Fundamentos de Listas en C#

1. ¿Qué es una lista?

En C#, una lista es una colección dinámica de elementos que puede crecer o reducir su tamaño en tiempo de ejecución.

La clase que se usa para listas genéricas es List<T>, donde T representa el tipo de dato que contendrá (por ejemplo, List<string>, List<int>, List<Fruit>). Fruit puede ser una clase definida por el programador.

Características principales:

- Se encuentra en el namespace System.Collections.Generic.
- Permite índices (acceso como un arreglo: miLista[0]).
- Puede almacenar tipos primitivos u objetos personalizados.
- Puede agregar, eliminar, insertar y buscar elementos fácilmente.

2. Tipos de colecciones en C#

Antes de trabajar con List<T> es importante conocer que existen varios tipos de colecciones:

- **Arreglos (Array)** → tamaño fijo, acceso rápido por índice.
- **List** → tamaño dinámico, más flexible que los arreglos.
- **Dictionary<TKey, TValue>** → pares clave-valor.
- **HashSet** → colección de elementos únicos.
- **Queue** y **Stack** → estructuras FIFO y LIFO.

Para empezar con colecciones, se recomienda comenzar por List<T> porque es la más versátil.

3. Creación de una lista

Antes de usar listas, se debe incluir el espacio de nombres:

```
using System.Collections.Generic;
```

Formas de crear una lista:

```
// Lista vacía
List<string> nombres = new List<string>();

// Lista con datos iniciales
List<string> frutas = new List<string> { "Manzana", "Pera", "Mango" };

// Lista de objetos personalizados
List<Fruit> frutasPersonalizadas = new List<Fruit>
```

```
{
    new Fruit { Name = "Manzana", Color = "Rojo" },
    new Fruit { Name = "Pera", Color = "Amarillo" }
};
```

---

4. Operaciones básicas con List

4.1. Agregar elementos

```
nombres.Add("Juan"); // Agregar al final
nombres.Insert(1, "Ana"); // Insertar en posición específica
```

4.2. Acceder a elementos

```
string primero = nombres[0];
Console.WriteLine(primero);
```

4.3. Recorrer elementos

```
foreach (var nombre in nombres)
{
    Console.WriteLine(nombre);
}
```

4.4. Modificar elementos

```
nombres[0] = "Pedro"; // Cambia el primer elemento
```

4.5. Eliminar elementos

```
nombres.Remove("Ana"); // Elimina por valor
nombres.RemoveAt(0);   // Elimina por índice
nombres.Clear();       // Vacía la lista
```

4.6. Contar elementos

```
Console.WriteLine(nombres.Count);
```

---

5. Listas de objetos personalizados

Una lista no solo guarda datos simples como string o int, también puede almacenar objetos de clases que tú definas.

```
class Fruit
{
    public string Name { get; set; }
    public string Color { get; set; }

    public override string ToString()
    {
        return $"Fruta: {Name}, Color: {Color}";
    }
}
```

Lista de objetos:

```
List<Fruit> frutas = new List<Fruit>
```

```
{
    new Fruit { Name = "Manzana", Color = "Rojo" },
    new Fruit { Name = "Pera", Color = "Amarillo" }
};
```

6. Métodos útiles de List<T>

Tabla 1.

Métodos útiles de List

Método	Descripción
`.Add(item)`	Agrega un elemento al final.
`.Insert(index, item)`	Inserta en posición específica.
`.Remove(item)`	Elimina por valor.
`.RemoveAt(index)`	Elimina por índice.
`.Clear()`	Elimina todos los elementos.
`.Contains(item)`	Verifica si existe el elemento.
`.IndexOf(item)`	Obtiene el índice de un elemento.
`.Sort()`	Ordena la lista (para tipos que implementan `IComparable`).
`.Find(predicate)`	Encuentra el primer elemento que cumpla una condición.
`.FindAll(predicate)`	Encuentra todos los elementos que cumplan una condición.

Ejemplo de búsqueda:

```
var amarillas = frutas.FindAll(f => f.Color == "Amarillo");
```

7. Buenas prácticas al usar listas

- Definir el tipo correcto en List<T> para evitar conversiones innecesarias.
- Usar var cuando el tipo es evidente.
- No recorrer una lista y modificarla a la vez.
- Usar ToString() en clases para mostrar datos legibles.
- Usar inicializadores de objetos para código más limpio.

8. Evolución de una lista al aplicar los métodos Add, Insert y Remove.

Así evoluciona una List<T> en memoria mientras aplicamos operaciones como Add, Insert y Remove.

Punto de partida

Cuando declaras una lista:

```
List<string> nombres = new List<string>();
```

Memoria:

```
nombres → [ ]  // Lista vacía (Count = 0)
```

Agregar elementos (Add)

```
nombres.Add("Juan");
nombres.Add("Ana");
```

Memoria después de las operaciones:

```
nombres → [ "Juan", "Ana" ]  // Count = 2
```

Internamente, la lista reserva un bloque de memoria y lo va ampliando cuando necesita más espacio (no crea un bloque nuevo en cada Add, sino que aumenta su capacidad de forma automática).

