Lesson 2

C# Language Essentials

Basic Syntax and Control Structures

C# is structured similarly to other C-like languages but offers enhancements that improve readability and maintainability.

Control Statements: C# provides the standard if, else, for, while, do-while, foreach, and switch constructs. foreach, for example, is designed for iterating over collections or any type that implements IEnumerable, simplifying code that deals with arrays or collections.

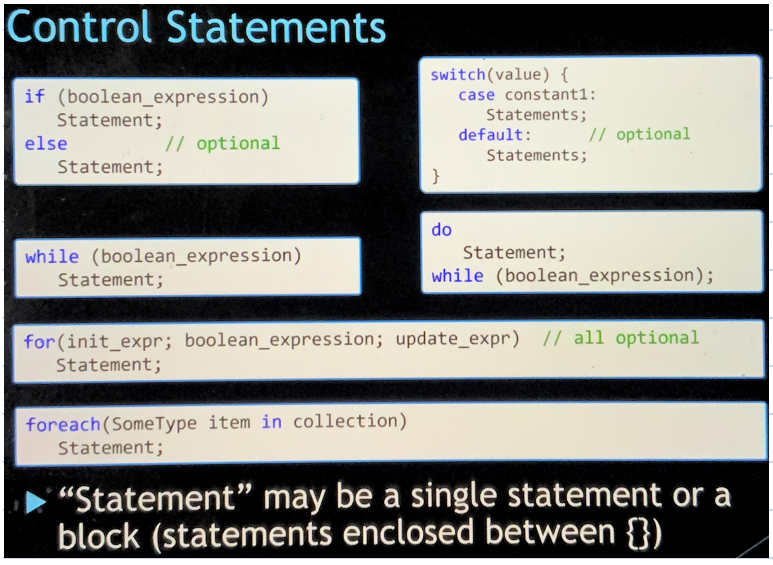
Switch Statements: C# requires a break statement between cases to prevent fall-through behavior (unlike C/C++). If fall-through behavior is needed, you can use goto, but this is generally discouraged to maintain readability.

Advanced Syntax and Pattern Matching

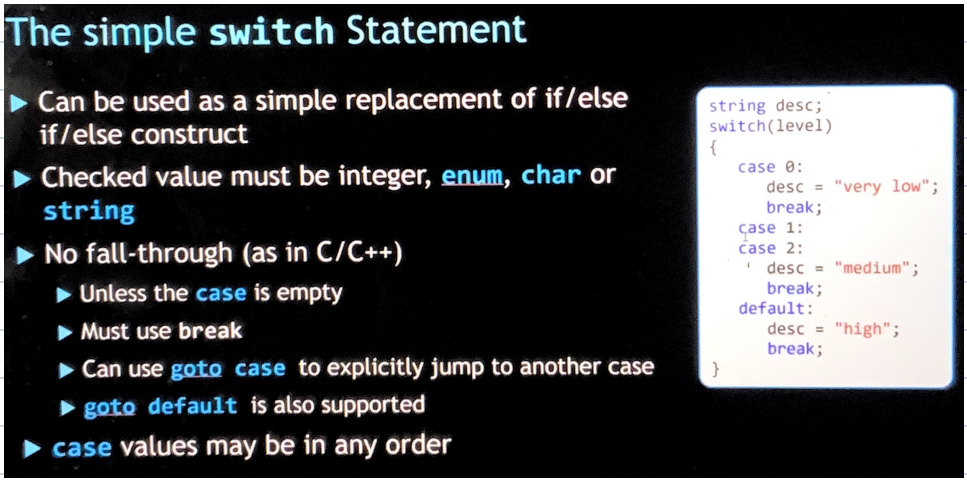
Recent C# versions introduced advanced pattern matching and a new switch syntax. This allows more expressive and concise code:

Pattern Matching: Enables more powerful conditions in switch and if statements by matching types and values directly. For instance, you can match based on object properties, conditions (like ranges), and types, allowing highly readable and expressive conditional checks.

