Collections

Prepared by:

Kareem Hany AbdElkader Ali

18/7/2024 ITI Summer Training

◆ Introduction:

Collections is a group of element that is used to store, organize, and manipulate groups of related objects or elements. Collections provide a way to manage multiple items as a single unit, allowing efficient access, modification, and iteration over the stored data.

◆ Characteristics of Collections:

- ◆ **Element Access**:
  - **Enumeration**: Every collection can be enumerated to access each element sequentially.
  - **Index-based Access**: Some collections allow accessing elements by index, such as `List<T>`, where elements are accessed by their position in the collection.
  - **Key-based Access**: Other collections, like `Dictionary<TKey, TValue>`, allow accessing elements by key, where each value is associated with a unique key.

- ◆ **Performance Profile**:
  - **Operations**: Different collections have varying performance profiles for operations like adding, finding, or removing elements.
  - **Selection Criteria**: Choose a collection type based on the operations that your application performs most frequently.

For example, `List<T>` might be suitable for frequent element access by index, while `Dictionary<TKey, TValue>` is ideal for fast lookups by key.

- **Dynamic Growth and Shrinkage**:
  - **Dynamic Resizing**: Most collections support dynamic resizing, allowing elements to be added or removed as needed without requiring fixed pre-allocation of memory.
  - **Exceptions**: Not all collections support dynamic resizing; for instance, `Array`, `Span<T>`, and `Memory<T>` have fixed sizes once initialized.

◆ Types of Collections:

There are 3 namespace that we used to work with collections:

- **System.Collections.Generic classes**
- **System.Collections classes (Now deprecated)**
- **System.Collections.Concurrent classes**

1. System.Collections.Generic Classes

The `System.Collections.Generic` namespace provides a range of strongly-typed collection classes designed for improved type safety and performance:

1. **List**: Resizable array-based list for storing elements of a specified type.
   - ★ **Properties**:
     - `Count`: Gets the number of elements contained in the `List<T>`.

- `Capacity`: Gets or sets the number of elements that the `List<T>` can contain before resizing is required.

★ **Methods**:

- `Add(T item)`: Adds an object to the end of the `List<T>`.
- `Insert(int index, T item)`: Inserts an element into the `List<T>` at the specified index.
- `Remove(T item)`: Removes the first occurrence of a specific object from the `List<T>`.
- `RemoveAt(int index)`: Removes the element at the specified index.
- `Clear()`: Removes all elements from the `List<T>`.

2. **Stack**: Last-In-First-Out (LIFO) collection, useful for stack-based operations.

★ **Properties**:

- Count: Gets the number of elements contained in the Stack<T>.

★ **Methods**:

- Push(T item): Pushes an object onto the top of the Stack<T>.
- Pop(): Pops the object at the top of the Stack<T> and removes it.
- Peek(): Returns the object at the top of the Stack<T> without removing it.
- Clear(): Removes all objects from the Stack<T>.
-
- Remove(T item): Removes the first occurrence of a specific object from the List<T>.
- RemoveAt(int index): Removes the element at the specified index.

- Clear(): Removes all elements from the List<T>.

3. **Queue**: First-In-First-Out (FIFO) collection, ideal for queue-based operations.
    - ★ **Properties:**
        - Count: Gets the number of elements contained in the Queue<T>.
    - ★ **Methods:**
        - Enqueue(T item): Adds an object to the end of the Queue<T>.
        - Dequeue(): Removes and returns the object at the beginning of the Queue<T>.
        - Peek(): Returns the object at the beginning of the Queue<T> without removing it.
        - Clear(): Removes all objects from the Queue<T>.

4. **LinkedList**: Doubly linked list implementation allowing efficient insertions and deletions.
    - ★ **Properties:**
        - Count: Gets the number of elements contained in the LinkedList<T>.
    - ★ **Methods:**
        - AddFirst(T value): Adds a new node containing the specified value at the start of the LinkedList<T>.
        - AddLast(T value): Adds a new node containing the specified value at the end of the LinkedList<T>.
        - Remove(T value): Removes the first occurrence of the specified value from the LinkedList<T>.
        - Find(T value): Searches for the first node that contains the specified value.

