1, 2);
            Assert.That(1, Is.EqualTo(list[0]));
            Assert.That(2, Is.EqualTo(list[1]));
            Assert.That(3, Is.EqualTo(list[2]));
        }

        [Test]
        public void Test_Remove()
        {
            var list = new LinkedList<int>();
            list.Add(1);
            list.Add(2);
            list.Remove(1);
            Assert.That(1, Is.EqualTo(list.Count));
            Assert.That(2, Is.EqualTo(list[0]));
        }

        [Test]
        public void Test_RemoveAt()
        {
            var list = new LinkedList<int>();
            list.Add(1);
            list.Add(2);
            list.RemoveAt(0);
            Assert.That(1, Is.EqualTo(list.Count));
            Assert.That(2, Is.EqualTo(list[0]));
        }

        [Test]
        public void Test_Iterator()
        {
            var list = new LinkedList<int>();
            list.Add(1);
            list.Add(2);
            list.Add(3);
            Assert.That(new[] { 1, 2, 3 }, Is.EqualTo(list.ToArray()));
        }
    }
}
```

Nombre todos los tipos de funciones que conozca y sus caracterisitcas



1. Nombre todos los tipos de funciones que conozca y sus características.

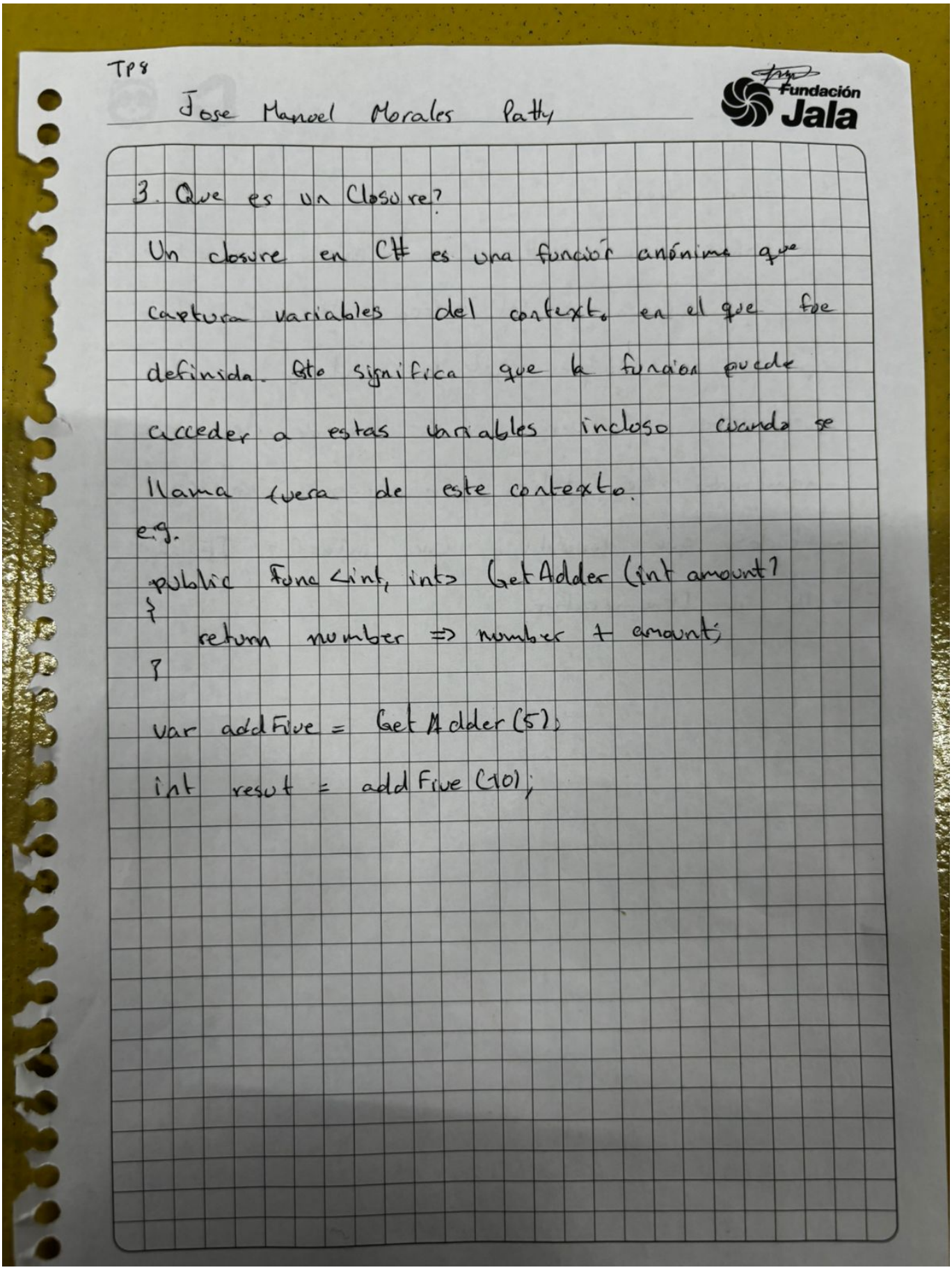