Expression-based switch: Allows switch to return values directly, making it ideal for cases where you want a result based on a condition.




```
namespace FinalizeDemo;

class Program
{
    static void CreateAndCollect()
    {
        /*using*/ DemoClass demo = new();

        string? s = Console.ReadLine();

        switch (s)
        {
            case "a":
                Console.WriteLine("You entered 'a'");
                goto case "b";
            case "b":
                Console.WriteLine("You entered 'b'");
                break;
        }
    }
```



Loop Control

- The **break** keyword
  - Causes exit from the innermost loop
- The **continue** keyword
  - Abandons the current iteration and jumps back to the start of the update clause or the body
- The **goto** keyword
  - Allows unconditional jump to a label
  - Almost never used

The Ternary Operator

- Contains 3 parts
- Syntax

```
A ? B : C
```

- Equivalent to

```
if(A) B else C
```

- But it's an expression, not a statement

Pattern Matching Operators

- C# 7.0 introduced pattern matching operators that enhanced the **if** and **switch** statements
- There are many types of constructs, here are some examples:

```
object greeting = "Hello, World!";
if (greeting is string message)
{
    Console.WriteLine(message.ToLower());  // output: hello, world!
}

if (input is not null)
{
    // ...
}
```

```
var numbers = new int[] { 10, 20, 30 };
Console.WriteLine(GetSourceLabel(numbers));  // output: 1

var letters = new List<char> { 'a', 'b', 'c', 'd' };
Console.WriteLine(GetSourceLabel(letters));  // output: 2

static int GetSourceLabel<T>(IEnumerable<T> source) => source switch
{
    Array array => 1,
    ICollection<T> collection => 2,
    _ => 3,
};
```

Relational patterns

You use a *relational pattern* to compare an expression result with a constant, as the following example shows:

C# Copy

```
Console.WriteLine(Classify(13));  // output: Too high
Console.WriteLine(Classify(double.NaN));  // output: Unknown
Console.WriteLine(Classify(2.4));  // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -4.0 => "Too low",
    > 10.0 => "Too high",
    double.NaN => "Unknown",
    _ => "Acceptable",
};
```

Pattern Matching Switch Expression

```
Console.WriteLine(GetDiscountInPercent(DayOfWeek.Friday));  // output: 5.0
Console.WriteLine(GetDiscountInPercent(DayOfWeek.Monday));  // output: 0.5
Console.WriteLine(GetDiscountInPercent((DayOfWeek)10));  // output: 0.0

static decimal GetDiscountInPercent(DayOfWeek dayOfWeek)
{
    return dayOfWeek switch
    {
        DayOfWeek.Monday => 0.5m,
        DayOfWeek.Tuesday => 12.5m,
        DayOfWeek.Wednesday => 7.5m,
        DayOfWeek.Thursday => 12.5m,
        DayOfWeek.Friday => 5.0m,
        DayOfWeek.Saturday => 2.5m,
        DayOfWeek.Sunday => 2.0m,
        _ => 0.0m
    };
};
```

Understanding IEnumerator and IEnumerable

In .NET, IEnumerator and IEnumerable are interfaces that support iteration over a collection of objects. These interfaces play a key role in making collections (like arrays, lists, and custom data structures) iterable with the foreach loop and LINQ.

IEnumerable Interface

IEnumerable is the primary interface for collections in .NET. It only has one method, GetEnumerator(), which returns an IEnumerator. Implementing IEnumerable on a class allows it to be used with foreach, LINQ, and other collection-based operations.

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Purpose: Enables a collection to be iterable.
Usage: Classes that implement IEnumerable can be iterated over using foreach.

