## **Lists, Queues, Stacks, and Sets: Essential Concrete Collections**

The .NET Base Class Library (BCL) provides a robust set of concrete collection classes that implement the various interfaces we've discussed. For most modern C# development, you'll predominantly use the **generic** versions of these classes due to their type safety, performance, and flexibility. The non-generic counterparts (e.g., ArrayList) are largely superseded, maintained primarily for backward compatibility. Among these, List<T> is arguably the most frequently used.

### **List<T> and ArrayList: Dynamically Sized Arrays**

* **List<T> (Generic):** Provides a dynamically sized array of objects of type T. It implements IList<T>, IReadOnlyList<T>, and IList (the non-generic version). All methods, including Add, Remove, and Insert, are publicly exposed and work intuitively.
* **ArrayList (Non-Generic):** The non-generic equivalent. It implements IList.
* **Internal Mechanism:** Both List<T> and ArrayList maintain an internal array. When this array reaches its capacity, a new, larger array is allocated, and the existing elements are copied over.
  + **Appending Elements:** Add() is generally efficient because there's often available space at the end of the internal array.
  + **Inserting/Removing Elements (Mid-List):** Can be slow because elements after the insertion/removal point must be shifted in memory.
  + **Searching:** Efficient if BinarySearch() is used on a sorted list; otherwise, it's a linear scan (O(n)).
* **List<T> Performance Advantage:** List<T> is significantly faster than ArrayList, especially when T is a value type. This is because List<T> avoids the boxing and unboxing overhead that ArrayList incurs when storing or retrieving value types (as ArrayList stores everything as object).
* **Constructors:**
  + new List<T>(): Creates an empty list.
  + new List<T>(IEnumerable<T> collection): Initializes the list with elements copied from an existing collection.
  + new List<T>(int capacity): Initializes the list with a specified initial capacity, reducing the number of internal array reallocations if you know the approximate size.
* **Key List<T> Members (highlights):**
  + **Add/Insert:** Add(T item), AddRange(IEnumerable<T> collection), Insert(int index, T item), InsertRange(int index, IEnumerable<T> collection).
  + **Remove:** Remove(T item), RemoveAt(int index), RemoveRange(int index, int count), RemoveAll(Predicate<T> match) (removes all elements matching a condition).
  + **Indexing:** this[int index] (indexer for direct access), GetRange(int index, int count).
  + **Conversion/Export:** ToArray(), CopyTo(), ConvertAll<TOutput>(Converter<T, TOutput> converter).
  + **Other:** Reverse(), Capacity (get/set internal array size), TrimExcess() (reduces capacity to current count).
* **ArrayList Disadvantage:** Requires explicit casting when retrieving elements, which incurs runtime overhead and can lead to InvalidCastException if the type is incorrect (compiler cannot verify).

| ArrayList al = new ArrayList(); al.Add("hello"); string first = (string)al[0]; // Requires cast // int val = (int)al[0]; // Compiles but fails at runtime |
| --- |

* + ArrayList can be useful when you *need* to store mixed-type elements without a common base type other than object, or for reflection scenarios. However, List<object> is often a safer and more performant alternative.
* **Converting ArrayList to List<T> (using LINQ):**

| ArrayList al = new ArrayList(); al.AddRange(new[] { 1, 5, 9 } ); List<int> list = al.Cast<int>().ToList(); // Cast<T>() and ToList() are LINQ extension methods |
| --- |

### 

### **LinkedList<T>: Doubly Linked Lists**

* LinkedList<T> provides a generic **doubly linked list**. Each node in the list contains a reference to the previous node, the next node, and the actual data (T).
* **Key Benefit (Efficient Insertion/Removal):** Adding or removing an element anywhere in the list is very efficient (O(1)) because it only involves creating/destroying a node and updating a few references.
* **Key Drawback (Inefficient Access/Searching):** LinkedList<T> does not support direct indexing (e.g., list[index]). Finding a specific node or element requires traversing the list from the beginning or end (O(n)). Binary search is not possible.
* **Interfaces:** Implements IEnumerable<T> and ICollection<T> (and their non-generic versions), but **not IList<T>** due to the lack of indexed access.
* **LinkedListNode<T>:** Elements are wrapped in LinkedListNode<T> instances, which expose Next, Previous, and Value properties.
* **Methods for Adding/Removing:**
  + AddFirst(), AddLast(): Adds to the beginning/end.
  + AddAfter(), AddBefore(): Adds relative to an existing node.
  + RemoveFirst(), RemoveLast(), Remove(T value), Remove(LinkedListNode<T> node).
* **Properties:** Count, First, Last (all are fast O(1) operations).
* **Searching:** Contains(), Find(), FindLast() (all are O(n) as they require traversal).
* **Usage Example:**

| var tune = new LinkedList<string>(); tune.AddFirst("do"); // do tune.AddLast("so"); // do - so tune.AddAfter(tune.First, "re"); // do - re - so // ... more operations foreach (string s in tune) Console.WriteLine(s); // Enumeration works |
| --- |

