Delegates: A delegate is a pointer to a method. That means, a delegate holds the address of a method which can be called using that delegate.

A delegate is also called type safe pointer.
A delegate is a way to call a method indirectly using a reference.

Syntax:

```
delegate returntype DelegateName(parameters);
```

Steps to use a Delegate

step1: Declare a Delegate that matches the method signature you want to point to.

```
delegate void mydelegate(string name);
```

step2: Create a method that matches the Delegate's signature

```
static void ~~Rajesh~~Name(string name)
{
    Console.WriteLine("Hello "+name);
}
```

step3: Instantiate the Delegate

```
mydelegate md = new mydelegate(~~Rajesh~~Name);
```

step4: Call the method using delegate

```
md("Rajesh");
```

```
using System;
class program{
delegate void mydelegate(string name);
 static void ~~my~~Name(string name)
  {Console.WriteLine("Hello "+name);
 }
```

```
static void main()
{
mydelegate md =
 new mydelegate
      (Name);
md("Rajesh");
}
```

Example

Delete for Different Arithmetic operation

```
Using System
Class program
{
delegate int Mathoperation(int a, int b);
 static int Add (int a, int b)
 {
  return a+b;
 }
 static int mul (int a ,int b)
  {
  return a*b;
  }
  Static void main( )
  {
   mathoperation mo = new Mathoperation ( Add);
    Mo(5,3);
    Mathoperation mo1 = new Mathoperation (mul);
    Mo1(5,3);
    Console.WriteLine ("Add:" +Mo(5,3));
    Console.WriteLine ("multiply:" +Mo1(5,3);
  } }
```

output: Add: 8
multiply: 15

Key Points about Delegate

- A delegate stores a reference to a method.
- Delegate ensures type safety (the method signature must match).
- used in event handling callbacks, & functional programming.

Program Example :-

```
delegate void MyDelegate (string message);
Class Program
{
static void DisplayMessage (string msg)
{
  Console.WriteLine (msg);
}
 static void Main ()
 {
   MyDelegate del = new MyDelegate (Displaymessage);
   del ("HelloWorld");
 }
}
```

Q. Delegate program example with no parameter

```
using system;
class program
{
  delegate void delegate_Name ( ); //declare delegate
  static void Main ( )
  {
    delegate_Name d = new delegate_Name (rajesh);
  } d( ); // invoke the delegate
  static void rajesh ( )
  { Console.WriteLine ("Delegate first program");
  }
}
```

output:

Delegate first program.

Q.2 Delegate program to Add two numbers

```
using system;
class program
{
  delegate int AddNumbers (int xc, int JJ);
  static void Main ( )
  {
    AddNumbers a = new AddNumbers (sum );
    Console.WriteLine ("Enter first Number: ");
    int num1 = Convert.ToInt32 (Console.ReadLine( ));
    Console.WriteLine ("Enter second Number: ");
    int num2 = Convert.ToInt32 (Console.ReadLine( ));
    int result = a[redacted] (num1, num2);
```

```
Console.WriteLine("The sum is: "+result);
}
static int sum(int x, int y) // int a, int b
{
    return x+y;          // return a+b;
}
}
```

Note

method that matches the delegate signature but the parameter names can be changed with the same type.

Output:
```
Enter first Number:
5
Enter second Number:
7
The sum is 12.
```

What is Delegate?

A delegate is a type-safe function pointer. It holds a reference to a method with a specific signature (return type and parameter).

Delegates are used to pass methods as argument to other methods, enabling callback mechanisms and event handling.

Syntax:

```
delegate returnType DelegateName(parameters);
```

3.2 Lamda Expression and its implementation

An anonymous function used to create delegates is called Lamda Expressions in c#. To create local functions to be passed as an argument, a lamda expression can be used.

Syntax:

```
(input-parameters) => expression;
where => is a lamda operator
```

Example:

```
using System;
namespace LamdaExpressionExample
{
  class program
  {
    delegate int cube(int n);
    static void main(string[] args)
    {
      cube getcube = DC => x*x*x;
      int res = getcube(10);
      Console.WriteLine("cube: "+res);
    }
  }
}
```

output: cube: 1000

- Built-in delegates like func, Action, and predicate work seamlessly with Lamda Expressions.
- Linq methods like where, select, and orderBy use Lamda expressions for querying Collections.
- Lamda Expressions are used in functional programming, events and Collection methods.