2.1. Función: En C# una función es un bloque de código que puede realizar una tarea específica y puede devolver un valor. Las funciones pueden tener parámetros que permiten pasar valores al bloque de código. Las funciones tienen la característica de siempre retornar algo como resultado del cuerpo de la función.

2.2 Procedimientos: En C#, un procedimiento es similar a una función pero esta no devuelve valores, los conocidos métodos void.

2.3 Delegados: Un delegado en C# es un tipo que representa referencias a métodos con un tipo de lista de parámetros y retorno particular.

2.4 Expresiones Lambda: Una expresión lambda es una función anónima que puedes usar para crear delegados o expresiones de árbol. Las expresiones lambda usan el operador "=>", que se lee como "va a". A la izquierda del operador "=>" se especifican los parámetros de entrada y a la derecha se encuentra la expresión o la declaración del cuerpo de la función.

Que es un Closure?




## Extention Functions de las funciones: Filter, Map, Reduce

### Extesion Methods

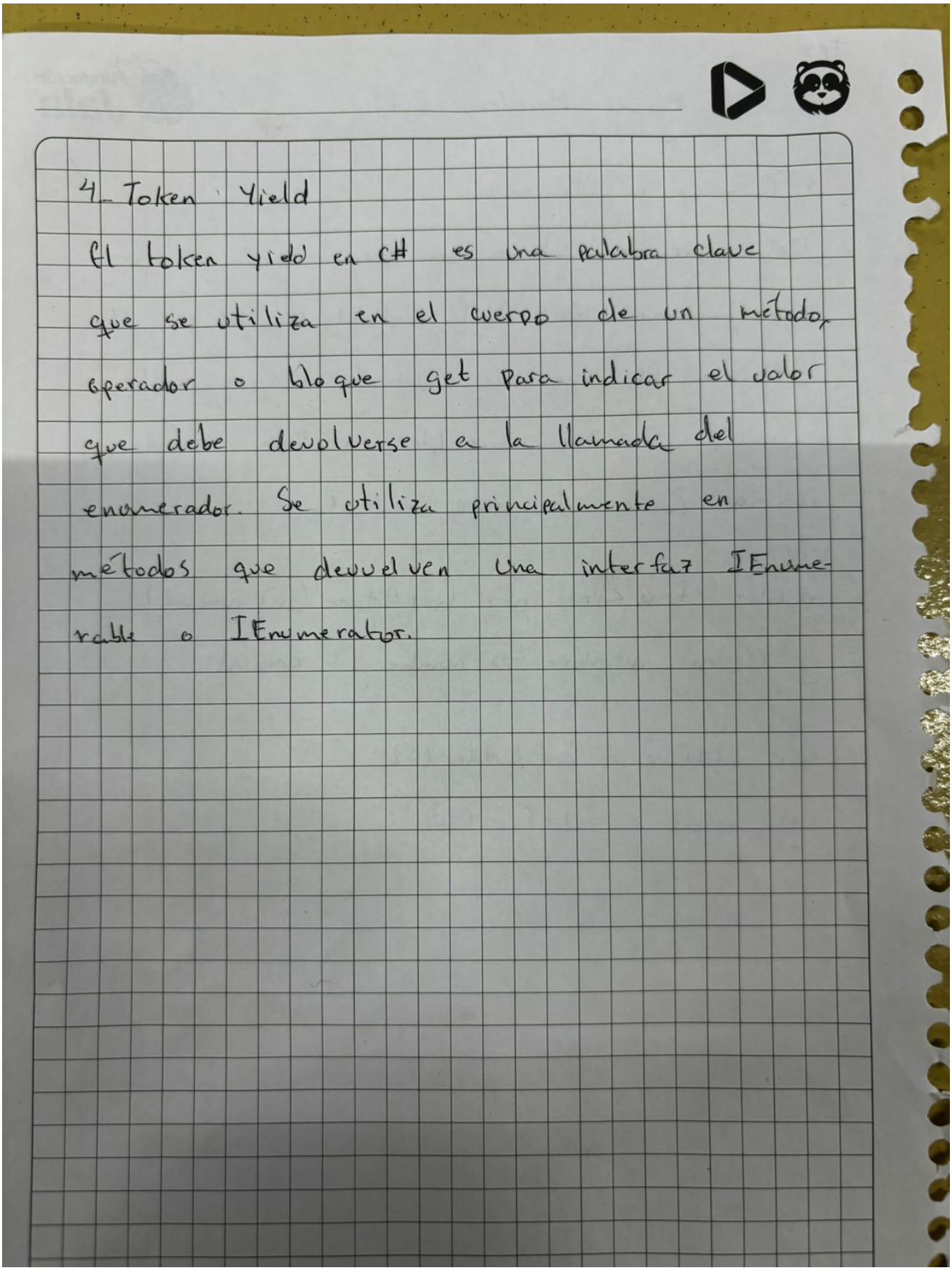
```
public static class ExtensionMethods
{
    public static IEnumerable<T> Filter<T>(this IEnumerable<T> collection,
Func<T, bool> predicate)
    {
        foreach (var item in collection)
        {
            if (predicate(item))
            {
                yield return item;
            }
        }
    }