### 

### **Queue<T> and Queue: First-In, First-Out (FIFO)**

* Queue<T> (Generic) and Queue (Non-Generic) implement a **First-In, First-Out (FIFO)** data structure, similar to a real-world queue.
* **Operations:**
  + Enqueue(T item): Adds an item to the *tail* of the queue.
  + Dequeue(): Retrieves and removes the item from the *head* of the queue. Throws an exception if the queue is empty.
  + Peek(): Retrieves the item at the *head* without removing it.
  + Count: Returns the number of elements.
  + ToArray(): Copies elements to a new array.
* **Interfaces:** Implement IEnumerable<T>, ICollection, and IEnumerable. They do **not** implement IList<T> as elements cannot be accessed by index directly.
* **Internal Implementation:** Internally, queues use a dynamically resizing array, much like List<T>. Enqueue and Dequeue operations are typically very quick (O(1)), except when an internal array resize is required.
* **Usage Example:**

| var q = new Queue<int>(); q.Enqueue(10); // Queue: [10] q.Enqueue(20); // Queue: [10, 20] (10 is at head)  Console.WriteLine(q.Peek()); // Prints 10 (Queue: [10, 20]) Console.WriteLine(q.Dequeue()); // Prints 10 (Queue: [20]) Console.WriteLine(q.Dequeue()); // Prints 20 (Queue: []) // Console.WriteLine(q.Dequeue()); // Throws exception (queue empty) |
| --- |

### **Stack<T> and Stack: Last-In, First-Out (LIFO)**

* Stack<T> (Generic) and Stack (Non-Generic) implement a **Last-In, First-Out (LIFO)** data structure, like a stack of plates.
* **Operations:**
  + Push(T item): Adds an item to the *top* of the stack.
  + Pop(): Retrieves and removes the item from the *top* of the stack. Throws an exception if the stack is empty.
  + Peek(): Retrieves the item at the *top* without removing it.
  + Count: Returns the number of elements.
  + ToArray(): Copies elements to a new array.
* **Interfaces:** Implement IEnumerable<T>, ICollection, and IEnumerable. They do **not** implement IList<T>.
* **Internal Implementation:** Similar to Queue<T> and List<T>, stacks use a dynamically resizing internal array. Push and Pop are generally very fast (O(1)).
* **Usage Example:**

| var s = new Stack<int>(); s.Push(1); // Stack = 1 s.Push(2); // Stack = 1,2 s.Push(3); // Stack = 1,2,3  Console.WriteLine(s.Peek()); // Prints 3, Stack = 1,2,3 Console.WriteLine(s.Pop()); // Prints 3, Stack = 1,2 Console.WriteLine(s.Pop()); // Prints 2, Stack = 1 // Console.WriteLine(s.Pop()); // Throws exception |
| --- |

### **BitArray: Compact Boolean Collection**

* A BitArray is a dynamically sized collection specifically for bool values.
* **Memory Efficiency:** It is highly memory efficient, using only **one bit per boolean value**, unlike bool which typically occupies one byte.
* **Indexing:** The indexer bits[index] reads and writes individual bits.
* **Bitwise Operations:** Provides methods for common bitwise operations: And, Or, Xor, Not. These modify the BitArray in place.

| var bits = new BitArray(2); // Two bits, initialized to false bits[1] = true; // bits: [false, true] bits.Xor(bits); // XOR bits with itself (results in all false) Console.WriteLine(bits[1]); // False |
| --- |

### **HashSet<T> and SortedSet<T>: Unique Collections**

These are **set** data structures, designed to store unique elements.

* **Common Features (ISet<T> interface):**
  + **No Duplicates:** They automatically ignore attempts to add duplicate elements.
  + **Fast Contains:** Their Contains methods execute very quickly (average O(1) for HashSet<T>, O(log n) for SortedSet<T>) due to their underlying implementations.
  + **No Positional Access:** You cannot access elements by index.
  + RemoveWhere(Predicate<T> match): Removes elements matching a condition.
* **HashSet<T>:**
  + **Order:** Does *not* maintain elements in any particular order.
  + **Implementation:** Uses a **hash table** internally (stores only keys).
* **SortedSet<T>:**
  + **Order:** Maintains elements in **sorted order**.
  + **Implementation:** Uses a **red/black tree** internally.
  + **Additional Members:** Offers Min, Max properties, Reverse() for reverse enumeration, and GetViewBetween(T lowerValue, T upperValue) for sub-ranges. Can accept an IComparer<T> in its constructor for custom ordering.
* **Set Operations (Highlights):** Both HashSet<T> and SortedSet<T> provide powerful set operations:  
  + **Destructive (modify the set):**
    - UnionWith(IEnumerable<T> other): Adds all elements from other to the current set (excluding duplicates).
    - IntersectWith(IEnumerable<T> other): Keeps only elements present in *both* sets.
    - ExceptWith(IEnumerable<T> other): Removes elements present in other from the current set.
    - SymmetricExceptWith(IEnumerable<T> other): Keeps only elements that are unique to *either* set (not in both).
  + **Non-Destructive (query the set):**
    - IsSubsetOf(IEnumerable<T> other)
    - IsProperSubsetOf(IEnumerable<T> other)
    - IsSupersetOf(IEnumerable<T> other)
    - IsProperSupersetOf(IEnumerable<T> other)
    - Overlaps(IEnumerable<T> other)
    - SetEquals(IEnumerable<T> other) (checks if both sets contain the exact same elements).
* **Usage Example (HashSet<T>):**

| var letters = new HashSet<char>("the quick brown fox"); // Initialized from string (IEnumerable<char>) Console.WriteLine(letters.Contains('t')); // true Console.WriteLine(letters.Contains('j')); // false foreach (char c in letters) Console.Write(c); // Output: the quickbrownfx (order not guaranteed, no duplicates)  var vowels = new HashSet<char>("aeiou"); letters.IntersectWith(vowels); // Modifies 'letters' to only contain common elements foreach (char c in letters) Console.Write(c); // Output: euio |
| --- |