IEnumerator Interface

IEnumerator provides the mechanisms to traverse a collection one element at a time. When you use a foreach loop in C#, an IEnumerator is used behind the scenes to access each item.

```
public interface IEnumerator
{
    bool MoveNext();     // Moves to the next element in the collection.
    object Current { get; } // Gets the current element in the collection.
    void Reset();        // Resets the enumerator to the beginning.
}
```

Key Methods and Properties in IEnumerator

MoveNext(): Advances the enumerator to the next element in the collection. This method returns true if it successfully moves to the next item, and false if it has passed the end of the collection.
Current: Gets the item at the current position of the enumerator. You can think of Current as pointing to the current element in the collection as the enumerator moves through it.
Reset(): Resets the enumerator back to its initial position (before the first element in the collection). Reset() isn't frequently used in foreach loops but may be useful if you need to manually restart an iteration.

Example: Implementing IEnumerable and IEnumerator in a Custom Collection

Here's an example that demonstrates a custom collection class (NumberCollection) implementing IEnumerable and an internal enumerator class (NumberEnumerator) implementing IEnumerator.

```
using System;
using System.Collections;

public class NumberCollection : IEnumerable
{
    private int[] numbers = { 1, 2, 3, 4, 5 };

    public IEnumerator GetEnumerator()
    {
        return new NumberEnumerator(numbers);
    }

    // Inner class that implements IEnumerator
    private class NumberEnumerator : IEnumerator
    {
        private int[] _numbers;
        private int _position = -1;  // Start position before the first element

        public NumberEnumerator(int[] numbers)
        {
            _numbers = numbers;
        }

        // Move to the next element in the collection
        public bool MoveNext()
        {
            _position++;
            return _position < _numbers.Length;
        }

        // Return the current element in the collection
        public object Current
        {
            get
            {
                if (_position < 0 || _position >= _numbers.Length)
                    throw new InvalidOperationException();
                return _numbers[_position];
            }
        }

        // Reset the enumerator position to before the first element
        public void Reset()
        {
            _position = -1;
        }
    }
}
```

Explanation of the Example

NumberCollection Class: Implements IEnumerable, allowing it to return an enumerator (NumberEnumerator) that iterates over its elements.

NumberEnumerator Class: This nested class implements IEnumerator to handle the actual iteration logic.

MoveNext(): Advances the position by one, allowing the foreach loop to continue to the next element.

Current: Returns the current element. It throws an exception if accessed out of bounds, ensuring the collection is accessed only when valid.

Reset(): Resets the position to -1, so iteration can start from the beginning again.

Control Flow and Iteration

Loop Structures

C# offers a variety of loops:

for and while loops: Basic loops, similar to those in other languages.

foreach loop: Extends functionality, enabling iteration over any object that implements IEnumerable or IEnumerable<T>, including collections and arrays. It's often safer and more readable than indexed loops for collection traversal.

Yielding Control with yield

The yield keyword simplifies implementing iterators in C#:

Generators: By using yield return, the method returns each value in sequence, pausing and resuming execution each time. This behavior is particularly useful for iterating over large datasets without consuming extra memory.

yield break: Stops the iteration early without completing the entire collection.

goto Statements

While rarely used, goto allows jumping between code labels. It's generally discouraged because it breaks structured programming principles, but it has niche applications in code generation and exceptional cases where optimized jumps are needed.

```
class MyCollection
{
    1 reference
    public IEnumerable<int> Range(int max)
    {
        for (int i = 0; i < max; i++)
        {
            yield return i;

        }


        for (int i = max - 1; i >= 0; i--)
        {
            yield return i;
        }
    }

}
```

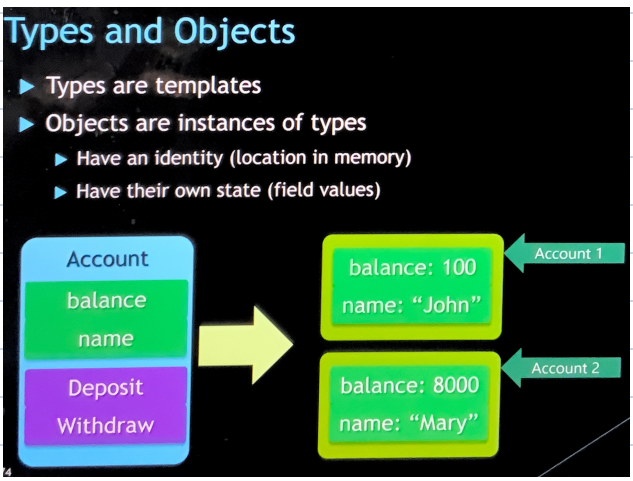
```
public IEnumerable<int> Range(int max)
{
    for (int i = 0; i < max; i++)
    {
        if (i == 5)
        {
            yield break;
        }

        yield return i;


    }


    for (int i = max - 1; i >= 0; i--)
    {
        yield return i;
    }
```