1. func Delegate :- Represents a method that takes input parameters and returns a value. The last parameter is the return-type. fun Delegate can take 0 to 16 input parameters.

Syntax:

func< PT1, PT2 ..., RPT> Delegate_variable = (parameters list) => Lamda Expression;

eg func< int, int... int> Sum = (x,y) => x+y;

Programple Example

```
using System;
namespace funLamdaExample
{
  class program
  {
    static void Main(string[] args)
    {
      func<int, int, int> add = (a, b) => a+b;
      Console.WriteLine("Addition: "+add(10,5));
      func<string, string> msg = name => "Hello "+
       name+"!";
      Console.WriteLine(msg("Rajesh"));
      func<string> getmessage = () => "Hello Lamda";
      Console.WriteLine(getmessage());
    }
  }
}
```

output: Addition : 15
Hello Rajesh
Hello Lamda

2. Action Delegate Represents a method that takes input parameters but does not return a value

Syntax

Action < T1, T2, --- >

↳ T1, T2 are input parameter types.

Example

```
// Action with no input parameters
Action Delegate variable_Name = ( ) => Console.
WriteLine ("HelloWorld!");
Delegatevariable_Name ( );

//Action with one input parameter
Action<string> DO = name => Console.WriteLine
("Hello" + name);
DO ("Rajesh");

// Action with multiple input parameters
Action<string, int> DO = (name, age) =>
Console.WriteLine("name and Age is:" + name + "
+age");
DO ("Rajesh", 28);
```

3. Predicate Built-in Delegate :- Represents a method that takes one input parameter and returns a boolean value (true or false).

Syntax

```
Predicate<T>
```

↳ T is input parameter type

Example

```
Predicate<int> iseven = num => num % 2 == 0;
Console.WriteLine(iseven(4)); // output: True
```

Program example :

```
using System;
class Program
{
    static void Main()
    {
        Predicate<int> isEven = num =>
        {
            bool result = num % 2 == 0;
            Console.WriteLine("num is even: " + result);
            return result;
        };
        Console.WriteLine(isEven(4));
        Console.WriteLine(isEven(5));
    }
}
```

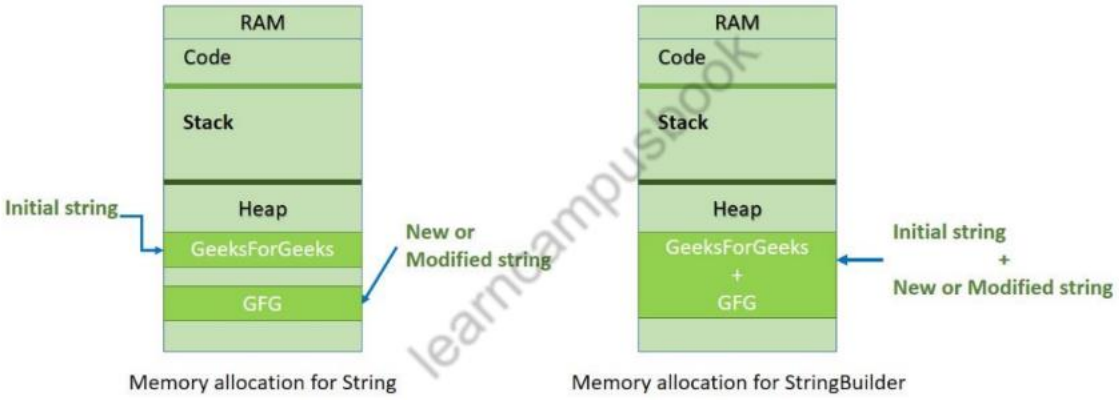
```
num is even: True
True
num is even: false
false
```

3.4 String Manipulation and String Builder in C#

In C#, strings are an essential part of programming. They are used to store and manipulate textual data. However, since strings in C# are immutable (cannot be changed once created), operations like concatenation, replacement, or modification create new string instances, which can be inefficient in terms of memory and performance.

To address these performance issues, StringBuilder is used, which provides a mutable string representation, making it more efficient for frequent string modifications.

StringBuilder is a dynamic object. It doesn't create a new object in the memory but dynamically expands the needed memory to accommodate the modified or new string.