    public static IEnumerable<TResult> Map<T, TResult>(
        this IEnumerable<T> collection,
        Func<T, TResult> selector
    )
    {
        foreach (var item in collection)
        {
            yield return selector(item);
        }
    }

    public static TResult Reduce<T, TResult>(
        this IEnumerable<T> collection,
        Func<TResult, T, TResult> reducer,
        TResult initial
    )
    {
        var result = initial;

        foreach (var item in collection)
        {
            result = reducer(result, item);
        }

        return result;
    }
}
```

## Program

```
class Program
{
    static void Main(string[] args)
    {
        var numbers = new List<int> { 1, 2, 3, 4, 5 };
```

```
        var evenNumbers = numbers.Filter(n => n % 2 == 0);
        Console.WriteLine("Even numbers: " + string.Join(", ",
evenNumbers));

        var squares = numbers.Map(n => n * n);
        Console.WriteLine("Squares: " + string.Join(", ", squares));

        var sum = numbers.Reduce((a, b) => a + b, 0);
        Console.WriteLine("Sum: " + sum);
    }
}
```

## Investigar el uso del token yield



4- Token Yield

El token yield en C# es una palabra clave que se utiliza en el cuerpo de un método, operador o bloque get para indicar el valor que debe devolverse a la llamada del enumerador. Se utiliza principalmente en métodos que devuelven una interfaz IEnumerable o IEnumerator.