1. **ArrayList Basics**

Exercise: Create an `ArrayList` that stores integers. Add 5 integers, remove the 3rd one, and print all elements.

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


class Program
{
    static void Main()
    {
        ArrayList numbers = new ArrayList() { 1, 2, 3, 4, 5 };
        numbers.RemoveAt(2);
        foreach (var number in numbers)
        {
            Console.WriteLine(number);
        }
    }
}
```

```

### 2. **HashTable Basic Operations**

**Exercise**: Create a `Hashtable` that stores key-value pairs of string and int. Add, update, and remove an entry, then print all entries.

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

class Program
{
    static void Main()
    {
        Hashtable hashtable = new Hashtable();
        hashtable.Add("Apple", 10);
        hashtable.Add("Banana", 20);
        hashtable["Apple"] = 15; // Update value
        hashtable.Remove("Banana");

        foreach (DictionaryEntry entry in hashtable)
```
```

```
        {
            Console.WriteLine($"{entry.Key}: {entry.Value}");
        }
    }
}
```

3. **Dictionary Creation and Access**

Exercise: Create a `Dictionary<string, string>` for storing country names and their capitals. Add and retrieve an entry.

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

class Program
{
    static void Main()
    {
        Dictionary<string, string> capitals = new Dictionary<string, string>();
```

```
        capitals.Add("USA", "Washington, D.C.");
        capitals.Add("France", "Paris");

        Console.WriteLine(capitals["France"]);
    }
}
```

4. **Stack Push and Pop**

Exercise: Implement a basic stack using the `Stack` class. Push 3 items onto the stack, then pop and print each item.

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

class Program
{
    static void Main()
    {
        Stack stack = new Stack();
```

```
        stack.Push(1);
        stack.Push(2);
        stack.Push(3);

        while (stack.Count > 0)
        {
            Console.WriteLine(stack.Pop());
        }
    }
}
```

5. **Queue Enqueue and Dequeue**

Exercise: Create a queue of strings using `Queue`. Enqueue 3 items and dequeue them, printing each one.

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

class Program
```

```
{
    static void Main()
    {
        Queue queue = new Queue();
        queue.Enqueue("First");
        queue.Enqueue("Second");
        queue.Enqueue("Third");

        while (queue.Count > 0)
        {
            Console.WriteLine(queue.Dequeue());
        }
    }
}
```

6. **LinkedList Basic Operations**

Exercise: Create a `LinkedList<int>` and perform insertion and deletion of elements at different positions.

```csharp
```

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

class Program
{
    static void Main()
    {
        LinkedList<int> list = new LinkedList<int>();
        list.AddLast(10);
        list.AddLast(20);
        list.AddFirst(5);
        list.Remove(10);

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

7. **Binary Tree Implementation**

Exercise: Implement a basic binary tree and traverse it in-order.

```csharp
using System;

class Node
{
    public int Data;
    public Node Left;
    public Node Right;

    public Node(int data)
    {
        Data = data;
        Left = Right = null;
    }
}

class BinaryTree
{
```

```
    public Node Root;

    public void InOrderTraversal(Node node)
    {
        if (node == null)
            return;

        InOrderTraversal(node.Left);
        Console.WriteLine(node.Data);
        InOrderTraversal(node.Right);
    }
}

class Program
{
    static void Main()
    {
        BinaryTree tree = new BinaryTree();
        tree.Root = new Node(1);
        tree.Root.Left = new Node(2);
        tree.Root.Right = new Node(3);
```

```
        tree.InOrderTraversal(tree.Root);
    }
}
```

8. **IEnumerable Implementation**

Exercise: Implement a class that implements `IEnumerable<int>` and returns a list of integers.

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

class NumberCollection : IEnumerable<int>
{
    private List<int> _numbers = new List<int> { 1, 2, 3, 4, 5 };

    public IEnumerator<int> GetEnumerator()
    {
```

```
        return _numbers.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

class Program
{
    static void Main()
    {
        var numbers = new NumberCollection();
        foreach (var number in numbers)
        {
            Console.WriteLine(number);
        }
    }
}
```

9. **IComparable Interface**

Exercise: Implement the `IComparable` interface for a class and sort a list of its objects.

