



Programación y Laboratorio II

Clase 07

Encapsulamiento

Prof. Mauricio Cerizza

Encapsulamiento

- ¿Qué es el encapsulamiento?
- Ocultamiento de la información
- Modificadores de acceso
- Abstracción y encapsulamiento
- Buenas prácticas
- Encapsulamiento en la Base Class Library

Propiedades

- ¿Qué es una propiedad?
- Descriptores de acceso
- Asignación y lectura

Enumerados

- ¿Qué es un enumerado?
- Declaración y uso

Indexadores

- ¿Qué es un indexador?
- Asignación y lectura
- Otras características



01.

Encapsulamiento

¿Qué es el ENCAPSULAMIENTO?

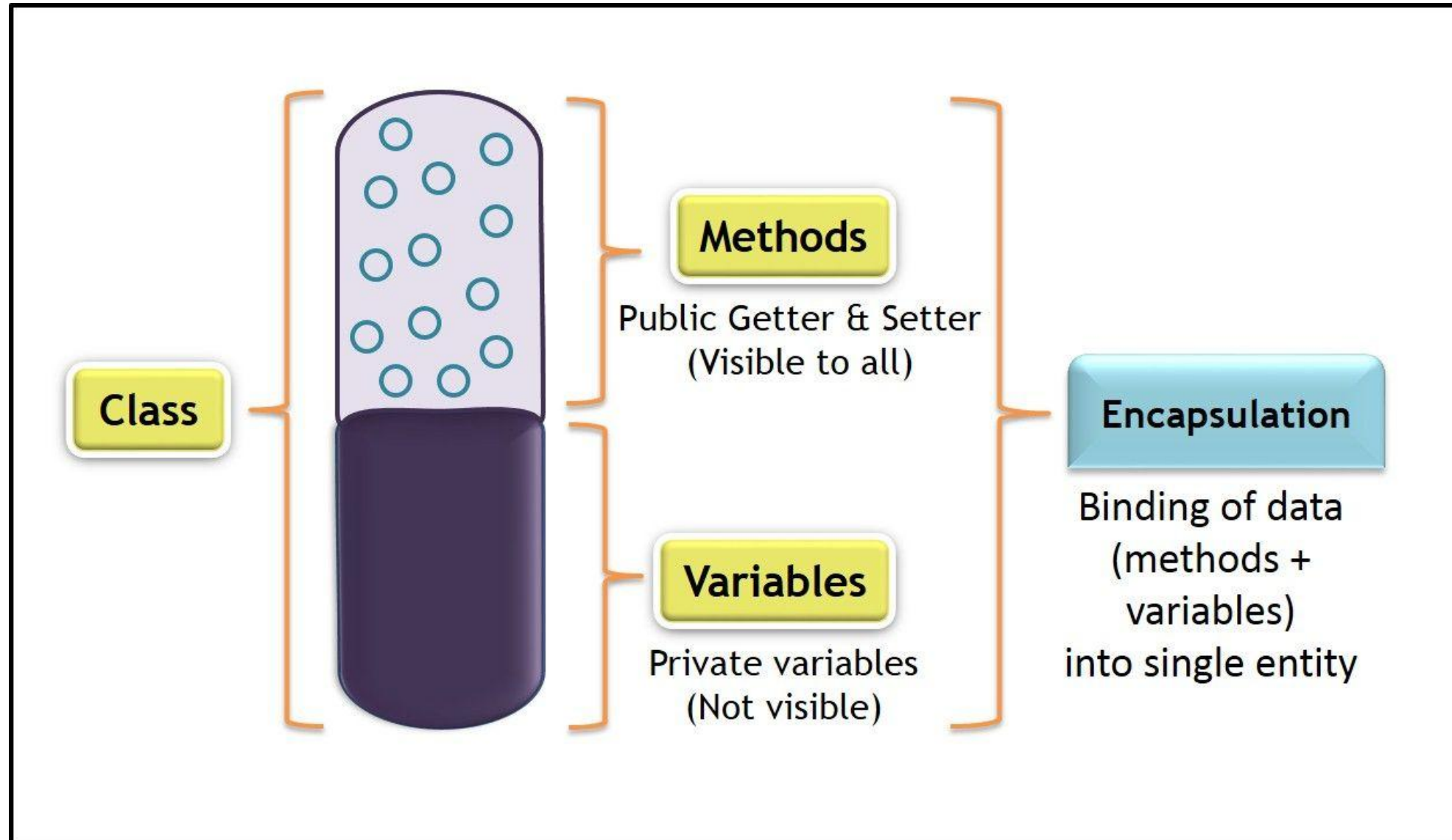
El **encapsulamiento** (*encapsulation*) consiste en agrupar los datos del objeto junto con los métodos que operan sobre esos datos.