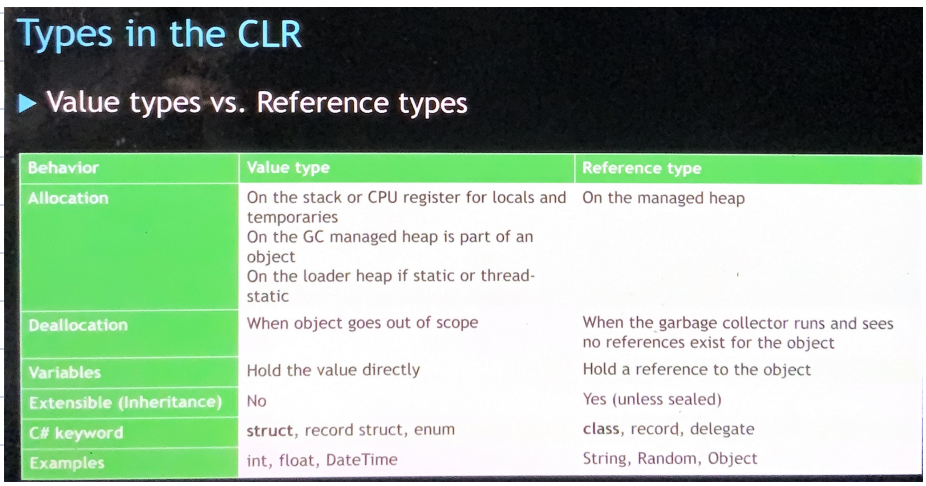
Memory Management: Value Types vs. Reference Types

C# handles memory allocation differently for two primary data types:

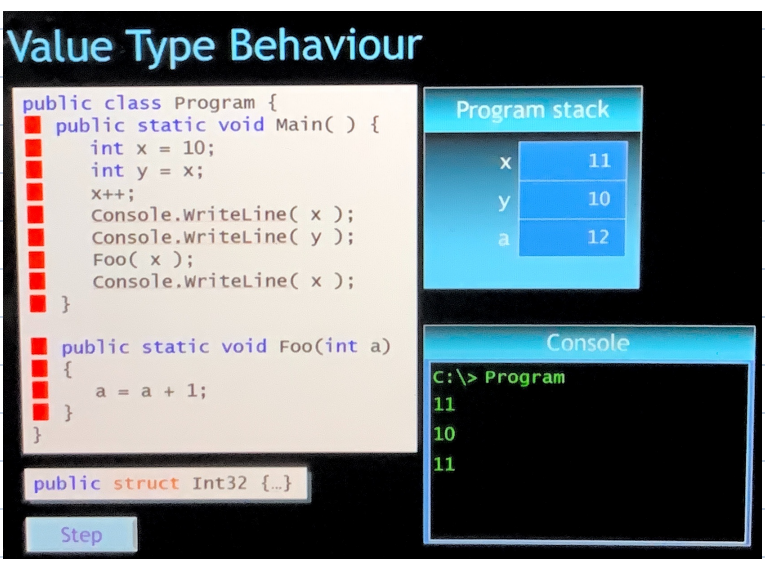
Value Types: These types, like int and struct, are stored on the stack or in CPU registers. They are copied when passed to functions, making them fast for small, frequently-used data (e.g., coordinates). Value types cannot contain null, but Nullable<T> or shorthand T? allows them to represent null values.

