# Introduction to Collection Classes and Generics

Collections are essential components in programming that allow developers to store, manage, and manipulate groups of related objects efficiently. They provide flexible and dynamic ways to handle data compared to traditional arrays. This guide will delve into various collection classes, interfaces, and generic concepts, providing detailed explanations and examples to help you understand and utilize them effectively in your programming endeavors.

---

## 1. ArrayList

### Overview

- **Namespace:** `System.Collections`

- **Description:** `ArrayList` is a non-generic, dynamically sized collection that can store objects of any type. It is essentially an array that can grow or shrink as needed, providing flexibility over traditional fixed-size arrays.

### Key Features

- **Dynamic Sizing:** Automatically resizes as elements are added or removed.

- **Non-generic:** Stores elements as objects, which may require type casting.

- **Indexed Access:** Allows accessing elements by index, similar to arrays.

- **Performance:** Offers good performance for add and retrieve operations but may be slower compared to generic collections due to boxing and unboxing overhead.

### Common Methods

- `Add(object value)`: Adds an object to the end of the ArrayList.

- `Insert(int index, object value)`: Inserts an object at the specified index.

- `Remove(object value)`: Removes the first occurrence of a specific object.

- `RemoveAt(int index)`: Removes the element at the specified index.

- `Contains(object value)`: Determines whether the ArrayList contains a specific object.
- `Sort()`: Sorts the elements in the ArrayList.

### Example Usage

```csharp
using System;
using System.Collections;

public class ArrayListExample
{
    public static void Main()
    {
        ArrayList list = new ArrayList();

        // Adding elements
        list.Add(1);
        list.Add("two");
        list.Add(3.0);
```

```
        // Inserting element
        list.Insert(1, "inserted element");

        // Accessing elements
        Console.WriteLine(list[0]); // Output: 1

        // Removing elements
        list.Remove("two");

        // Iterating through the ArrayList
        foreach (var item in list)
        {
            Console.WriteLine(item);
        }
    }
}
```

### When to Use

- When you need a simple, dynamically sized collection and are working with legacy code that doesn't support generics.

- For storing heterogeneous collections where elements are of different types.

### Drawbacks

- **Type Safety:** Lack of type safety can lead to runtime errors.

- **Performance Overhead:** Boxing and unboxing operations can degrade performance.

- **Obsolete:** In modern applications, generic collections like `List<T>` are preferred for type safety and performance.

---

## 2. HashTable

### Overview

- **Namespace:** `System.Collections`

- **Description:** `Hashtable` is a collection that stores key-value pairs and provides fast retrieval

based on the key. It uses a hash code of the key to organize and access data efficiently.

### Key Features

- **Key-Value Storage:** Stores data in pairs, allowing efficient lookup by key.

- **Non-generic:** Keys and values are stored as objects.

- **Unordered:** Does not maintain any order of elements.

- **Performance:** Offers constant-time complexity for add, remove, and lookup operations under ideal conditions.

### Common Methods

- `Add(object key, object value)`: Adds an element with the specified key and value.

- `Remove(object key)`: Removes the element with the specified key.

- `ContainsKey(object key)`: Determines whether the Hashtable contains a specific key.

- `ContainsValue(object value)`: Determines whether the Hashtable contains a specific value.

- `Clear()`: Removes all elements from the Hashtable.

### Example Usage

```csharp
using System;
using System.Collections;

public class HashtableExample
{
    public static void Main()
    {
        Hashtable hashtable = new Hashtable();

        // Adding key-value pairs
        hashtable.Add("001", "John Doe");
        hashtable.Add("002", "Jane Smith");
        hashtable.Add("003", "Jim Brown");

        // Accessing value by key
```

```
        Console.WriteLine(hashtable["002"]); // Output: Jane Smith

        // Checking for key existence
        if (hashtable.ContainsKey("003"))
        {
            Console.WriteLine("Key found!");
        }

        // Removing a key-value pair
        hashtable.Remove("001");

        // Iterating through the Hashtable
        foreach (DictionaryEntry entry in hashtable)
        {
            Console.WriteLine($"{entry.Key}: {entry.Value}");
        }
    }
}
```

### When to Use

- When you need fast lookups, additions, and deletions based on keys.