Nos permite **ocultar los detalles de la implementación** y **proteger el estado del objeto**.

*El encapsulamiento nos ayuda a **manejar la complejidad** de la solución al restringir el acceso de los usuarios de la clase a dicha complejidad.*

Encapsulation en inglés significa encerrar algo en una cápsula o como una cápsula.

Encapsulamiento

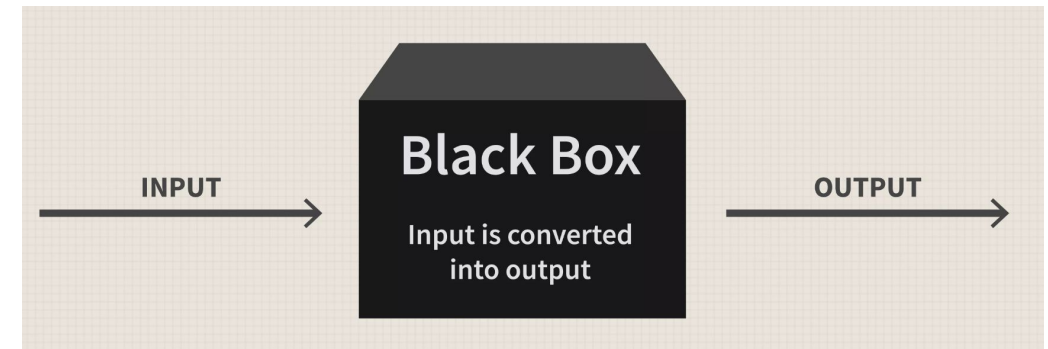


Ocultamiento de la información

No se especifica si el contenedor es “transparente”, “translúcido” u “opaco”.

Que algo esté encapsulado no significa que esté oculto.

Pero que esté oculto sí implica cierto nivel de encapsulamiento.