Insertar en una posición específica (Insert)

```
nombres.Insert(1, "Pedro");
```

Memoria:

```
nombres → [ "Juan", "Pedro", "Ana" ]  // Count = 3
```

El elemento en la posición 1 (Pedro) se desplaza y se reacomodan los índices.

Acceso por índice

```
string segundo = nombres[1]; // "Pedro"
```

Memoria (sin cambios en estructura):

```
Índice:       0        1        2
nombres → [ "Juan", "Pedro", "Ana" ]
```

El acceso por índice es O(1) (muy rápido).

O(1): No importa si procesas 10 elementos o 10 millones: el número de pasos que ejecuta el algoritmo es siempre el mismo.

Eliminar elementos

Por valor (Remove)

```
nombres.Remove("Pedro");
```

Memoria:

```
nombres → [ "Juan", "Ana" ]  // Count = 2
```

Por índice (RemoveAt)

```
nombres.RemoveAt(0);
```

Memoria:

```
nombres → [ "Ana" ]  // Count = 1
```

Limpiar lista (Clear)

```
nombres.Clear();
```

Memoria:

```
nombres → [ ]  // Lista vacía (Count = 0)
```

`Clear()` no elimina la lista, solo quita las referencias a los elementos.

Visualización con Objetos Personalizados

Ejemplo con `Fruit`:

```
List<Fruit> frutas = new List<Fruit>
{
    new Fruit { Name = "Manzana", Color = "Rojo" },
    new Fruit { Name = "Pera", Color = "Amarillo" }
};
```

Memoria (referencias a objetos):

```
frutas → [  ref#1  ,  ref#2  ]
ref#1 → { Name="Manzana", Color="Rojo" }
ref#2 → { Name="Pera", Color="Amarillo" }
```

La lista almacena referencias a objetos, no los objetos en sí.

Evolución paso a paso

Supongamos:

```
frutas.Add(new Fruit { Name = "Papaya", Color = "Amarillo" });
frutas.Insert(1, new Fruit { Name = "Mango", Color = "Rojo" });
frutas.RemoveAt(0);
```

Visual:

Después de Add:

```
[ ref#1(Manzana) , ref#2(Pera) , ref#3(Papaya) ]
```

Después de `Insert`:

```
[ ref#1(Manzana) , ref#4(Mango) , ref#2(Pera) , ref#3(Papaya) ]
```

Después de `RemoveAt(0)`:

```
[ ref#4(Mango) , ref#2(Pera) , ref#3(Papaya) ]
```

9. Ventajas de Usar List<T> ante Uso de ArrayList

En C#, List<T> y ArrayList son colecciones similares en que ambas almacenan elementos de forma secuencial, pero hay diferencias importantes. Las principales ventajas de List<T> sobre ArrayList son:

9.1. Tipado genérico (type safety)

- List<T> es genérica, lo que significa que almacena elementos de un tipo específico (int, string, Empleado, etc.) definido en el momento de su creación.
- Esto evita conversiones boxing/unboxing y errores de tipo en tiempo de compilación.

Ejemplo:

```
List<int> numeros = new List<int>(); // Solo enteros
numeros.Add(10);    // Correcto
// numeros.Add("texto"); // Error en compilación
```

En cambio, ArrayList almacena elementos como object, por lo que permite cualquier tipo, pero introduce riesgo de errores en tiempo de ejecución.

Ejemplo:

```
ArrayList numeros = new ArrayList(); // Acepta cualquier tipo

numeros.Add(10);       // Boxing implícito (int -> object)
numeros.Add("texto");  // Se agrega un string sin error de compilación

foreach (int num in numeros) // Unboxing: error en tiempo de ejecución al
encontrar "texto"
{
    Console.WriteLine(num);
}
```

9.2. Mayor rendimiento con tipos de valor

- Como List<T> no necesita convertir tipos de valor a object (boxing) ni de object al tipo original (unboxing), es más rápida y eficiente al trabajar con int, double, struct, etc.
- ArrayList sí necesita esas conversiones, lo que genera sobrecarga.

9.3. Compatibilidad con LINQ

- List<T> funciona de forma nativa con LINQ (Where, Select, OrderBy, etc.), mientras que con ArrayList habría que hacer casting o convertirla primero.

```
var mayores = numeros.Where(n => n > 5);
```

9.4. Mayor seguridad y legibilidad

- Al usar List<T> se sabe de antemano qué tipo de datos contiene, mejorando la autocompletación y reduciendo errores.
- Con ArrayList, hay que recordar o verificar el tipo de cada elemento manualmente.

9.5. Compatibilidad futura

- ArrayList es una colección de la era .NET Framework 1.0 y está en desuso para código nuevo.
- List<T> es la opción moderna recomendada por Microsoft y la usada en la mayoría de librerías actuales.