5. **HashSet**: Unordered collection of unique elements, optimized for fast access and set operations.
    - ★ **Properties:**
        - Count: Gets the number of elements contained in the HashSet<T>.
    - ★ **Methods:**
        - Add(T item): Adds an element to the HashSet<T>.
        - Remove(T item): Removes the specified element from the HashSet<T>.
        - Contains(T item): Determines whether the HashSet<T> contains the specified element.
        - Clear(): Removes all elements from the HashSet<T>.

6. **SortedSet**: Collection of unique elements sorted in a specific order defined by a comparer.
    - ★ **Properties:**
        - Count: Gets the number of elements contained in the SortedSet<T>.
    - ★ **Methods:**
        - Add(T item): Adds an element to the SortedSet<T>.
        - Remove(T item): Removes the specified element from the SortedSet<T>.
        - Contains(T item): Determines whether the SortedSet<T> contains the specified element.
        - Clear(): Removes all elements from the SortedSet<T>.

7. **Dictionary**: Key-value pair collection allowing fast lookup by key.
    - ★ **Properties:**

- Count: Gets the number of key-value pairs contained in the Dictionary<TKey, TValue>.
- Keys: Gets a collection of keys in the Dictionary<TKey, TValue>.
- Values: Gets a collection of values in the Dictionary<TKey, TValue>.

★ **Methods:**

- Add(TKey key, TValue value): Adds a key-value pair to the Dictionary<TKey, TValue>.
- Remove(TKey key): Removes the value with the specified key from the Dictionary<TKey, TValue>.
- TryGetValue(TKey key, out TValue value): Tries to get the value associated with the specified key.
- Clear(): Removes all keys and values from the Dictionary<TKey, TValue>.

8. **SortedDictionary**: Sorted key-value pair collection based on keys.

★ **Properties:**

- Count: Gets the number of key-value pairs contained in the SortedDictionary<TKey, TValue>.
- Keys: Gets a collection of keys in the SortedDictionary<TKey, TValue>.
- Values: Gets a collection of values in the SortedDictionary<TKey, TValue>.

★ **Methods:**

- Add(TKey key, TValue value): Adds a key-value pair to the SortedDictionary<TKey, TValue>.
- Remove(TKey key): Removes the value with the specified key from the SortedDictionary<TKey, TValue>.
- TryGetValue(TKey key, out TValue value): Tries to get the value associated with the specified key.
- Clear(): Removes all keys and values from the SortedDictionary<TKey, TValue>.

9. **SortedList**: Sorted key-value pair collection implemented using a list structure.
    - ★ **Properties**:
        - Count: Gets the number of key-value pairs contained in the SortedList<TKey, TValue>.
        - Keys: Gets a collection of keys in the SortedList<TKey, TValue>.
        - Values: Gets a collection of values in the SortedList<TKey, TValue>.
    - ★ **Methods**:
        - Add(TKey key, TValue value): Adds a key-value pair to the SortedList<TKey, TValue>.
        - Remove(TKey key): Removes the value with the specified key from the SortedList<TKey, TValue>.
        - TryGetValue(TKey key, out TValue value): Tries to get the value associated with the specified key.
        - Clear(): Removes all keys and values from the SortedList<TKey, TValue>.

2. System.Collections Classes (Legacy)

The `System.Collections` namespace includes legacy non-generic collection classes, now generally superseded by their `System.Collections.Generic` equivalents:

1. **ArrayList**: Resizable array-based list accommodating elements of any type (non-generic).
    - ★ **Properties:**
        - Count: Gets the number of elements contained in the ArrayList.
        - Capacity: Gets or sets the number of elements that the ArrayList can contain before resizing is required.
        - Item[int index]: Gets or sets the element at the specified index.
    - ★ **Methods:**

- Add(object value): Adds an object to the end of the ArrayList.
- AddRange(ICollection c): Adds the elements of an ICollection to the end of the ArrayList.
- Clear(): Removes all elements from the ArrayList.
- Contains(object value): Determines whether an element is in the ArrayList.
- CopyTo(Array array): Copies the entire ArrayList to a compatible one-dimensional array.
- Insert(int index, object value): Inserts an element into the ArrayList at the specified index.
- Remove(object value): Removes the first occurrence of a specific object from the ArrayList.
- RemoveAt(int index): Removes the element at the specified index.
- Sort(): Sorts the elements in the ArrayList.
- ToArray(): Copies the ArrayList to a new array.