Ocultamiento de la información

Abstracción

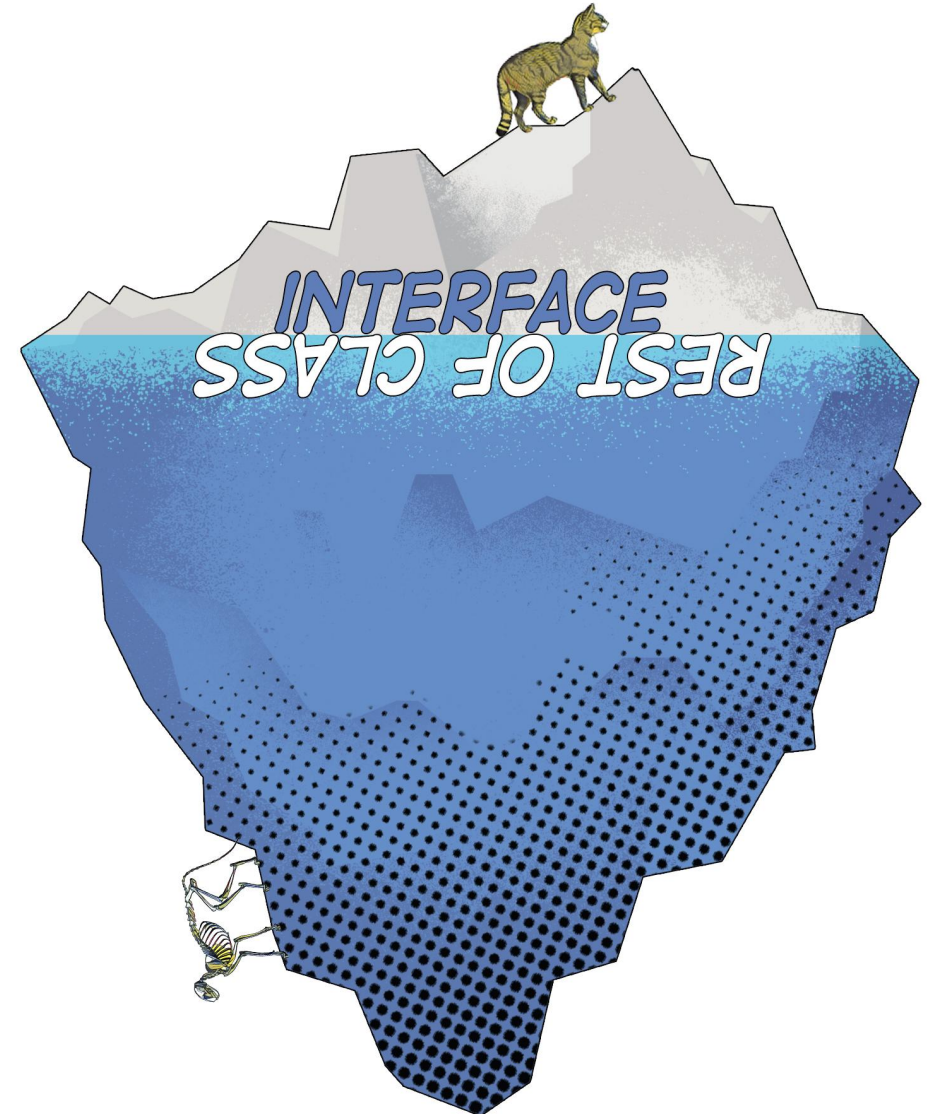
Determino qué información del objeto es relevante y cuál no. Identifico sus datos y operaciones.

Encapsulamiento

Agrupo esos datos y operaciones en un contenedor (una clase). Los aísló del exterior.

Ocultamiento de la información

Estas clases permiten restringir el acceso a sus miembros a través de **modificadores de acceso / visibilidad**.



Modificadores de acceso

Modificadores de acceso	
Modificador	Descripción
<i>public</i>	Accesible desde cualquier parte.
<i>private</i>	Sólo accesible desde dentro del mismo tipo.
<i>internal</i>	Sólo accesible desde dentro del mismo ensamblado / proyecto.
<i>protected</i>	Sólo accesible desde dentro del mismo tipo o tipos derivados (hijos).

Abstracción y encapsulamiento



Aplicando **abstracción** a la clase **Cafetera**

Responsabilidad:

Preparar una infusión de café a partir de un suministro de agua y granos de café molidos.

Interfaz de la clase / operaciones públicas:

- Cargar agua
- Suministrar café
- Encender
- Apagar
- Consultar nivel de agua

¿Dónde está el **encapsulamiento**?

El material plástico que recubre la cafetera nos oculta la complejidad y detalles del mecanismo que resuelve la preparación del café.

- No sabemos cómo transforma la energía eléctrica en calor.
- No sabemos con exactitud a qué temperatura calienta el agua.
- No sabemos cómo está armado el circuito de agua.
- No sabemos a qué presión impacta el agua en el café.
- Y por sobre todo, no podemos realizar ninguna modificación sobre todo lo antes nombrado.

Buenas prácticas

Minimizar la accesibilidad de clases y sus miembros

Por defecto, preferir el nivel de visibilidad más estricto. Hacerse la pregunta: ¿qué preserva mejor la integridad de la abstracción?

No exponer el estado del objeto

Se deben exponer, y sólo si es necesario, a través de métodos de acceso.

Ocultar los detalles de la implementación

A los usuarios de la clase no deben saber ni interesarse en cómo está implementada la solución, sólo qué hace.

No asuman cómo será utilizada la clase

El objeto debe funcionar correctamente bajo cualquier uso posible y no permitir usos indebidos.

Steve McConnell



En 1998, McConnell fue nombrado como una de las tres personas más influyentes en la industria por la revista Software Development Magazine, junto con Bill Gates y Linus Torvalds.

