Teoría

En C#, hay varios tipos de datos compuestos o múltiples que puedes utilizar para almacenar y organizar colecciones de elementos. Algunos de los tipos más comunes son:

1. Arrays

- **Definición**: Una colección de elementos del mismo tipo almacenados en una secuencia contigua.
- **Ejemplo**: `int[] numeros = {1, 2, 3, 4};`
- **Características**: Tamaño fijo una vez creado, permite acceso por índice.

1.A. Arrays Multidimensionales

- **Definición**: C# soporta no solo arrays unidimensionales, sino también arrays **multidimensionales** (matrices) que tienen un tamaño fijo en cada dimensión al momento de su creación.
- **Ejemplo**:
  - Array bidimensional: `int[,] matriz = new int[3,2];`
  - Array tridimensional: `int[,,] cubo = new int[2, 3, 4];`
- **Características**: Tamaño fijo para cada dimensión, acceso directo por índice.

2. List (`List<T>`)

- **Definición**: Una lista genérica que puede crecer y decrecer dinámicamente.
- **Ejemplo**: `List<int> numeros = new List<int> {1, 2, 3, 4};`
- **Características**: Tamaño dinámico, permite acceso por índice, acepta duplicados.

3. Dictionary (`Dictionary<TKey, TValue>`)

- **Definición**: Una colección de pares clave-valor.
- **Ejemplo**: `Dictionary<string, int> edades = new Dictionary<string, int> { {"Juan", 25}, {"Ana", 30} };`
- **Características**: Clave única, permite acceso a valores a través de claves.

4. HashSet (`HashSet<T>`)

- **Definición**: Una colección que almacena elementos únicos y no ordenados.
- **Ejemplo**: `HashSet<int> numeros = new HashSet<int> {1, 2, 3, 4};`
- **Características**: No permite duplicados, optimizada para búsqueda rápida.

5. Queue (`Queue<T>`)

- **Definición**: Una colección que sigue el principio FIFO (First In, First Out).

- **Ejemplo**: `Queue<int> fila = new Queue<int>(); fila.Enqueue(1); fila.Enqueue(2);`
- **Características**: Los elementos se agregan al final y se eliminan desde el frente.

6. Stack (`Stack<T>`)

- **Definición**: Una colección que sigue el principio LIFO (Last In, First Out).
- **Ejemplo**: `Stack<int> pila = new Stack<int>(); pila.Push(1); pila.Push(2);`
- **Características**: Los elementos se agregan y se eliminan desde el tope de la pila.

7. LinkedList (`LinkedList<T>`)

- **Definición**: Una lista enlazada que permite insertar y eliminar elementos en cualquier lugar de la lista de manera eficiente.
- **Ejemplo**: `LinkedList<int> listaEnlazada = new LinkedList<int>();`
- **Características**: Permite nodos con referencias a los anteriores y siguientes elementos.

8. ObservableCollection (`ObservableCollection<T>`)

- **Definición**: Una colección que notifica a los suscriptores cuando hay cambios en sus elementos (agregados o eliminados).
- **Ejemplo**: `ObservableCollection<string> nombres = new ObservableCollection<string>();`
- **Características**: Utilizada para vincular datos en aplicaciones que usan el patrón MVVM.

9. SortedList (`SortedList<TKey, TValue>`)

- **Definición**: Una colección de pares clave-valor ordenada por las claves.
- **Ejemplo**: `SortedList<int, string> listaOrdenada = new SortedList<int, string>();`
- **Características**: Los elementos están ordenados automáticamente según la clave.

10. SortedSet (`SortedSet<T>`)

- **Definición**: Un conjunto que almacena elementos únicos y los ordena automáticamente.
- **Ejemplo**: `SortedSet<int> numeros = new SortedSet<int> {3, 1, 2};`
- **Características**: No permite duplicados, los elementos están ordenados.

11. Tuple (`Tuple<T1, T2, ...>`)

- **Definición**: Una estructura de datos que puede contener un número fijo de elementos de diferentes tipos.
- **Ejemplo**: `Tuple<int, string> tupla = new Tuple<int, string>(1, "uno");`
- **Características**: Inmutable, permite agrupar elementos de diferentes tipos.

12. KeyValuePair (`KeyValuePair<TKey, TValue>`)

- **Definición**: Representa un par clave-valor como un tipo de valor.
- **Ejemplo**: `KeyValuePair<int, string> par = new KeyValuePair<int, string>(1, "uno");`
- **Características**: Utilizado principalmente en estructuras como `Dictionary`.