3.4.1 Declaration and Initialization of StringBuilder

StringBuilder can be declared and initialized the same way as class, "s" is the object of StringBuilder class. Also, we can pass a string value(here "MicroInfoTech") as an argument to the constructor of StringBuilder.

```
StringBuilder s = new StringBuilder();
or
StringBuilder s = new StringBuilder("MicroInfoTech");
```

3.4.2 Defining the capacity of StringBuilder
Although the StringBuilder is a dynamic object that allows you to expand the number of characters in the string that it encapsulates, you can specify a value for the maximum number of characters that it can hold. This value is called the capacity of the StringBuilder object.

```
StringBuilder s = new StringBuilder(20);
or
StringBuilder s = new StringBuilder("MicroInfoTech ", 20);
```

1. String Manipulation in C#

String manipulation refers to performing operations on strings such as concatenation, searching, replacing, trimming, and extracting substrings.

```
using System;
class Program
{
```

```
    static void Main()
    {
        string str1 = "Hello";
        string str2 = "World";

        // String Concatenation
        string result = str1 + " " + str2;
        Console.WriteLine("Concatenation: " + result);

        // Substring Extraction
        Console.WriteLine("Substring (first 5 chars): " + result.Substring(0, 5));

        // Convert to Upper & Lower Case
        Console.WriteLine("Uppercase: " + result.ToUpper());
        Console.WriteLine("Lowercase: " + result.ToLower());

        // Replace part of the string
        Console.WriteLine("Replaced: " + result.Replace("World", "C#"));

        // Splitting a string
        string sentence = "Apple, Banana, Cherry";
        string[] words = sentence.Split(',');
        Console.WriteLine("Splitted Words:");
        foreach (string word in words)
        {
            Console.WriteLine(word.Trim()); // Trim removes extra spaces
        }
    }
}
```

```
Concatenation: Hello World
Substring (first 5 chars): Hello
Uppercase: HELLO WORLD
Lowercase: hello world
Replaced: Hello C#
Splitted Words:
Apple
Banana
Cherry
```

1.2 String Comparison

```
using System;
class Program
{
    static void Main()
    {
        string str1 = "Hello";
        string str2 = "hello";

        // Case-sensitive comparison
```

```
        bool isEqual = str1 == str2;
        Console.WriteLine("Case-sensitive comparison: " + isEqual);

        // Case-insensitive comparison
        bool isEqualIgnoreCase = string.Equals(str1, str2, StringComparison.OrdinalIgnoreCase);
        Console.WriteLine("Case-insensitive comparison: " + isEqualIgnoreCase);
    }
}
```
Output:
Case-sensitive comparison: False
Case-insensitive comparison: True

1.3 String Formatting

String formatting allows embedding values in a string efficiently.

Example: Using **string.Format()**

```
using System;

class Program
{
    static void Main()
    {
        string name = "Rajesh";
        int age = 25;

        string formatted = string.Format("My name is {0} and I am {1} years old.", name, age);
        Console.WriteLine(formatted);

        // Using string interpolation (C# 6+)
        Console.WriteLine($"My name is {name} and I am {age} years old.");
    }
}
```
Output:
My name is Rajesh and I am 25 years old.
My name is Rajesh and I am 25 years old.

2. StringBuilder in C#

The **StringBuilder** class is found in the System.Text namespace and is used when multiple string modifications are required, improving performance over immutable string.

2.1 Creating and Using StringBuilder

```
using System;
using System.Text; // Required for StringBuilder

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

```
        // Creating a StringBuilder instance
        StringBuilder sb = new StringBuilder("Hello");

        // Append text
        sb.Append(" World");
        Console.WriteLine("After Append: " + sb);

        // Insert text at a specific position
        sb.Insert(6, "Beautiful ");
        Console.WriteLine("After Insert: " + sb);

        // Replace text
        sb.Replace("World", "C#");
        Console.WriteLine("After Replace: " + sb);

        // Remove a part of the string
        sb.Remove(6, 9);
        Console.WriteLine("After Remove: " + sb);
    }
}
```

Output:

```
After Append: Hello World
After Insert: Hello Beautiful World
After Replace: Hello Beautiful C#
After Remove: Hello C#
```

3.3.Event Handling

Event handling is a mechanism where a class (publisher) triggers an event, and other classes (subscribers) respond to it. It is used to handle user actions like button clicks, key presses, etc.

Event handling in C# allows a program to respond to actions like button clicks, key presses, or other user interactions. It follows a structured approach using **delegates**, **events**, and **event handlers**.

An event is a notification sent by an object to signal the occurrence of an action. Events in .NET follow the observer design pattern.

The class who raises events is called Publisher, and the class who receives the notification is called Subscriber. There can be multiple subscribers of a single event. Typically, a publisher raises an event when some action occurred. The subscribers, who are interested in getting a notification when an action occurred, should register with an event and handle it.

In C#, an event is an encapsulated delegate. It is dependent on the delegate. The delegate defines the signature for the event handler method of the subscriber class.

Declare an Event

An event can be declared in two steps:

1. Define a delegate.
2. Declare a variable of the delegate with event keyword.

Example of declaring Event