Buenas prácticas

Evitar usar clases internas

Salvo casos puntuales, las clases internas rompen el encapsulamiento.

Favorecer el tiempo de lectura del código sobre el de escritura

Nos podemos ver tentados de romper el encapsulamiento para alcanzar una solución más fácil, pero la realidad es que se pasa mucho más tiempo leyendo código que escribiéndolo.

Tener cuidado de no romper la semántica del encapsulamiento

La interfaz de la clase debe ser clara y no estar ligada a la implementación interna.

Mantener un bajo nivel de acoplamiento

El acoplamiento se refiere a la conexión y dependencia entre dos clases. Cuanto más bajo sea mejor.

Steve McConnell



En 1998, McConnell fue nombrado como una de las tres personas más influyentes en la industria por la revista Software Development Magazine, junto con Bill Gates y Linus Torvalds.

Encapsulamiento en la Base Class Library

```
1 public class List<T> : ICollection<T>, IEnumerable<T>, IEnumerable,
2 IList<T>, IReadOnlyCollection<T>, IReadOnlyList<T>, ICollection, IList
3 {
4     public List();
5     public List(IEnumerable<T> collection);
6     public List(int capacity);
7
8     public T this[int index] { get; set; }
9
10    public int Count { get; }
11    public int Capacity { get; set; }
12
13    public void Add(T item);
14    public void AddRange(IEnumerable<T> collection);
15    public void Clear();
16    public bool Contains(T item);
17    public int IndexOf(T item, int index, int count);
18    public int IndexOf(T item, int index);
19    public int IndexOf(T item);
20    public bool Remove(T item);
21    public void RemoveAt(int index);
22    public void Sort(Comparison<T> comparison);
23    public void Sort(int index, int count, IComparer<T>? comparer);
24    public void Sort();
25    public void Sort(IComparer<T>? comparer);
26    public T[] ToArray();
27 }
```

```
1 /*
2  * La clase List implementa una colección de tamaño variable que usa un
3  * array de objetos para almacenar los elementos.
4  * Una lista tiene capacidad, que es el tamaño asignado al array interno.
5  * A medida que nuevos elementos se agregan a la lista, la capacidad de la lista
6  * es automáticamente incrementada a demanda rearmando el array interno.
7  */
8 public class List<T> : IList<T>, System.Collections.IList, IReadOnlyList<T>
9 {
10     // Capacidad por defecto
11     private const int _defaultCapacity = 4;
12     // La lista internamente guarda los elementos en un array.
13     private T[] _items;
14     // Tamaño actual de la lista.
15     private int _size;
16     private int _version;
17
18     /*
19     * Agrega el objeto pasado como argumento al final de la lista.
20     * El tamaño de la lista es incrementado en uno. Si se requiere, la capacidad de la lista
21     * se duplica antes de agregar el nuevo elemento.
22     */
23     public void Add(T item) {
24         if (_size == _items.Length) EnsureCapacity(_size + 1); // Establece la nueva capacidad.
25         // Incrementa el valor del atributo size
26         // y almacena en ese índice del array el nuevo elemento.
27         _items[_size++] = item;
28         _version++;
29     }
30 }
```