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

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

    public int CompareTo(Student other)
    {
        return this.Age.CompareTo(other.Age);
    }
}

class Program
{
```

```
    static void Main()
    {
        List<Student> students = new List<Student>
        {
            new Student { Name = "Alice", Age = 22 },
            new Student { Name = "Bob", Age = 20 },
            new Student { Name = "Charlie", Age = 23 }
        };

        students.Sort();

        foreach (var student in students)
        {
            Console.WriteLine($"{student.Name}, {student.Age}");
        }
    }
}
```

10. **IComparer Interface**

Exercise: Implement a custom comparer using `IComparer` to sort a list of strings by length.

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

class StringLengthComparer : IComparer<string>
{
    public int Compare(string x, string y)
    {
        return x.Length.CompareTo(y.Length);
    }
}

class Program
{
    static void Main()
    {
        List<string> words = new List<string> { "apple", "banana", "cherry" };
        words.Sort(new StringLengthComparer());
```

```
        foreach (var word in words)
        {
            Console.WriteLine(word);
        }
    }
}
```

11. **Indexers in C#**

Exercise: Create a class with an indexer to access elements in a private array.

```csharp
using System;

class IndexerExample
{
    private int[] _array = new int[5];

    public int this[int index]
    {
```

```
        get { return _array[index]; }
        set { _array[index] = value; }
    }
}

class Program
{
    static void Main()
    {
        var example = new IndexerExample();
        example[0] = 10;
        example[1] = 20;
        Console.WriteLine(example[0]);
        Console.WriteLine(example[1]);
    }
}
```

12. **Generic Class Creation**

Exercise: Create a generic class `Box<T>` that stores an object of type `T`.

```csharp
using System;

class Box<T>
{
    public T Value { get; set; }

    public Box(T value)
    {
        Value = value;
    }

    public void Display()
    {
        Console.WriteLine(Value);
    }
}

class Program
{
    static void Main()
```

```
    {
        var intBox = new Box<int>(123);
        intBox.Display();

        var stringBox = new Box<string>("Hello");
        stringBox.Display();
    }
}
```

13. **Generic Method Creation**

Exercise: Implement a generic method that swaps two elements.

```csharp
using System;

class Program
{
    static void Swap<T>(ref T a, ref T b)
    {
        T temp = a;
```

```
        a = b;
        b = temp;
    }

    static void Main()
    {
        int x = 1, y = 2;
        Swap(ref x, ref y);
        Console.WriteLine($"x: {x}, y: {y}");

        string s1 = "Hello", s2 = "World";
        Swap(ref s1, ref s2);
        Console.WriteLine($"s1: {s1}, s2: {s2}");
    }
}
```

14. **Generic Constraints**

Exercise: Create a generic class `Calculator<T>` with a constraint that `T` must be a `struct`. Implement a method

to add two `T` values.

```csharp
using System;

class Calculator<T> where T : struct
{
    public T Add(T a, T b)
    {
        dynamic x = a;
        dynamic y = b;
        return x + y;
    }
}

class Program
{
    static void Main()
    {
        var calculator = new Calculator<int>();
        Console.WriteLine(calculator.Add(3, 5));
```

```
        var doubleCalculator = new Calculator<double>();
        Console.WriteLine(doubleCalculator.Add(2.5, 4.5));
    }
}
```

15. **Generic Delegate**

Exercise: Create a generic delegate and use it to pass different types of methods.

```csharp
using System;

delegate T Operation<T>(T a, T b);

class Program
{
    static int Add(int x, int y) => x + y;
    static string Concat(string x, string y) => x + y;
```

```
    static void Main()
    {
        Operation<int> intOperation = Add;
        Console.WriteLine(intOperation(5, 10));

        Operation<string> stringOperation = Concat;
        Console.WriteLine(stringOperation("Hello, ", "World!"));
    }
}
```

16. **Generic Interface Implementation**

Exercise: Implement a generic interface and create a class that uses it.

```csharp
using System;

interface IRepository<T>
{
```

```
    void Add(T item);
    T Get(int id);
}

class Repository<T> : IRepository<T>
{
    private T[] items = new T[10];
    public void Add(T item)
    {
        items[0] = item;
    }

    public T Get(int id)
    {
        return items[id];
    }
}

class Program
{
    static void Main()
```

```
    {
        var repo = new Repository<string>();
        repo.Add("Hello");
        Console.WriteLine(repo.Get(0));
    }
}
```

17. **Generic Collection: List<T>**

Exercise: Create a `List<int>` and perform various operations like adding, removing, and finding elements.