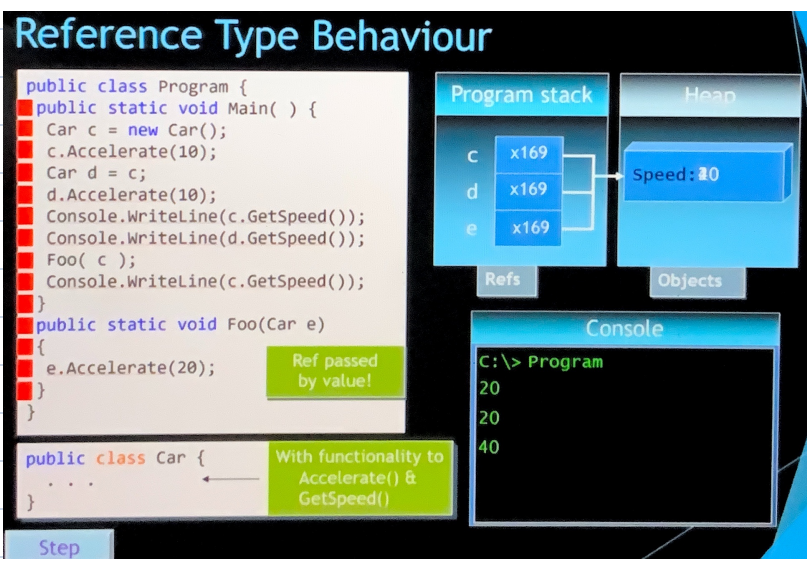
Reference Types: Objects like classes and arrays are stored on the heap and accessed via pointers. When passed to methods, only the reference (pointer) is copied, not the data itself, making them efficient for large or complex data. Reference types can hold null, indicating the absence of data.