- Suitable for scenarios where order of elements is not important.

- For storing collections with non-generic types in legacy applications.

### Drawbacks

- **Type Safety:** Lack of generics leads to potential runtime errors.

- **Obsolete:** Modern applications prefer `Dictionary<TKey, TValue>` for type safety and better performance.

- **Thread Safety:** Not thread-safe without manual synchronization.

---

## 3. Dictionary

### Overview

- **Namespace:** `System.Collections.Generic`

- **Description:** `Dictionary<TKey, TValue>` is a generic collection that stores key-value pairs, providing fast retrieval based on keys with type safety.

### Key Features

- **Generic:** Enforces type safety for both keys and values.

- **Key-Value Storage:** Efficient storage and retrieval of data using keys.

- **Unordered:** Does not maintain order; entries are organized based on hash codes.

- **Performance:** Provides O(1) time complexity for most operations.

### Common Methods

- `Add(TKey key, TValue value)`: Adds an element with the specified key and value.

- `Remove(TKey key)`: Removes the element with the specified key.

- `TryGetValue(TKey key, out TValue value)`: Gets the value associated with the specified key.
- `ContainsKey(TKey key)`: Determines whether the dictionary contains the specified key.
- `Clear()`: Removes all elements from the dictionary.

### Example Usage

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

public class DictionaryExample
{
    public static void Main()
    {
        Dictionary<int, string> dictionary = new Dictionary<int, string>();

        // Adding key-value pairs
        dictionary.Add(1, "Apple");
```

```
        dictionary.Add(2, "Banana");
        dictionary.Add(3, "Cherry");

        // Accessing value by key
        Console.WriteLine(dictionary[2]); // Output: Banana

        // Using TryGetValue
        if (dictionary.TryGetValue(3, out string value))
        {
            Console.WriteLine(value); // Output: Cherry
        }

        // Removing a key-value pair
        dictionary.Remove(1);

        // Iterating through the Dictionary
        foreach (var kvp in dictionary)
        {
            Console.WriteLine($"{kvp.Key}: {kvp.Value}");
        }
```

```
    }
}
```

### When to Use

- When you need efficient lookup, addition, and removal of elements by key with type safety.

- Suitable for storing large datasets where quick access is required.

- When keys and values are of specific, known types.

### Advantages over Hashtable

- **Type Safety:** Prevents runtime errors due to type mismatches.

- **Performance:** Eliminates boxing/unboxing overhead.

- **Enhanced Features:** Supports additional functionalities like initialization with capacity and custom comparers.

### Drawbacks

- **Unordered:** Does not maintain insertion order; use `SortedDictionary` or `OrderedDictionary` if order matters.

- **Key Uniqueness:** Keys must be unique; attempting to add duplicate keys will throw exceptions.

---

## 4. Stack

### Overview

- **Namespace:** `System.Collections` (non-generic), `System.Collections.Generic` (generic)

- **Description:** `Stack` is a collection that follows the Last-In-First-Out (LIFO) principle. Elements are added and removed from the top of the stack.

### Key Features

- **LIFO Behavior:** The last element added is the first to be removed.

- **Generic and Non-generic Versions:** Supports both type-safe and non-type-safe implementations.

- **Operations:** Primary operations are Push (add), Pop (remove), and Peek (view top element).

### Common Methods

- `Push(T item)`: Adds an item to the top of the stack.
- `Pop()`: Removes and returns the item at the top of the stack.
- `Peek()`: Returns the item at the top without removing it.
- `Contains(T item)`: Determines whether the stack contains a specific item.
- `Clear()`: Removes all items from the stack.

### Example Usage

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

public class StackExample
{
    public static void Main()
```

```
{
    Stack<string> stack = new Stack<string>();

    // Pushing items onto the stack
    stack.Push("First");
    stack.Push("Second");
    stack.Push("Third");

    // Peeking at the top item
    Console.WriteLine(stack.Peek()); // Output: Third

    // Popping items from the stack
    Console.WriteLine(stack.Pop()); // Output: Third
    Console.WriteLine(stack.Pop()); // Output: Second

    // Checking if stack contains an item
    if (stack.Contains("First"))
    {
```

```
            Console.WriteLine("Stack contains 'First'");
        }

        // Iterating through the stack
        foreach (var item in stack)
        {
            Console.WriteLine(item);
        }
    }
}
```

### When to Use

- When you need to process items in reverse order of their addition.