13. BitArray

- **Definición**: Una colección de valores booleanos almacenados de manera eficiente como bits.
- **Ejemplo**: `BitArray bits = new BitArray(4);`
- **Características**: Optimizada para operaciones lógicas bit a bit.

14. `String` (aunque no es estrictamente un array)

- **Definición**: Las **cadenas de texto** (`string`) en C# pueden considerarse como una colección de caracteres con tamaño fijo, ya que una vez creada una cadena, su contenido y su longitud no se pueden modificar.
- **Ejemplo**: `string texto = "Hola";`
- **Características**: Inmutable, su longitud no puede cambiar después de la creación.

Ejercicios

Resumen de conceptos clave en los ejercicios:

- **Array**: Manipulación básica de arrays unidimensionales.
- **Array Multidimensional**: Uso de arrays bidimensionales para representar tablas.
- **Dictionary**: Uso de pares clave-valor para gestionar un inventario.
- **HashSet**: Manejo de conjuntos de elementos únicos y operaciones como inserción, eliminación y verificación de existencia.

Ejercicio 1: Operaciones con Arrays

Objetivo: Crear y manipular un array unidimensional.

Enunciado: Crea un programa que haga lo siguiente:

1. Crea un array de enteros con 5 posiciones.
2. Rellena el array con los valores 10, 20, 30, 40, 50.
3. Imprime por consola el contenido del array.
4. Modifica el valor de la tercera posición para que sea 35.
5. Vuelve a imprimir el array.
6. Calcula e imprime la suma de todos los valores del array.

Ejercicio 2: Matriz de Temperaturas (Array Multidimensional)

Objetivo: Trabajar con arrays multidimensionales (matrices).

Enunciado: Crea un programa que:

1. Cree un array bidimensional (matriz) de 3x3 que almacene temperaturas en grados Celsius.
2. Inicializa la matriz con los siguientes valores:

   [10, 12, 14]
   [15, 18, 20]
   [22, 24, 26]

3. Imprime la matriz por consola en formato de tabla.
4. Calcula la temperatura media de la matriz e imprímela por consola.

Ejercicio 3: Gestión de Inventario con Dictionary

Objetivo: Usar un `Dictionary` para gestionar pares clave-valor.

Enunciado: Crea un programa que:

1. Cree un `Dictionary` donde la clave sea el nombre de un producto (cadena) y el valor sea la cantidad disponible (entero).
2. Añade al diccionario los siguientes productos y cantidades:
    - "Manzanas" - 50
    - "Naranjas" - 30
    - "Peras" - 20
3. Imprime todos los productos y sus cantidades por consola.
4. Actualiza la cantidad de "Naranjas" a 40.
5. Añade un nuevo producto "Plátanos" con una cantidad de 25.
6. Elimina el producto "Peras".
7. Imprime el inventario final.

Ejercicio 4: Conjunto de Estudiantes con HashSet

Objetivo: Usar un `HashSet` para almacenar elementos únicos.

Enunciado: Crea un programa que:

1. Cree un `HashSet` para almacenar los nombres de estudiantes en una clase.
2. Añade los siguientes estudiantes: "Ana", "Juan", "Pedro", "Lucía".
3. Intenta añadir a "Ana" de nuevo al `HashSet` y explica qué ocurre.
4. Imprime todos los nombres de estudiantes.
5. Comprueba si el estudiante "Lucía" está en el conjunto.
6. Elimina a "Pedro" del conjunto.
7. Vuelve a imprimir el conjunto de estudiantes.

Ejercicio Final: Máquina de Cambio de Monedas con `enum`

Objetivo: Implementar una máquina de cambio de monedas usando un `enum` que represente los tipos de monedas, y calcular el número mínimo de monedas necesarias para una cantidad dada.

Enunciado:

1. Crea un `enum` llamado `Monedas` que tenga los siguientes valores asociados a cada tipo de moneda:
    - `Moneda50 = 50`
    - `Moneda20 = 20`
    - `Moneda10 = 10`
    - `Moneda5 = 5`
    - `Moneda2 = 2`
    - `Moneda1 = 1`
2. Implementa un método que reciba una cantidad en céntimos y utilice el `enum` para calcular el número mínimo de monedas necesarias para obtener ese valor.
3. El programa debe solicitar al usuario que ingrese una cantidad de dinero en céntimos y luego mostrar el número mínimo de monedas de cada tipo necesarias.