2. **Stack**: LIFO collection for stack-based operations.
   - ★ **Properties:**
     - Count: Gets the number of elements contained in the Stack.
   - ★ **Methods:**
     - Push(object value): Pushes an object onto the top of the Stack.
     - Pop(): Pops the object at the top of the Stack and removes it.
     - Peek(): Returns the object at the top of the Stack without removing it.
     - Clear(): Removes all objects from the Stack.
     - Contains(object obj): Determines whether an element is in the Stack.
     - ToArray(): Copies the Stack to a new array.

3. **Queue**: FIFO collection for queue-based operations.
    - ★ **Properties:**
        - Count: Gets the number of elements contained in the Queue.
    - ★ **Methods:**
        - Enqueue(object value): Adds an object to the end of the Queue.
        - Dequeue(): Removes and returns the object at the beginning of the Queue.
        - Peek(): Returns the object at the beginning of the Queue without removing it.
        - Clear(): Removes all objects from the Queue.
        - Contains(object obj): Determines whether an element is in the Queue.
        - ToArray(): Copies the Queue to a new array.

4. **Hashtable**: Key-value pair collection allowing fast lookup (non-generic).
    - ★ **Properties:**
        - Count: Gets the number of key-value pairs contained in the Hashtable.
        - Keys: Gets an ICollection containing the keys of the Hashtable.
        - Values: Gets an ICollection containing the values of the Hashtable.
    - ★ **Methods:**
        - **A**dd(object key, object value): Adds an element with the provided key and value into the Hashtable.
        - Contains(object key): Determines whether the Hashtable contains a specific key.
        - ContainsValue(object value): Determines whether the Hashtable contains a specific value.
        - Remove(object key): Removes the element with the specified key from the Hashtable.

- Clear(): Removes all keys and values from the Hashtable.
- TryGetValue(object key, out object value): Gets the value associated with the specified key.

3. System.Collections.Concurrent Classes

The `System.Collections.Concurrent` namespace offers thread-safe collection classes designed for concurrent access scenarios without the need for external synchronization:

- **BlockingCollection**: Provides blocking and bounding capabilities for producer-consumer scenarios.
  - ★ **Properties:**
    - Count: Gets the number of elements contained in the BlockingCollection<T>.
    - IsEmpty: Gets a value indicating whether the BlockingCollection<T> is empty.
    - IsCompleted: Gets a value indicating whether the BlockingCollection<T> has been marked as complete for adding.
  - ★ **Methods:**
    - Add(T item): Adds an item to the BlockingCollection<T>. Blocks if the collection is full.
    - Add(T item, CancellationToken cancellationToken): Adds an item to the BlockingCollection<T> with cancellation support.
    - CompleteAdding(): Marks the BlockingCollection<T> as complete for adding operations.
    - TryAdd(T item): Attempts to add an item to the BlockingCollection<T>, returning a value indicating success.
    - TryTake(out T item): Attempts to take an item from the BlockingCollection<T>, returning a value indicating success.

        - Take(): Removes and returns an item from the BlockingCollection<T>, blocking if necessary.

- **ConcurrentBag**: Unordered collection optimized for concurrent access.
    - ★ **Properties:**
        - Count: Gets the number of elements contained in the ConcurrentBag<T>.
    - ★ **Methods:**
        - Add(T item): Adds an item to the ConcurrentBag<T>.
        - TryTake(out T item): Attempts to remove and return an item from the ConcurrentBag<T>.
        - TryPeek(out T item): Attempts to return an item from the ConcurrentBag<T> without removing it.

- **ConcurrentStack**: LIFO collection optimized for concurrent access.
    - ★ **Properties:**
        - Count: Gets the number of elements contained in the ConcurrentStack<T>.
    - ★ **Methods:**
        - Push(T item): Adds an item to the top of the ConcurrentStack<T>.
        - TryPop(out T item): Attempts to pop and return an item from the top of the ConcurrentStack<T>.
        - TryPeek(out T item): Attempts to return the item at the top of the ConcurrentStack<T> without removing it.