- Suitable for algorithms like depth-first search, undo mechanisms, and expression evaluation.

### Advantages

- **Simple and Efficient:** Provides efficient addition and removal operations.

- **Thread Safety:** Can be made thread-safe using concurrent collections like `ConcurrentStack<T>`.

### Drawbacks

- **Limited Access:** Can only access the top element directly.

- **No Random Access:** Cannot access elements by index.

---

## 5. Queue

### Overview

- **Namespace:** `System.Collections` (non-generic), `System.Collections.Generic` (generic)

- **Description:** `Queue` is a collection that follows the First-In-First-Out (FIFO) principle. Elements are added at the end and removed from the front.

### Key Features

- **FIFO Behavior:** The first element added is the first to be removed.

- **Generic and Non-generic Versions:** Supports both type-safe and non-type-safe implementations.

- **Operations:** Primary operations are Enqueue (add) and Dequeue (remove).

### Common Methods

- `Enqueue(T item)`: Adds an item to the end of the queue.

- `Dequeue()`: Removes and returns the item at the front of the queue.

- `Peek()`: Returns the item at the front without removing it.

- `Contains(T item)`: Determines whether the queue contains a specific item.

- `Clear()`: Removes all items from the queue.

### Example Usage

```csharp

using System;
```

```
using System.Collections.Generic;

public class QueueExample
{
    public static void Main()
    {
        Queue<int> queue = new Queue<int>();

        // Enqueuing items
        queue.Enqueue(1);
        queue.Enqueue(2);
        queue.Enqueue(3);

        // Peeking at the front item
        Console.WriteLine(queue.Peek()); // Output: 1

        // Dequeuing items
        Console.WriteLine(queue.Dequeue()); //
Output: 1
        Console.WriteLine(queue.Dequeue()); //
Output: 2
```

```
        // Checking if queue contains an item
        if (queue.Contains(3))
        {
            Console.WriteLine("Queue contains 3");
        }

        // Iterating through the queue
        foreach (var item in queue)
        {
            Console.WriteLine(item);
        }
    }
}
```

### When to Use

- When you need to process items in the order they were added.
- Suitable for scenarios like task scheduling, breadth-first search algorithms, and buffering data.

### Advantages

- **Orderly Processing:** Ensures elements are processed in a predictable order.

- **Thread Safety:** Can be made thread-safe using concurrent collections like `ConcurrentQueue<T>`.

### Drawbacks

- **Limited Access:** Can only access elements at the front and end directly.

- **No Random Access:** Cannot access elements by index.

---

## 6. LinkedList

### Overview

- **Namespace:** `System.Collections.Generic`

- **Description:** `LinkedList<T>` is a generic collection of nodes where each node contains a value and references to the next and/or previous nodes. It

allows efficient insertions and deletions at any position.

### Key Features

- **Doubly Linked:** Each node points to both the next and previous nodes.

- **Efficient Insertions/Deletions:** Adding or removing elements is efficient, especially in the middle of the list.

- **Sequential Access:** Accessing elements by position requires traversal from the beginning or end.

### Common Methods and Properties

- `AddFirst(T value)`: Adds an item at the start of the list.

- `AddLast(T value)`: Adds an item at the end of the list.

- `AddBefore(LinkedListNode<T> node, T value)`: Adds an item before the specified node.

- `AddAfter(LinkedListNode<T> node, T value)`: Adds an item after the specified node.