Encapsulamiento en la Base Class Library

```
1  /*
2  * El método EnsureCapacity está oculto (privado) y es utilizado internamente para
3  * asegurar que la capacidad de la lista es al menos el mínimo valor requerido.
4  * Si la capacidad de la lista es menor al mínimo, la capacidad es incrementada al doble
5  * de la capacidad actual o al mínimo.
6  */
7  private void EnsureCapacity(int min) { // el argumento min representa a la mínima capacidad requerida.
8      if (_items.Length < min) { // Si el tamaño del array es menor al mínimo requerido...
9          // Calcula la nueva capacidad
10         int newCapacity = _items.Length == 0? _defaultCapacity : _items.Length * 2;
11
12         // Se asegura que no supere la capacidad máxima de un array.
13         if ((uint)newCapacity > Array.MaxLength) newCapacity = Array.MaxLength;
14
15         // Si la nueva capacidad es menor al mínimo, se establece con el valor del mínimo.
16         if (newCapacity < min) newCapacity = min;
17
18         // Establece la nueva capacidad.
19         Capacity = newCapacity;
20     }
21 }
```

```
1  /*
2  * Obtiene o modifica la capacidad de la lista.
3  * La capacidad es el tamaño del array interno utilizado
4  * para almacenar los elementos de la lista.
5  * Cuando la propiedad es asignada, el array interno es reasignado
6  * con la capacidad dada.
7  */
8  public int Capacity {
9      get {
10         Contract.Ensures(Contract.Result<int>() >= 0);
11         return _items.Length;
12     }
13     set {
14         if (value < _size) {
15             ThrowHelper.ThrowArgumentOutOfRangeException(ExceptionArgument.value,
16                 ExceptionResource.ArgumentOutOfRange_SmallCapacity);
17         }
18         Contract.EndContractBlock();
19
20         if (value != _items.Length) {
21             if (value > 0) {
22                 T[] newItems = new T[value];
23                 if (_size > 0) {
24                     Array.Copy(_items, 0, newItems, 0, _size);
25                 }
26                 _items = newItems;
27             }
28             else {
29                 _items = _emptyArray;
30             }
31         }
32     }
33 }
```




02.

Propiedades

¿Qué son las PROPIEDADES?

Una **propiedad** es un miembro que proporciona un mecanismo flexible para leer, escribir o calcular el valor de un atributo.

No son más que otra forma de escribir métodos de acceso (getters y setters).

Nos ayudan a aplicar el encapsulamiento:

- *Habilitan una forma segura de obtener y modificar el estado de un objeto.*
- *Aportan al ocultamiento de los detalles de la implementación.*

Descriptores de acceso

```
1 private int totalGoles;
2
3 public int GetTotalGoles()
4 {
5     return totalGoles;
6 }
7
8 public void SetTotalGoles(int value)
9 {
10     totalGoles = value;
11 }
```

```
1 private int totalGoles;
2
3 public int TotalGoles
4 {
5     get
6     {
7         return totalGoles;
8     }
9     set
10    {
11        this.totalGoles = value;
12    }
13 }
```


Descriptores de acceso

```
1 private int totalGoles;
2 private int totalPartidos;
3
4 public float PromedioGoles
5 {
6     get
7     {
8         float prom = totalGoles / (float)totalPartidos;
9         return prom;
10    }
11 }
12
13 public int TotalPartidos
14 {
15     set
16     {
17         totalPartidos = value;
18     }
19 }
```

SÓLO LECTURA


SÓLO ESCRITURA

Asignación y lectura

```
1 public class Torneo
2 {
3     private int totalGoles;
4     private int totalPartidos;
5
6     public Torneo(int totalGoles, int totalPartidos)
7     {
8         this.totalGoles = totalGoles;
9         this.totalPartidos = totalPartidos;
10    }
11
12    public float PromedioGoles
13    {
14        get
15        {
16            float prom = totalGoles / (float)totalPartidos;
17            return prom;
18        }
19    }
20
21    public int TotalPartidos
22    {
23        set
24        {
25            totalPartidos = value;
26        }
27    }
28 }
```

```
1 public static void Main(string[] args)
2 {
3     Torneo torneo = new Torneo(10, 5);
4
5     float promedioGoles = torneo.PromedioGoles;
6     Console.WriteLine($"El promedio de goles es de: {promedioGoles}");
7
8     torneo.TotalPartidos = 6;
9     Console.WriteLine($"El promedio de goles es de: {torneo.PromedioGoles}");
10 }
```

```
El promedio de goles es de: 2
El promedio de goles es de: 0,5
```

- 
- 1 —
 - 2 —
 - 3 —

03.

Enumerados

¿Qué es un **ENUMERADO**?

Un **enumerado** es un **tipo de dato** que representa un conjunto de constantes numéricas con nombre.

Sirven para definir y limitar el conjunto de valores que pueden ser asignados a una variable o parámetro de entrada.

Sus valores son números enteros y tienen base-cero.