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

class Program
{
    static void Main()
    {
```

```
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
        numbers.Add(6);
        numbers.Remove(3);

        int index = numbers.FindIndex(x => x == 5);
        Console.WriteLine($"Index of 5: {index}");

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

18. **Generic Collection: Dictionary<TKey, TValue>**

Exercise: Create a `Dictionary<int, string>` and perform operations like adding, updating, and retrieving elements.

```csharp
```

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

class Program
{
    static void Main()
    {
        Dictionary<int, string> employees = new
Dictionary<int, string>();
        employees.Add(1, "Alice");
        employees.Add(2, "Bob");

        employees[2] = "Charlie"; // Update entry
        Console.WriteLine($"Employee 2:
{employees[2]}");

        foreach (var employee in employees)
        {
            Console.WriteLine($"{employee.Key}:
{employee.Value}");
        }
```

```
    }
}
```

19. **Generic Collection: Stack<T>**

Exercise: Create a `Stack<string>` and perform push, pop, and peek operations.

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

class Program
{
    static void Main()
    {
        Stack<string> stack = new Stack<string>();
        stack.Push("First");
        stack.Push("Second");

        Console.WriteLine(stack.Peek()); // Should print "Second"
```

```
        Console.WriteLine(stack.Pop());  // Should print "Second"
        Console.WriteLine(stack.Pop());  // Should print "First"
    }
}
```

20. **Generic Collection: Queue<T>**

Exercise: Create a `Queue<int>` and perform enqueue and dequeue operations.

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

class Program
{
    static void Main()
    {
        Queue<int> queue = new Queue<int>();
        queue.Enqueue(10);
```

```
        queue.Enqueue(20);
        queue.Enqueue(30);

        Console.WriteLine(queue.Dequeue()); // Should print 10
        Console.WriteLine(queue.Dequeue()); // Should print 20
    }
}
```

21. **Generic Collection: LinkedList<T>**

Exercise: Create a `LinkedList<string>` and perform operations like adding nodes at the beginning, end, and in the middle.

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

class Program
{
    static void Main()
```

```
    {
        LinkedList<string> linkedList = new LinkedList<string>();
        linkedList.AddLast("End");
        linkedList.AddFirst("Start");
        linkedList.AddAfter(linkedList.First, "Middle");

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

22. **Binary Search Tree Implementation**

Exercise: Implement a binary search tree and provide methods for inserting and searching elements.

```csharp
using System;
```

```
class Node
{
    public int Data;
    public Node Left;
    public Node Right;

    public Node(int data)
    {
        Data = data;
        Left = Right = null;
    }
}

class BinarySearchTree
{
    public Node Root;

    public void Insert(int data)
    {
        Root = InsertRec(Root, data);
```

```
}

private Node InsertRec(Node root, int data)
{
   if (root == null)
   {
      root = new Node(data);
      return root;
   }

   if (data < root.Data)
      root.Left = InsertRec(root.Left, data);
   else if (data > root.Data)
      root.Right = InsertRec(root.Right, data);

   return root;
}

public bool Search(int data)
{
   return SearchRec(Root, data) != null;
```

```
    }

    private Node SearchRec(Node root, int data)
    {
        if (root == null || root.Data == data)
            return root;

        if (data < root.Data)
            return SearchRec(root.Left, data);

        return SearchRec(root.Right, data);
    }
}

class Program
{
    static void Main()
    {
        BinarySearchTree bst = new
BinarySearchTree();
        bst.Insert(50);
```

```
        bst.Insert(30);
        bst.Insert(70);

        Console.WriteLine(bst.Search(30)); // Should return true
        Console.WriteLine(bst.Search(100)); // Should return false
    }
}
```

23. **Using Indexers with Generics**

Exercise: Create a generic class with an indexer to manage a collection of elements.

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

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

```
    public T this[int index]
    {
        get { return items[index]; }
        set { items.Insert(index, value); }
    }

    public void Add(T item)
    {
        items.Add(item);
    }
}

class Program
{
    static void Main()
    {
        var collection = new Collection<int>();
        collection.Add(1);
        collection.Add(2);
```

```
        Console.WriteLine(collection[0]); // Should print 1
        collection[1] = 10;
        Console.WriteLine(collection[1]); // Should print 10
    }
}
```

24. **Advanced Generics: Multiple Type Parameters**

Exercise: Create a generic class with two type parameters and use it in a program.