- `Remove(T value)`: Removes the first occurrence of the specified value.
- `RemoveFirst()`: Removes the first node.
- `RemoveLast()`: Removes the last node.
- `First`: Gets the first node.
- `Last`: Gets the last node.

### Example Usage

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

public class LinkedListExample
{
    public static void Main()
    {
        LinkedList<string> linkedList = new LinkedList<string>();

        // Adding items
```

```
        linkedList.AddLast("First");
        linkedList.AddLast("Second");
        linkedList.AddLast("Third");

        // Adding at specific positions
        LinkedListNode<string> secondNode = linkedList.Find("Second");
        linkedList.AddBefore(secondNode, "Inserted Before Second");
        linkedList.AddAfter(secondNode, "Inserted After Second");

        // Removing items
        linkedList.Remove("Third");

        // Iterating through the LinkedList
        foreach (var item in linkedList)
        {
            Console.WriteLine(item);
        }
    }
```

}
```

### When to Use

- When your application requires frequent insertions and deletions in the middle of the collection.

- Suitable for implementing queues, stacks, and other complex data structures.

- When the size of the collection changes frequently.

### Advantages

- **Efficient Modifications:** Insertions and deletions are O(1) operations when the node is known.

- **Flexibility:** Can easily grow and shrink dynamically.

### Drawbacks

- **Memory Overhead:** Each node requires additional memory for pointers.

- **Access Time:** Accessing elements by index is O(n), slower compared to arrays or lists.
```

---

## 7. BinaryTree

### Overview

- **Namespace:** Custom implementation or use `System.Collections.Generic.SortedSet<T>` as a basic binary search tree.

- **Description:** A `BinaryTree` is a tree data structure where each node has at most two children referred to as the left child and the right child. It's commonly used for efficient searching and sorting.

### Key Features

- **Hierarchical Structure:** Organizes data in a hierarchical manner.

- **Binary Search Tree Property:** For each node, all elements in the left subtree are less than the node, and all elements in the right subtree are greater.

- **Efficient Operations:** Allows efficient search, insert, and delete operations, typically in O(log n) time.

### Common Operations

- **Insertion:** Adding a new node while maintaining the BST property.

- **Deletion:** Removing a node and reorganizing the tree.

- **Search:** Finding a node with a specific value.

- **Traversal:** Visiting all nodes in a specific order (In-order, Pre-order, Post-order).

### Example Implementation

```csharp
using System;

public class BinarySearchTreeNode<T> where T : IComparable<T>
{
    public T Value;
    public BinarySearchTreeNode<T> Left;
    public BinarySearchTreeNode<T> Right;
```

```
    public BinarySearchTreeNode(T value)
    {
        Value = value;
    }

    public void Insert(T newValue)
    {
        if (newValue.CompareTo(Value) < 0)
        {
            if (Left == null)
                Left = new
BinarySearchTreeNode<T>(newValue);
            else
                Left.Insert(newValue);
        }
        else
        {
            if (Right == null)
                Right = new
BinarySearchTreeNode<T>(newValue);
            else
```

```
            Right.Insert(newValue);
        }
    }

    public bool Contains(T searchValue)
    {
        if (searchValue.CompareTo(Value) == 0)
            return true;
        else if (searchValue.CompareTo(Value) < 0)
            return Left != null &&
Left.Contains(searchValue);
        else
            return Right != null &&
Right.Contains(searchValue);
    }

    public void InOrderTraversal(Action<T> action)
    {
        Left?.InOrderTraversal(action);
        action(Value);
        Right?.InOrderTraversal(action);
```

```
    }
}

public class BinaryTreeExample
{
    public static void Main()
    {
        var bst = new BinarySearchTreeNode<int>(50);
        bst.Insert(30);
        bst.Insert(70);
        bst.Insert(20);
        bst.Insert(40);
        bst.Insert(60);
        bst.Insert(80);

        // Search for a value
        Console.WriteLine(bst.Contains(40)); // Output: True
        Console.WriteLine(bst.Contains(25)); // Output: False
```