- **ConcurrentQueue**: FIFO collection optimized for concurrent access.
    - ★ **Properties:**

        - Count: Gets the number of elements contained in the ConcurrentQueue<T>.
    - ★ **Methods:**
        - Enqueue(T item): Adds an item to the end of the ConcurrentQueue<T>.
        - TryDequeue(out T item): Attempts to dequeue and return an item from the beginning of the ConcurrentQueue<T>.
        - TryPeek(out T item): Attempts to return the item at the beginning of the ConcurrentQueue<T> without removing it.

- ◆ **ConcurrentDictionary**: Thread-safe dictionary implementation for concurrent key-value operations.
    - ★ **Properties:**
        - Count: Gets the number of key-value pairs contained in the ConcurrentDictionary<TKey, TValue>.
        - Keys: Gets a collection containing the keys of the ConcurrentDictionary<TKey, TValue>.
        - Values: Gets a collection containing the values of the ConcurrentDictionary<TKey, TValue>.
    - ★ **Methods:**
        - TryAdd(TKey key, TValue value): Attempts to add a key-value pair to the ConcurrentDictionary<TKey, TValue>.
        - TryRemove(TKey key, out TValue value): Attempts to remove a key-value pair from the ConcurrentDictionary<TKey, TValue>.
        - TryGetValue(TKey key, out TValue value): Attempts to get the value associated with the specified key.
        - AddOrUpdate(TKey key, TValue addValue, Func<TKey, TValue, TValue> updateValueFactory): Adds a key-value pair if the

key does not exist or updates the value if the key already exists.
    - GetOrAdd(TKey key, TValue value): Adds a key-value pair if the key does not exist or returns the existing value.

◆ Generic and Non-Generic Collections :

Generic Collections

- **Namespace**: `System.Collections.Generic`
- **Type Safety**: Enforces type safety (e.g., `List<int>`)
- **Performance**: Better performance, no boxing/unboxing
- **Examples**: `List<T>`, `Dictionary<TKey, TValue>`, `HashSet<T>`

Non-Generic Collections

- **Namespace**: `System.Collections`
- **Type Safety**: No type safety (e.g., `ArrayList`)
- **Performance**: May involve boxing/unboxing
- **Examples**: `ArrayList`, `Hashtable`, `Queue`

Generic Collection Example Using List<T>:

```
1. using System;
2. using System.Collections.Generic;
3. 
4. class Program
5. {
6.     static void Main()
7.     {
8.         // Create a List of integers
9.         List<int> numbers = new List<int>();
10.
11.          // Add elements to the list
12.          numbers.Add(1);
13.          numbers.Add(2);
```

```
14.          numbers.Add(3);
15.
16.          // Access elements
17.          foreach (int number in numbers)
18.          {
19.            Console.WriteLine(number); // Output: 1
  2 3
20.          }
21.
22.          // Uncommenting the following line will
  cause a compile-time error
23.          // numbers.Add("hello"); // Error: Cannot
  add 'string' to 'List<int>'
24.      }
25.  }
```

In this example:

- List<int> can only store integers.
- Attempting to add a string will result in a compile-time error.

Non-Generic Collection Example Using ArrayList:

```
1.using System;
2.using System.Collections;
3.
4.class Program
5.{
6.    static void Main()
7.    {
8.        // Create an ArrayList
9.        ArrayList items = new ArrayList();
10.
11.          // Add elements to the ArrayList
12.          items.Add(1);
13.          items.Add("hello");
14.          items.Add(3.14);
15.
```

```
16.          // Access elements
17.          foreach (var item in items)
18.          {
19.              Console.WriteLine(item); // Output: 1
  hello 3.14
20.          }
21.
22.          // Type casting is needed to retrieve
  items
23.          int number = (int)items[0]; // No
  compile-time error, but runtime error if casting is
  incorrect
24.      }
25.  }
```

In this example:

- ArrayList can store any type of object.
- You need to cast objects to their original type when retrieving them, which can lead to runtime errors if the cast is incorrect.

Use generic collections for type safety and better performance. Non-generic collections are less type-safe and may have performance issues but offer flexibility.

◆ Conclusion:

Understanding these collection types is essential for selecting the appropriate data structures based on performance requirements, thread safety considerations, and specific application needs. System.Collections.Generic classes offer type safety and efficiency, while System.Collections.Concurrent classes cater to concurrent and parallel processing requirements in multi-threaded environments.