```csharp
using System;

class Pair<T1, T2>
{
    public T1 First { get; set; }
    public T2 Second { get; set; }

    public Pair(T1 first, T2 second)
```

```
    {
        First = first;
        Second = second;
    }

    public void Display()
    {
        Console.WriteLine($"First: {First}, Second: {Second}");
    }
}

class Program
{
    static void Main()
    {
        var pair = new Pair<string, int>("Hello", 123);
        pair.Display();
    }
}
```

25. **Generic Constraints: Reference and Value Types**

Exercise: Create a generic method that accepts only reference types and another one that accepts only value types.

```csharp
using System;


class Program
{
    static void PrintReferenceType<T>(T value) where T : class
    {
        Console.WriteLine($"Reference type: {value}");
    }


    static void PrintValueType<T>(T value) where T : struct
    {
        Console.WriteLine($"Value type: {value}");
    }
```

```
    static void Main()
    {
        PrintReferenceType("Hello");
        PrintValueType(123);
    }
}
```

26. **Lambda Expressions with Generics**

Exercise: Create a generic method that accepts a lambda expression as a parameter and applies it to a list of items.

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

class Program
{
    static
```

```
 void ApplyOperation<T>(List<T> items, Func<T, T> operation)
    {
        for (int i = 0; i < items.Count; i++)
        {
            items[i] = operation(items[i]);
        }
    }

    static void Main()
    {
        var numbers = new List<int> { 1, 2, 3, 4, 5 };
        ApplyOperation(numbers, x => x * 2);

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

27. **Generic Interface with Constraints**

Exercise: Create a generic interface with a constraint and implement it in a class.

```csharp
using System;

interface IStorable<T> where T : class
{
    void Store(T item);
    T Retrieve();
}

class Storage<T> : IStorable<T> where T : class
{
    private T _item;

    public void Store(T item)
    {
        _item = item;
    }
```

```
    public T Retrieve()
    {
        return _item;
    }
}

class Program
{
    static void Main()
    {
        var storage = new Storage<string>();
        storage.Store("Hello World");
        Console.WriteLine(storage.Retrieve());
    }
}
```

28. **Extension Methods with Generics**

Exercise: Create an extension method for a generic collection that returns the first item or a default value.

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


static class Extensions
{
    public static T FirstOrDefault<T>(this List<T> list)
    {
        return list.Count > 0 ? list[0] : default(T);
    }
}


class Program
{
    static void Main()
    {
        var numbers = new List<int> { 1, 2, 3 };
```

```
        Console.WriteLine(numbers.FirstOrDefault());
// Should print 1

        var emptyList = new List<int>();

Console.WriteLine(emptyList.FirstOrDefault()); //
Should print 0 (default int value)
    }
  }
  ```

29. **Using Action and Func Delegates**
  Exercise: Use `Action` and `Func` delegates to
create methods that process and return data.
  ```csharp
  using System;

  class Program
  {
    static void Process(Action<string> action)
    {
      action("Hello");
```

```
    }

    static int Calculate(Func<int, int, int> func)
    {
        return func(2, 3);
    }

    static void Main()
    {
        Process(msg => Console.WriteLine(msg));
        int result = Calculate((x, y) => x * y);
        Console.WriteLine(result);
    }
}
```

30. **Advanced Binary Tree Traversals**

Exercise: Extend the binary tree implementation to include pre-order and post-order traversal methods.

```csharp
```

```
using System;

class Node
{
    public int Data;
    public Node Left;
    public Node Right;

    public Node(int data)
    {
        Data = data;
        Left = Right = null;
    }
}

class BinaryTree
{
    public Node Root;

    public void PreOrderTraversal(Node node)
    {
```

```
        if (node == null)
            return;

        Console.WriteLine(node.Data);
        PreOrderTraversal(node.Left);
        PreOrderTraversal(node.Right);
    }

    public void PostOrderTraversal(Node node)
    {
        if (node == null)
            return;

        PostOrderTraversal(node.Left);
        PostOrderTraversal(node.Right);
        Console.WriteLine(node.Data);
    }
}

class Program
{
```

```
    static void Main()
    {
        BinaryTree tree = new BinaryTree();
        tree.Root = new Node(1);
        tree.Root.Left = new Node(2);
        tree.Root.Right = new Node(3);

        Console.WriteLine("Pre-Order Traversal:");
        tree.PreOrderTraversal(tree.Root);

        Console.WriteLine("Post-Order Traversal:");
        tree.PostOrderTraversal(tree.Root);
    }
}
```