```
        // In-order traversal
        bst.InOrderTraversal(value => Console.WriteLine(value));
        // Output: 20 30 40 50 60 70 80
    }
}
```

### When to Use

- When you need to maintain a dynamic dataset with quick search, insert, and delete operations.
- Suitable for implementing sorted collections, expression parsing, and decision-making processes.

### Advantages

- **Efficient Searching:** Provides faster search compared to linear data structures.
- **Sorted Data:** In-order traversal retrieves data in a sorted manner.

### Drawbacks

- **Balance Issues:** Performance degrades to O(n) if the tree becomes unbalanced.

- **Complexity:** Implementation and maintenance can be complex.

- **Memory Overhead:** Requires additional memory for tree nodes.

### Balanced Trees

- **AVL Trees and Red-Black Trees:** Self-balancing binary search trees that maintain O(log n) operations by automatically balancing themselves during insertions and deletions.

---

## 8. IEnumerable, IComparable, and IComparer Interfaces

### 8.1 IEnumerable Interface

#### Overview

- **Namespace:** `System.Collections` (non-generic), `System.Collections.Generic` (generic)

- **Description:** The `IEnumerable` interface exposes an enumerator that supports simple iteration over a non-generic or generic collection.

#### Key Features

- **Iteration Support:** Allows the use of `foreach` loops to traverse collections.
- **Extension Methods:** Enables LINQ query operations on collections.

#### Members

- `GetEnumerator()`: Returns an enumerator that iterates through the collection.

#### Example Usage

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

public class CustomCollection<T> : IEnumerable<T>
```

```
{
    private List<T> items = new List<T>();

    public void Add(T item) => items.Add(item);

    public IEnumerator<T> GetEnumerator() =>
items.GetEnumerator();

    IEnumerator IEnumerable.GetEnumerator() =>
GetEnumerator();
}

public class IEnumerableExample
{
    public static void Main()
    {
        var collection = new CustomCollection<int>();
        collection.Add(1);
        collection.Add(2);
        collection.Add(3);
```

```
        foreach (var item in collection)
        {
            Console.WriteLine(item);
        }
    }
}
```

#### When to Use

- When creating custom collections that need to be iterable using `foreach`.
- To enable LINQ queries on custom collections.

### 8.2 IComparable Interface

#### Overview

- **Namespace:** `System`
- **Description:** The `IComparable` interface defines a generalized type-specific comparison method that a value type or class implements to order or sort its instances.

#### Key Features

- **Comparison Support:** Enables sorting of objects by defining a default comparison method.

- **Generic and Non-generic Versions:** `IComparable` and `IComparable<T>` for type safety.

#### Members

- `CompareTo(object obj)`: Compares the current instance with another object of the same type.

#### Example Usage

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

public class Person : IComparable<Person>
{
    public string Name { get; set; }
    public int Age { get; set; }
```

```
    public int CompareTo(Person other)
    {
        return Age.CompareTo(other.Age);
    }
}

public class IComparableExample
{
    public static void Main()
    {
        var people = new List<Person>
        {
            new Person { Name = "Alice", Age = 30 },
            new Person { Name = "Bob", Age = 25 },
            new Person { Name = "Charlie", Age = 35 }
        };

        people.Sort();

        foreach (var person in people)
        {
```

```
            Console.WriteLine($"{person.Name}: {person.Age}");
        }
    }
}
```

#### When to Use

- When you need to define a natural ordering for your objects.
- Enables sorting mechanisms like `List<T>.Sort()` to function correctly.

### 8.3 IComparer Interface

#### Overview

- **Namespace:** `System.Collections` (non-generic), `System.Collections.Generic` (generic)
- **Description:** The `IComparer` interface defines a method that compares two objects. It allows custom comparison logic outside of the objects being compared.

#### Key Features

- **Custom Comparison:** Enables defining multiple ways to compare objects.

- **Generic and Non-generic Versions:** `IComparer` and `IComparer<T>` for type safety.

#### Members

- `Compare(object x, object y)`: Compares two objects and returns an indication of their relative sort order.

#### Example Usage

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

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
```

```
}

public class NameComparer : IComparer<Person>
{
    public int Compare(Person x, Person y)
    {
        return string.Compare(x.Name, y.Name);
    }
}

public class IComparerExample
{
    public static void Main()
    {
        var people = new List<Person>
        {
            new Person { Name = "Charlie", Age = 35 },
            new Person { Name = "Alice", Age = 30 },
            new Person { Name = "Bob", Age = 25 }
        };
```

```
        people.Sort(new NameComparer());

        foreach (var person in people)
        {
            Console.WriteLine($"{person.Name}: {person.Age}");
        }
    }
}
```