Declaración y uso

```
1 public enum DiasDeLaSemana
2 {
3     Domingo,
4     Lunes,
5     Martes,
6     Miercoles,
7     Jueves,
8     Viernes,
9     Sabado
10 }
```

```
1 DiasDeLaSemana dia = DiasDeLaSemana.Lunes;
2 int i = (int) DiasDeLaSemana.Lunes;
3
4 Console.WriteLine(dia); // Mostrará "Lunes"
5 Console.WriteLine(i);   // Mostrará 1
```

Pueden ser declarados dentro de una clase (tipo anidado) o dentro de un espacio de nombres.

Declaración y uso

```
1 public enum DiasDeLaSemana
2 {
3     Domingo,
4     Lunes = 10,
5     Martes,
6     Miercoles,
7     Jueves,
8     Viernes,
9     Sabado
10 }
```

```
1 Console.WriteLine((int) DiasDeLaSemana.Lunes);
2 // Mostrará 10
3 Console.WriteLine((int) DiasDeLaSemana.Miercoles);
4 // Mostrará 12
```

A un enumerado se le podrá asignar un valor entero cambiando la numeración por defecto.

Ejercicios



- Ejercicio I01 - Puesto de atención

https://codeutnfra.github.io/programacion_2_laboratorio_2_apuntes/



04.

Indexadores

¿Qué es un **INDEXADOR**?

Indexar es ordenar una serie de datos o información de acuerdo a un criterio común a todos ellos, para facilitar su consulta y análisis a través de un índice.

Los **indexadores** nos permiten acceder a los datos del objeto a través de un índice.

La forma de acceder y el orden estará determinada por el criterio de indexación que se haya elegido.

Asignación y uso

```
1 public class RegistroTemperaturas
2 {
3     float[] tempeturas = new float[10]
4     {
5         56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
6         61.3F, 65.9F, 62.1F, 59.2F, 57.5F
7     };
8
9     public float this[int index]
10    {
11        get
12        {
13            return tempeturas[index];
14        }
15        set
16        {
17            tempeturas[index] = value;
18        }
19    }
20 }
```

```
1 static void Main(string[] args)
2 {
3     RegistroTemperaturas registroTemp = new RegistroTemperaturas();
4     const int indice = 2;
5
6     float temperatura = registroTemp[indice];
7
8     Console.WriteLine($"Indice {indice} - Temperatura: {temperatura}");
9
10    registroTemp[indice] = 45.3F;
11
12    Console.WriteLine($"Indice {indice} - Temperatura: {registroTemp[indice]}");
13 }
```

```
Indice 2 - Temperatura: 56,5
Indice 2 - Temperatura: 45,3
```

Asignación y uso

DICTIONARY<TKey, TValue>

```
1 public TValue this[TKey key] {
2     get {
3         int i = FindEntry(key);
4         if (i >= 0) return entries[i].value;
5         ThrowHelper.ThrowKeyNotFoundException();
6         return default(TValue);
7     }
8     set {
9         Insert(key, value, false);
10    }
11 }
```

LIST<T>

```
1 public T this[int index] {
2     get {
3         if ((uint) index >= (uint)_size) {
4             ThrowHelper.ThrowArgumentOutOfRangeException();
5         }
6         Contract.EndContractBlock();
7         return _items[index];
8     }
9
10    set {
11        if ((uint) index >= (uint)_size) {
12            ThrowHelper.ThrowArgumentOutOfRangeException();
13        }
14        Contract.EndContractBlock();
15        _items[index] = value;
16        _version++;
17    }
18 }
```

Otras características

Pueden trabajar con más de un índice

Como vimos en las matrices.

Se puede indexar por cualquier tipo

Como sucedió cuando trabajamos con diccionarios, el índice no tiene que ser necesariamente numérico, puede ser de cualquier tipo.

Se pueden sobrecargar

Cambiando orden, número o tipo de los índices.

No se pueden declarar indexadores estáticos

Los indexadores trabajan y se los invoca a través de la referencia a una instancia específica. Por ello llevan la palabra “this” en su declaración

Ejercicios



- Ejercicio I02 - ¿Consultaste el índice?

https://codeutnfra.github.io/programacion_2_laboratorio_2_apuntes/