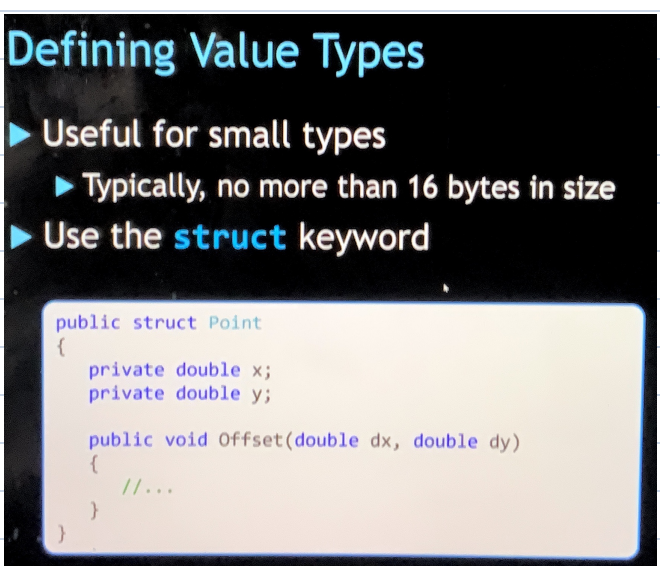


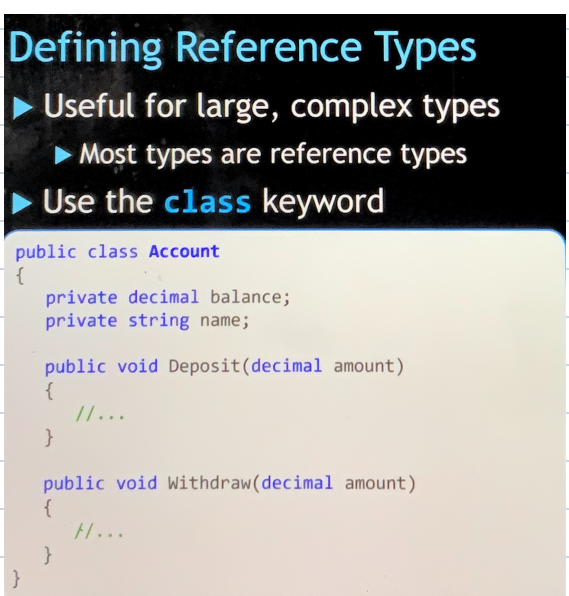


Behavior	Value type	Reference type
Allocation	On the stack or CPU register for locals and temporaries On the GC managed heap is part of an object On the loader heap if static or thread-static	On the managed heap
Deallocation	When object goes out of scope	When the garbage collector runs and sees no references exist for the object
Variables	Hold the value directly	Hold a reference to the object
Extensible (Inheritance)	No	Yes (unless sealed)
C# keyword	struct, record struct, enum	class, record, delegate
Examples	int, float, DateTime	String, Random, Object









The null Reference

- Reference type variables can hold the special value null
  - Don't reference any object
- Value type variables cannot hold null, unless declared as nullable
  - Always hold some value
- Can check equality / inequality with null

```
string name = Console.ReadLine();
Account? acc = GetAccount(name);
if(acc == null)
   Console.WriteLine("Account does not exist");
else
{
   acc.Deposit(100);
   SaveAccount(acc);
   acc = null;
}
```

The null Coalescing Operator

- A shorthand for the ternary operator that checks for null
- Syntax

  A ?? B
- Means

  A != null ? A : B

Nullable Value Types

- System.Nullable<T>
  - Represents value type of type T or null
- Contains two properties
  - HasValue
    - Boolean signifying whether object actually contains a valid value
  - Value
    - The actual value (of type T)
- C# offers the T? syntax for easier use

```
int? x;     // same as Nullable<int> x;
x = 5;      // ok
x = null;       // also ok
int y = (int)x; // ok as well
```

Null-Coalescing Assignment - ??= (>= C# 8.0)

- Very useful for default values

```
List<int> numbers = null;
int? i = null;
numbers ??= new List<int>();
numbers.Add(i ??= 17);
numbers.Add(i ??= 20);
Console.WriteLine(string.Join(" ", numbers)); // output: 17 17
Console.WriteLine(i); // output: 17
```

Nullable Reference Types (C# 8)

- Null-state analysis tracks the null-state of references
  - This static analysis emits warnings when your code may dereference null
  - Done by both the IDE and the compiler
  - To reduce the possibility to get System.NullReferenceException C# you can enable the Nullable context
    - Within the code use: #nullable enable or #nullable disable
    - In the project file set: <Nullable>enable</Nullable>
- Variable annotations:
  - To allow null assignment: use the ? As in nullable value type
    - string? name;
  - To override nullable analysis, use the !
    - name!.Length

Null-conditional operators

- The null-conditional operator lets you access members and elements only when the receiver is not-null, providing a null result otherwise:
  - int? length = customers?.Length; // null if customers is null
  - Customer first = customers?[0]; // null if customers is null
- The null-conditional operator is conveniently used together with the null coalescing operator ??:
  - int length = customers?.Length ?? 0; // 0 if customers is null
- The null-conditional operator exhibits short-circuiting behavior,
  - Element accesses and invocations will only be executed if the original receiver was not null:
  - int? first = customers?[0].Orders.Count();
  - Instead of: int? first = (customers != null) ? customers[0].Orders.Count() : null;
    - Except that customers is only evaluated once
- null-conditional operators can themselves be chained:
  - int? first = customers?[0].Orders?.Count();

Boxing and Unboxing

Boxing: Converts a value type to a reference type by copying the value type data to the heap. This operation incurs a small performance cost since it requires heap allocation.

Unboxing: Extracts the value type from a boxed object. Unboxing checks the type, and if successful, returns a copy of the data.

Performance Impact: Avoid excessive boxing and unboxing as they can slow down applications, particularly in tight loops. Using generics or method overloads is preferable for avoiding unnecessary boxing.