#### When to Use

- When you need multiple ways to compare and sort objects.
- Useful for sorting collections based on different criteria without modifying the objects themselves.

---

## 9. Indexers

### Overview

- **Description:** Indexers allow instances of a class or struct to be indexed like arrays. They provide a natural syntax for accessing data within an object using array-like notation.

### Key Features

- **Simplified Access:** Enables accessing elements in a collection-like manner.

- **Custom Implementation:** Define how the indexing operation works internally.

- **Overloading:** Support for multiple indexers with different parameters.

### Syntax

```csharp
public return_type this[parameter_list]
{
   get { /* return value */ }
   set { /* set value */ }
}
```

```

### Example Usage

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

public class WeekDays
{
    private Dictionary<int, string> days = new
Dictionary<int, string>
    {
        {1, "Monday"},
        {2, "Tuesday"},
        {3, "Wednesday"},
        {4, "Thursday"},
        {5, "Friday"},
        {6, "Saturday"},
        {7, "Sunday"}
    };
```
```

```
    public string this[int dayNumber]
    {
        get
        {
            return days.ContainsKey(dayNumber) ?
days[dayNumber] : "Invalid Day";
        }
        set
        {
            if (days.ContainsKey(dayNumber))
            {
                days[dayNumber] = value;
            }
            else
            {
                days.Add(dayNumber, value);
            }
        }
    }
}
```

```
public class IndexerExample
{
    public static void Main()
    {
        var week = new WeekDays();

        Console.WriteLine(week[3]); // Output: Wednesday

        week[3] = "Midweek";
        Console.WriteLine(week[3]); // Output: Midweek

        Console.WriteLine(week[8]); // Output: Invalid Day
    }
}
```

### When to Use

- When you want to provide array-like access to internal data structures of a class.

- Useful for creating custom collections or data structures.

### Advantages

- **Readable Code:** Improves code readability and usability.

- **Encapsulation:** Controls access to internal data while providing a familiar interface.

### Drawbacks

- **Complexity:** Overuse can make code harder to understand.

- **Performance:** Custom implementations may not be as efficient as built-in arrays.

---

## 10. Writing Generic Classes & Methods

### 10.1 Generic Classes

#### Overview

- **Description:** Generic classes allow you to define classes with placeholders for the type of data they store or use, providing type safety and code reusability.

#### Syntax

```csharp
public class GenericClass<T>
{
    public T Data { get; set; }

    public void Display()
    {
        Console.WriteLine(Data);
    }
}
```

#### Example Usage

```csharp
using System;

public class GenericClass<T>
{
    public T Data { get; set; }

    public GenericClass(T data)
    {
        Data = data;
    }

    public void Display()
    {
        Console.WriteLine($"Data: {Data}");
    }
}

public class GenericClassExample
```

```
{
    public static void Main()
    {
        var intInstance = new GenericClass<int>(10);
        intInstance.Display(); // Output: Data: 10

        var stringInstance = new GenericClass<string>("Hello World");
        stringInstance.Display(); // Output: Data: Hello World
    }
}
```

### 10.2 Generic Methods

#### Overview

- **Description:** Generic methods define type parameters independent of the class, allowing methods to operate on different data types while maintaining type safety.

#### Syntax

```csharp
public void GenericMethod<T>(T param)
{
    Console.WriteLine(param);
}
```

#### Example Usage

```csharp
using System;

public class GenericMethodExample
{
    public void Display<T>(T value)
    {
        Console.WriteLine($"Value: {value}");
    }
```