```
// Step 1: Define a delegate
   public delegate void NotifyHandler(string message);
```

```
  // Step 2: Declare an event
  public static event NotifyHandler NotifyEvent;
```

Steps in Event Handling:

1. **Define a Delegate** → Acts as a placeholder for the method signature.
2. **Declare an Event** → Based on the delegate.
3. **Create an Event Handler Method** → Defines what happens when the event occurs.
4. **Subscribe to the Event** → Attach the event handler to the event.
5. **Trigger (Raise) the Event** → Execute the event.
6. **Unsubscribe from the Event** (Optional) → Remove the event handler when it's no longer needed.

1.Program Example of Event Handling in C#

```
using System;
class EventExample
{
  // Step 1: Define a delegate (Event Handler)
  public delegate void MyEventHandler(string message);

  // Step 2: Declare an event
  public static event MyEventHandler MyEvent;

  // Step 3: Create an event handler method
  public static void EventHandlerMethod(string message)
  {
    Console.WriteLine("Event received: " + message);
  }

  static void Main()
  {
    // Step 4: Subscribe to the event
    MyEvent += EventHandlerMethod;

    // Step 5: Trigger the event
    if (MyEvent != null)
    {
      MyEvent("Hello, this is an event example!");
    }
    // Step 6: Unsubscribe from the event (Optional)
    MyEvent -= EventHandlerMethod;
  }
}
```

OUTPUT: Event received: Hello, this is an event example!

Handle Events Using Invoke() in C#

- Create an event and trigger it using `Invoke()`.
- Event handlers execute when the event is raised.
- **`Invoke()`**: Used to call an event safely.
- **`?.Invoke()`**: Ensures the event is not `null` before calling it, preventing runtime errors.

Notification Mechanism: Events notify subscribers when a specific action occurs.

2.Program example using Invoke()

```
using System;
class EventExample
{
   // Step 1: Define a delegate
   public delegate void MyEventHandler(string message);

   // Step 2: Declare an event
   public event MyEventHandler MyEvent;

   // Step 3: Method to trigger the event
   public void TriggerEvent()
   {
      MyEvent?.Invoke("Hello! Event has been triggered.");
   }
}
class Program
{
   static void Main()
   {
      // Step 4: Create an object
      EventExample obj = new EventExample();

      // Step 5: Subscribe to the event
      obj.MyEvent += MessageHandler;

      // Step 6: Invoke the event
      obj.TriggerEvent();
   }
   // Step 7: Event handler method
   static void MessageHandler(string message)
   {
      Console.WriteLine(message);
   }
}
```

OUTPUT:

```
Hello! Event has been triggered.
```

3.Program example of event handling in one class

```
using System;
class Program
{
   // Step 1: Define a delegate
   public delegate void NotifyHandler(string message);

   // Step 2: Declare an event
   public static event NotifyHandler NotifyEvent;

   // Step 3: Notify method to invoke the event
   public static void Notify(string message)
   {
      NotifyEvent?.Invoke(message);  // Safe event call
   }

   // Step 4: Event handler method
```

```
    public static void EventListener(string message)
    {
        Console.WriteLine("Event Received: " + message);
    }

    static void Main()
    {
        NotifyEvent += EventListener;  // Step 5: Subscribe to event
        Notify("Hello, Event Triggered!");  // Step 6: Invoke event
        NotifyEvent -= EventListener;  // Step 7: Unsubscribe (Optional)
    }
}
```

OUTPUT:

Event Received:Hello, Event Triggered!

3.4.3 Important Methods of StringBuilder Class:

The **`StringBuilder`** class in **C#** provides various methods to efficiently manipulate strings.

1**. Append(string value):** Adds a string at the end of the current StringBuilder content.

```
StringBuilder sb = new StringBuilder("Hello");
sb.Append(" World");
Console.WriteLine(sb); // Output: Hello World
```

2**. AppendLine():**Appends a new line (\n) after the appended text
.