Example Code for Boxing and Unboxing

```
using System;

class Program
{
    static void Main()
    {
        int value = 42;

        // Boxing: converting the value type 'int' to an object reference type
        object boxedValue = value;
        Console.WriteLine($"Boxed Value: {boxedValue} (Type: {boxedValue.GetType()})");

        // Unboxing: converting the object reference back to an int
        int unboxedValue = (int)boxedValue;
        Console.WriteLine($"Unboxed Value: {unboxedValue}");

        // Performance Impact Demonstration
        int sum = 0;

        // Boxing and unboxing in a tight loop - this can slow down performance
        for (int i = 0; i < 1000000; i++)
        {
            object box = i; // Boxing
            sum += (int)box; // Unboxing
        }

        Console.WriteLine($"Sum after boxing and unboxing in loop: {sum}");

        // Using generics to avoid boxing and unboxing
        sum = 0;
        for (int i = 0; i < 1000000; i++)
        {
            sum += AddGeneric(i); // No boxing or unboxing with generic method
        }

        Console.WriteLine($"Sum after using generics to avoid boxing: {sum}");
    }

    // Generic method avoids boxing and unboxing
    static T AddGeneric<T>(T value) where T : struct
    {
        return value;
    }
```

Explanation of Code

Boxing: When value (an int) is assigned to boxedValue (an object), it is boxed. This means the integer value is wrapped in an object and moved to the heap.

Unboxing: When boxedValue is cast back to an int, it is unboxed. The runtime checks that boxedValue contains an integer, and if so, it copies the value back to the stack.

Performance Impact: The loop that sums values using boxing and unboxing repeatedly allocates on the heap for each boxed int, adding overhead and slowing down performance.

Avoiding Boxing with Generics: The generic method AddGeneric<T>() avoids boxing by keeping the value as a struct without converting it to an object. This prevents repeated heap allocations and provides better performance.

Output Example

Boxed Value: 42 (Type: System.Int32)
Unboxed Value: 42
Sum after boxing and unboxing in loop: 1783293664
Sum after using generics to avoid boxing: 1783293664

The Importance of Avoiding Boxing with Generics and Overloading

Generics and method overloading prevent boxing by ensuring value types don't need to be cast to **object**. .NET's core libraries, like **Console.WriteLine**, heavily use method overloading for this reason, as it provides better performance by avoiding boxing/unboxing for primitive types.

Generics in C#

Generics provide a way to define type-safe, reusable components:

Type Safety without Boxing: Generics allow you to create collections and methods that work with any type, eliminating the need for boxing/unboxing and enabling compile-time type checks.
Enhanced Performance: Since generics operate directly on the specified type, they avoid the overhead associated with boxed objects, resulting in faster, more memory-efficient code.

Mastering Generics: Avoid boxing by using generics wherever possible, especially in collections or methods working with different types.
Memory Management: Use struct for small, frequently used objects. Rely on Dispose to free up unmanaged resources instead of relying on finalizers.
GC Optimization: Keep GC in mind with short- and long-lived objects to minimize collection cycles. Use workstation or server GC modes depending on your application's requirements.

Memory Allocation Techniques

.NET supports both stack and heap allocation:

Stack Allocation: Temporary variables (e.g., primitive types) are stored on the stack, which has a limited but highly efficient allocation method.
Heap Allocation: For objects that require memory beyond the scope of a single function, heap allocation is used. Although managed by the GC, frequent heap allocation can lead to fragmentation, increasing GC workload.
stackalloc: Allows stack-based allocation for arrays. Although faster than heap allocation, stack memory is limited and has a short lifetime tied to the function scope.

Performance Tuning for Memory-Intensive Applications

When working on memory-intensive .NET applications, you may consider adjusting the GC mode to suit your application:

Server GC: Optimized for applications with high throughput needs, like web servers, with concurrent GC threads across multiple cores.
Workstation GC: Better for desktop applications with low-latency needs, running GC on a single thread to minimize interruption.
.NET's GC.Collect() can force a collection, but it's generally better to let .NET handle memory as calling GC.Collect() can lead to inefficient memory usage and performance issues.