```
    public static void Main()
    {
        var example = new GenericMethodExample();

        example.Display(100); // Output: Value: 100
        example.Display("Generic Method"); // Output: Value: Generic Method
        example.Display(3.14); // Output: Value: 3.14
    }
}
```

### Benefits of Generics

- **Type Safety:** Errors are caught at compile-time rather than runtime.
- **Performance:** Eliminates the need for boxing/unboxing, improving performance.
- **Code Reusability:** Write code once and reuse it for different data types.
- **Maintainability:** Reduces code duplication and simplifies maintenance.

### When to Use

- When writing classes or methods that need to work with any data type.

- For creating collection classes like lists, queues, stacks, etc.

- When the exact data type is not known until runtime.

---

## 11. Generic Constraints

### Overview

- **Description:** Generic constraints specify the requirements that a type argument must meet, allowing you to use specific functionalities within generic classes or methods.

### Types of Constraints

1. **Where T : struct** - T must be a value type.

2. **Where T : class** - T must be a reference type.

3. **Where T : new()** - T must have a parameterless constructor.

4. **Where T : BaseClass** - T must inherit from `BaseClass`.

5. **Where T : Interface** - T must implement the specified interface.

6. **Where T : U** - T must be or derive from U.

### Example Usage

```csharp
using System;

public class GenericConstraintExample<T> where T : new()
{
    public T CreateInstance()
    {
        return new T();
    }
}
```

```
public class SampleClass
{
    public void Display()
    {
        Console.WriteLine("SampleClass instance created.");
    }
}

public class Program
{
    public static void Main()
    {
        var example = new GenericConstraintExample<SampleClass>();
        var instance = example.CreateInstance();
        instance.Display(); // Output: SampleClass instance created.
    }
}
```

### Benefits

- **Enhanced Functionality:** Allows using specific methods or properties of the constrained types.

- **Type Safety:** Ensures that type arguments meet specific requirements.

- **Flexibility:** Enables more precise control over generic type behavior.

### When to Use

- When your generic code relies on certain features of the type parameter.

- To enforce that type arguments implement certain interfaces or inherit from specific classes.

- To ensure that type parameters can be instantiated within generic code.

---

## 12. Generic Delegates

### Overview

- **Description:** Generic delegates define delegate types with type parameters, allowing methods with different parameter and return types to be assigned to the same delegate type.

### Predefined Generic Delegates

- **Func<TResult>**: Represents a method that returns a value.

- **Func<T, TResult>**: Represents a method with one parameter and a return value.

- **Action<T>**: Represents a method that takes parameters but does not return a value.

- **Predicate<T>**: Represents a method that defines a set of criteria and determines whether the specified object meets those criteria.

### Custom Generic Delegate Syntax

```csharp
public delegate TOutput MyDelegate<TInput, TOutput>(TInput input);
```

### Example Usage

```csharp
using System;

public class GenericDelegateExample
{
    // Define a generic delegate
    public delegate T Transformer<T>(T arg);

    public static int Square(int x) => x * x;
    public static string Reverse(string s) => new string(s.ToCharArray().Reverse().ToArray());

    public static void Main()
    {
        Transformer<int> intTransformer = Square;
        Console.WriteLine(intTransformer(5)); // Output: 25
```

```
        Transformer<string> stringTransformer =
Reverse;
        Console.WriteLine(stringTransformer("Hello"));
// Output: olleH
    }
}
```

### Benefits

- **Flexibility:** Allows creating delegates that can work with various types.
- **Type Safety:** Ensures that methods assigned to delegates match the specified type parameters.
- **Code Reusability:** Reduces the need to define multiple delegate types for different data types.

### When to Use

- When you need delegates that operate on different types without defining multiple delegate types.
- For defining callback methods and event handlers with varying type requirements.
- In functional programming scenarios where methods are passed as arguments.

---

## 13. Generic Interfaces

### Overview

- **Description:** Generic interfaces define contracts that can work with any data type specified at implementation time, providing flexibility and type safety.

### Example Definition