```
sb.AppendLine("Welcome to C#.");
Console.WriteLine(sb);
// Output:
// Hello World
// Welcome to C#.
```

3. **AppendFormat():** Appends a formatted string, similar to string.Format().

```
sb.AppendFormat(" My age is {0}.", 25);
Console.WriteLine(sb); // Output: Hello World Welcome to C#. My age is 25.
```

4**. Insert(int index, string value) :** Inserts a string at a specified position (index).

```
sb.Insert(6, "Beautiful ");
Console.WriteLine(sb); // Output: Hello Beautiful World Welcome to C#. My age is 25.
```

5. **Replace(string oldValue, string newValue):** Replaces all occurrences of a substring with another string.

```
sb.Replace("World", "C#");
Console.WriteLine(sb); // Output: Hello Beautiful C# Welcome to C#. My age is 25.
```

Program Example:

```
using System;
using System.Text;  // Required for StringBuilder

class Program
```

```
{
   static void Main()
   {
      // Step 1: Create a StringBuilder instance
      StringBuilder sb = new StringBuilder("Hello");

      // 1Append - Adding text at the end
      sb.Append(" World");
      Console.WriteLine("After Append: " + sb);

      // 2AppendLine - Adding text with a new line
      sb.AppendLine("! Welcome to C#.");
      Console.WriteLine("After AppendLine: " + sb);

      // 3AppendFormat - Formatting and appending
      sb.AppendFormat(" The year is {0}.", 2025);
      Console.WriteLine("After AppendFormat: " + sb);

      // 4Insert - Inserting text at a specific index
      sb.Insert(6, "Beautiful ");
      Console.WriteLine("After Insert: " + sb);

      // 5Replace - Replacing text
      sb.Replace("World", "C#");
      Console.WriteLine("After Replace: " + sb);
   }
}
```

Output:

```
After Append: Hello World
After AppendLine: Hello World
! Welcome to C#.

After AppendFormat: Hello World
! Welcome to C#.
 The year is 2025.

After Insert: Hello Beautiful World
! Welcome to C#.
 The year is 2025.

After Replace: Hello Beautiful C#
! Welcome to C#.
 The year is 2025.
```

Collections in C#: Generic and Non-Generic

In C#, collections are used to store and manage groups of objects. They provide a way to work with data structures like lists, queues, stacks, and dictionaries. Collections in C# can be broadly categorized into two types:

1. Non-Generic Collections
2. Generic Collections

1. Non-Generic Collections

Non-generic collections are part of the System.Collections namespace. They can store any type of object because they work with System.Object. However, they are not type-safe, which can lead to runtime errors if the wrong type is used.

Common Non-Generic Collections:

- **ArrayList**: A dynamically sized array that can store any type of object.
- **Hashtable**: A collection of key-value pairs, where keys are hashed for quick lookup.
- **Queue:** A first-in, first-out (FIFO) collection.
- **Stack:** A last-in, first-out (LIFO) collection.

Example of Non-Generic Collections:

```
using System;
using System.Collections;
class Program
{
   static void Main()
   {
      // ArrayList example
      ArrayList arrayList = new ArrayList();
      arrayList.Add(10);      // Add an integer
      arrayList.Add("Hello"); // Add a string
      arrayList.Add(3.14);    // Add a double
      Console.WriteLine("ArrayList:");
      foreach (var item in arrayList)
      {
         Console.WriteLine(item);
      }
      // Hashtable example
      Hashtable hashtable = new Hashtable();
      hashtable.Add("ID", 101);
      hashtable.Add("Name", "John Doe");
      hashtable.Add("Salary", 50000.50);
      Console.WriteLine("\nHashtable:");
      foreach (DictionaryEntry entry in hashtable)
      {
         Console.WriteLine($"{entry.Key}: {entry.Value}");
      }
   }
}
```

Output:

```
ArrayList:
10
Hello
3.14

Hashtable:
ID: 101
Name: John Doe
Salary: 50000.5
```

2. Generic Collections

Generic collections are part of the System.Collections.Generic namespace. They are type-safe, meaning you specify the type of objects they can store at compile time. This eliminates the risk of runtime type errors and improves performance by avoiding boxing and unboxing.

Common Generic Collections:

- **List<T>:** A dynamically sized list of a specific type.
- **Dictionary<TKey, TValue>:** A collection of key-value pairs with specific types.
- **Queue<T>:** A FIFO collection of a specific type.
- **Stack<T>:** A LIFO collection of a specific type.

Example of Generic Collections:

