Generics in C# – Learning Material

1. Introduction

Generics in C# allow us to create classes, interfaces, methods, and delegates with placeholders for types. They help us write flexible, reusable, and type-safe code.

Analogy: Think of a lunch box — it can store sandwiches, fruits, or snacks. Similarly, a generic lets you decide the data type when using it.

2. Why Use Generics?

- Type Safety – No need for type casting, reducing runtime errors.
- Code Reusability – Write once, use for any data type.
- Performance – Avoids Boxing/Unboxing overhead.
- Readability – Cleaner and more maintainable code.

📌 Boxing & Unboxing

Before generics, collections like ArrayList stored items as object. When you store a value type (like int) inside an object, Boxing happens (value wrapped in a reference type). When you retrieve it back, Unboxing happens.

Drawbacks:
- Slower performance.
- Possible runtime errors if type mismatches.

Example without Generics (Boxing & Unboxing):

```
using System;
using System.Collections;

class Program
{
   static void Main()
   {
      ArrayList list = new ArrayList();
      int num = 10;

      // Boxing: int → object
      list.Add(num);

      // Unboxing: object → int
      int value = (int)list[0];

      Console.WriteLine("Value: " + value);
   }
}
```

Example with Generics (No Boxing/Unboxing):

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

```
class Program
{
   static void Main()
   {
      List<int> list = new List<int>();
      int num = 10;

      // No Boxing
      list.Add(num);

      // No Unboxing
      int value = list[0];

      Console.WriteLine("Value: " + value);
   }
}
```

3. Syntax of Generics

```
// Generic Class
public class MyClass<T>
{
   public T Data { get; set; }
}

// Generic Method
public T Display<T>(T value)
{
   return value;
}
```

4. Example – Generic Swap Method

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

class Program
{
   static void Main()
   {
      int x = 5, y = 10;
```

```
        Console.WriteLine($"Before Swap: x = {x}, y = {y}");
        Utility.Swap(ref x, ref y);
        Console.WriteLine($"After Swap: x = {x}, y = {y}");

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

5. Generics with Constraints

```
public class DataProcessor<T> where T : int,string
{
    public void Process(T data)
    {
        Console.WriteLine("Processing " + data.ToString());
    }
}
```

Practice Question – Generic Finder

Problem:
Write a generic class Finder<T> that:
- Takes an array of type T
- Has a method FindElement(T element) which returns:
  - true if the element is present in the array
  - false otherwise

Requirements:
- Use Equals() method for comparison so it works for any data type.

Test with:
- An int array
- A string array