```csharp
public interface IRepository<T>
{
    void Add(T item);
    T Get(int id);
    IEnumerable<T> GetAll();
}
```

### Example Implementation

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

public interface IRepository<T>
{
    void Add(T item);
    T Get(int id);
    IEnumerable<T> GetAll();
}

public class Repository<T> : IRepository<T>
{
    private readonly Dictionary<int, T> storage = new Dictionary<int, T>();
    private int currentId = 0;

    public void Add(T item)
```

```
    {
        storage.Add(currentId++, item);
    }

    public T Get(int id)
    {
        return storage.ContainsKey(id) ? storage[id] :
default;
    }

    public IEnumerable<T> GetAll()
    {
        return storage.Values;
    }
}

public class GenericInterfaceExample
{
    public static void Main()
    {
```

```
        IRepository<string> stringRepo = new
Repository<string>();
        stringRepo.Add("Item 1");
        stringRepo.Add("Item 2");

        Console.WriteLine(stringRepo.Get(0)); //
Output: Item 1

        foreach (var item in stringRepo.GetAll())
        {
            Console.WriteLine(item);
        }
    }
}
```

### Benefits

- **Flexibility:** Allows implementations to specify concrete types as needed.
- **Type Safety:** Enforces that implementations and consumers use the correct types.

- **Reusability:** Promotes code reuse across different types and scenarios.

### When to Use

- When defining contracts that operate on various data types.

- For creating collection interfaces like `IEnumerable<T>`, `IComparable<T>`, etc.

- In designing scalable and maintainable systems where components interact through well-defined contracts.

---

## 14. Generic Collection Classes

### Overview

- **Description:** Generic collection classes provide strongly-typed collections that improve performance and type safety over non-generic collections.

### Common Generic Collections

1. **List<T>**
   - **Usage:** Dynamic array, allows indexed access.
   - **Features:** Efficient for adding/removing at the end, supports sorting, searching.
2. **Dictionary<TKey, TValue>**
   - **Usage:** Key-value pairs with fast lookup.
   - **Features:** Efficient insertion, removal, and retrieval by key.
3. **Queue<T>**
   - **Usage:** FIFO collection.
   - **Features:** Efficient enqueuing and dequeuing.
4. **Stack<T>**
   - **Usage:** LIFO collection.
   - **Features:** Efficient push and pop operations.
5. **HashSet<T>**
   - **Usage:** Collection of unique elements.
   - **Features:** Efficient operations, no duplicates allowed.
6. **SortedList<TKey, TValue>**
   - **Usage:** Sorted key-value pairs.
   - **Features:** Maintains elements in sorted order.

7. **SortedSet<T>**
   - **Usage:** Sorted collection of unique elements.
   - **Features:** Combines features of `HashSet` and sorted collections.

### Example Usage of List<T>

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

public class GenericCollectionExample
{
    public static void Main()
    {
        List<int> numbers = new List<int> {1, 3, 5, 7};

        numbers.Add(9);
        numbers.Insert(2, 4);
```

```
        Console.WriteLine("Count: " + numbers.Count);
// Output: Count: 6

        numbers.Remove(3);

        foreach (var number in numbers)
        {
            Console.WriteLine(number);
        }
    }
}
```

### Benefits

- **Type Safety:** Compile-time type checking prevents errors.
- **Performance:** Eliminates boxing/unboxing, reduces runtime overhead.
- **Rich Functionality:** Provides extensive methods for manipulation and querying.
- **Consistency:** Standardized behavior across different collection types.

### When to Use

- Prefer generic collections over non-generic ones in modern applications.

- When you need collections that are efficient, type-safe, and easy to use.

- Suitable for most data storage and manipulation tasks in applications.

---

# Conclusion

Understanding and effectively utilizing various collection classes, interfaces, and generic programming concepts is fundamental for building efficient and robust applications. Collections provide versatile ways to store and manage data, while generics enhance code reusability, performance, and type safety. By mastering these topics, developers can write cleaner, more efficient, and maintainable code tailored to diverse programming needs.