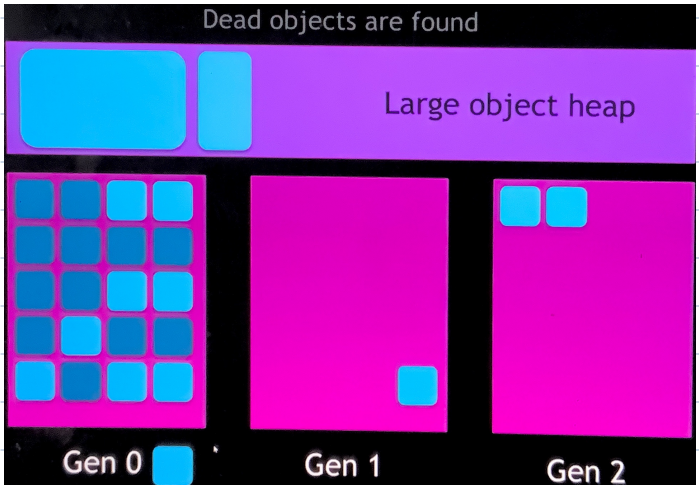
Garbage Collection (GC)

Garbage Collection in .NET manages memory by automatically reclaiming unused objects, with three primary generations:

Gen 0: Short-lived objects (e.g., temporary data structures). GC frequently collects these, often without causing noticeable performance delays.
Gen 1: Objects that survive a Gen 0 collection are promoted to Gen 1. These are medium-lived objects.
Gen 2: Long-lived objects (e.g., data in collections that remain active for the application's lifetime). Collecting Gen 2 objects is more resource-intensive.
.NET also includes the Large Object Heap (LOH) for objects larger than 85 KB (e.g., large arrays). These objects aren't moved in memory during collection but may be compacted if fragmentation becomes an issue.

Generational Collection Process

Graph Building: The GC starts by identifying root objects (those referenced by the stack, CPU registers, static variables, or GC handles) and builds a graph of all reachable objects.
Mark and Sweep: Unreachable objects (not referenced in the graph) are marked as garbage. The GC then compacts memory, eliminating gaps and freeing space for new allocations, improving cache efficiency.



Generations Usage

- When an object is created, it's part of generation 0
- Generation 0 has a limit (typically 256KB)
- When this limit is reached a garbage collection is attempted on generation 0 objects only
  - Surviving objects are incremented to generation 1
  - If enough memory is freed, this concludes this collection
  - If generation 1 is full (typically 2MB), another collection is attempted
  - On generations 1 and 0
- Same reasoning for generation 2
  - Its target size is around 10MB

Large Object Heap

- Any object that is larger than 85000 bytes is considered a large object
  - Do not assume this number
- Allocated from a special heap that is not compacted
  - Managed similarly to a native heap
  - Can be compacted by setting a mode, see below
- Always part of generation 2
- You can request compaction in next collection

```
GCSettings.LargeObjectHeapCompactionMode =
    GCLargeObjectHeapCompactionMode.CompactOnce;
```

Finalization and the Dispose Pattern

In .NET, some objects require cleanup beyond memory management, such as database connections or file handles.

Finalizers: The GC will automatically call finalizers (~ClassName()) before collecting objects to clean up unmanaged resources. Finalizers introduce overhead, as they delay the object's collection until a subsequent GC cycle.
IDisposable Interface and Dispose Method: For predictable resource management, objects should implement IDisposable, allowing you to manually release resources by calling Dispose. Using the using statement ensures Dispose is called, regardless of exceptions.
Suppress Finalization: Calling GC.SuppressFinalize() inside Dispose prevents the GC from running the finalizer, which optimizes performance by reducing unnecessary overhead.

Unsafe Code and Pointers

In special cases, .NET allows **unsafe code** where direct memory manipulation with pointers is possible:

Interop and Performance: Unsafe code is useful for interop scenarios or where maximum performance is essential. However, it's prone to errors, bypasses memory safety, and can lead to security issues.
stackalloc **and** Span<T>: Unsafe code allows direct stack allocation and manipulation using pointers or Span<T>, reducing allocations for temporary data. To use unsafe code, you must compile with allow unsafe enabled.