```
using System;
using System.Collections.Generic;
class Program
{
   static void Main()
   {
      // List<T> example
      List<int> intList = new List<int>();
      intList.Add(10);
      intList.Add(20);
      intList.Add(30);
      Console.WriteLine("List<int>:");
      foreach (int item in intList)
      {
         Console.WriteLine(item);
      }
      // Dictionary<TKey, TValue> example
      Dictionary<string, int> dictionary = new Dictionary<string, int>();
      dictionary.Add("Apple", 1);
      dictionary.Add("Banana", 2);
      dictionary.Add("Cherry", 3);
      Console.WriteLine("\nDictionary<string, int>:");
      foreach (KeyValuePair<string, int> entry in dictionary)
      {
         Console.WriteLine($"{entry.Key}: {entry.Value}");
      }
   }
```

Output:

```
List<int>:
10
20
30

Dictionary<string, int>:
Apple: 1
Banana: 2
Cherry: 3
```

Key Differences Between Generic and Non-Generic Collections:

Feature	Non-Generic Collections	Generic Collections
Type Safety	Not type-safe (stores object)	Type-safe (specifies type T)
Performance	Slower (boxing/unboxing)	Faster (no boxing/unboxing)
Namespace	System.Collections	System.Collections.Generic
Example Classes	ArrayList, Hashtable	List<T>, Dictionary<TKey, TValue>

- **Non-generic collections are flexible but not type-safe and can lead to runtime errors.**
- **Generic collections are type-safe, performant, and preferred in modern C# programming**.

1. Boxing
Boxing is the process of converting a value type (e.g., int, float, struct) to a reference type (e.g., object). When boxing occurs, the value type is wrapped inside an object and stored on the heap (memory used for dynamic allocation).

Example of Boxing:
1. Boxing
Boxing is the process of converting a value type (e.g., int, float, struct) to a reference type (e.g., object). When boxing occurs, the value type is wrapped inside an object and stored on the heap (memory used for dynamic allocation).

Example of Boxing:
```
int number = 42;          // Value type (stored on the stack)
object boxed = number;    // Boxing: value type is converted to reference type
```

What Happens During Boxing?
Memory is allocated on the heap to store the value type.
The value type is copied from the stack to the heap.
A reference to the heap location is stored in the object variable.

2. Unboxing
Unboxing is the reverse process of boxing. It converts a reference type (e.g., object) back to a value type. During unboxing, the value is copied from the heap to the stack.

Example of Unboxing:
```
object boxed = 42;        // Reference type (boxed value on the heap)
int unboxed = (int)boxed; // Unboxing: reference type is converted back to value type
```

What Happens During Unboxing?
The runtime checks if the object contains a value of the correct type.
If the type is correct, the value is copied from the heap to the stack.
If the type is incorrect, an InvalidCastException is thrown.

Boxing and unboxing are commonly used in non-generic collections (e.g., ArrayList, Hashtable) because these collections store elements as object.

Example Program of Boxing and Unboxing:
```
using System;

class Program
{
```

```
    static void Main()
    {
        // Boxing
        int value = 123;              // Value type (int)
        object boxed = value;         // Boxing: int to object
        Console.WriteLine($"Boxed value: {boxed}");

        // Unboxing
        int unboxed = (int)boxed;     // Unboxing: object to int
        Console.WriteLine($"Unboxed value: {unboxed}");

        // Invalid unboxing (throws InvalidCastException)
        try
        {
            double invalidUnbox = (double)boxed; // This will throw an exception
        }
        catch (InvalidCastException ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
}
```

Output:
Boxed value: 123
Unboxed value: 123
Error: Specified cast is not valid.

Why Avoid Boxing and Unboxing?

- **Performance:** Boxing and unboxing involve memory allocation and copying, which can slow down your program.
- **Type Safety:** Unboxing requires explicit casting, which can lead to runtime errors if the types don't match.
- **Alternatives:** Use generic collections (e.g., List<T>, Dictionary<TKey, TValue>) to avoid boxing and unboxing altogether.

Example Without Boxing and Unboxing (Using Generics):

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

class Program
{
    static void Main()
    {
        // Using a generic list to avoid boxing/unboxing
        List<int> numbers = new List<int>();
        numbers.Add(123); // No boxing occurs

        int value = numbers[0]; // No unboxing occurs
        Console.WriteLine($"Value: {value}");
    }
}
```

Output:
Value: 123