Module 2

Boxing & Unboxing

- **Boxing:** Converts a value type (like int, double, char) into an object type (object or System.Object).
- **Unboxing:** Extracts the value type from an object. This requires explicit type casting.

Here is a code example demonstrating boxing and unboxing:

```
// Boxing example
int a = 123;
object o = a; // Boxing the int 'a' into an object 'o'

Console.WriteLine(a); // Output: 123
Console.WriteLine(o); // Output: 123

// Unboxing example
int b = (int)o; // Unboxing the object 'o' back into an int 'b'

Console.WriteLine(b); // Output: 123

// Example showing type information
string s = "something";
Console.WriteLine(s.GetType()); // Output: System.String
Console.WriteLine(s.GetType().IsValueType); // Output: False (string is a reference type)
```

Access Modifiers

- **Note:** Access modifiers help encapsulate and protect data while enabling interaction where necessary.

Here is a table summarizing the C# access modifiers:

Modifier	**Accessibility**	**Description**
public	Everywhere	Accessible by any other code in the same assembly or another assembly.
private	Inside the same class only	Accessible only by code in the same class.
protected	Inside the same class and derived classes	Accessible by code in the same class or in any class that derives from that class.
internal	Inside the same assembly	Accessible by any code in the same assembly, but not from another assembly.
protected internal	Inside the same assembly and derived classes	Accessible by any code in the same assembly OR by code in a derived class in another assembly.
private protected	Inside the same class and derived classes (same assembly)	Accessible by code in the same class OR by derived classes *in the same assembly*.

Classes

Static Class

- **Definition:** Cannot be instantiated and contains only static members (methods, properties, and fields).
- **Key Points:**
    - Cannot create objects.
    - Cannot inherit or be inherited.
    - All members must be static.
- **Usage:** Used for functionality that doesn't need an object, like utility/helper classes (Math, Console).
- **Instantiated?** Cannot create objects (no new keyword).
- **Memory Trick:** "Static = Stays the Same, No Objects Needed!"

Here is an example of using a static class method:

```
// Assuming a static class MathHelper exists with a static Add method
var sum = MathHelper.Add(10, 30);
Console.WriteLine(sum); // Output will depend on the implementation of MathHelper.
```

Abstract Class

- **Definition:** An abstract class cannot be instantiated and serves as a blueprint for derived classes.
- **Key Points:**
  - Cannot create objects (no new keyword).
  - Can have abstract methods (must be implemented in derived classes).
  - Can have regular methods (can be inherited directly).
- **Memory Trick:** "Abstract = Incomplete, Needs a Derived Class!"

Partial Class

- Allows defining parts of a single class in multiple .cs files within the same namespace or project.
- Useful for large classes or when working with generated code.
- **Memory Trick:** "Partial = Split Apart, Still One Class!"

Generic

- Used when the specific data Type of a property or method will be defined later (e.g., when the class or method is used).
- **Memory Trick:** "Generic = Flexible Type, Defined Later!"

Here is an example of a generic class:

```
public class Box<T>
{
    public T Value { get; set; } // The type of Value is determined when the Box is created
}

// Examples of using the generic Box class
var box1 = new Box<int> { Value = 10 }; // Box holding an integer
var box2 = new Box<string> { Value = "Hello" }; // Box holding a string

Console.WriteLine(box1.Value); // Output: 10
Console.WriteLine(box2.Value); // Output: Hello
```

Sealed Class

- **Definition:** Cannot be inherited by other classes.
- **Usage:** When you want to prevent others from extending your class for security, stability, or design reasons.
- **Memory Trick:** "Sealed = Final Stop, No Inheritance!"

Record Class

- **Definition:** A reference type that provides built-in functionality for encapsulating data.
- **Key Rule:** Immutable by default and provides value-based equality.
- **Usage:** When you need data-focused objects (like DTOs, API models) that care about values, not references.
- **Memory Trick:** "Record = Data Snapshot, Value Matters!"

Nested Class (not taught in class)

- **Definition:** A class defined within another class.
- **Key Rule:** The inner class is accessible via the outer class.
- **Usage:** Helper types that are only meaningful inside their parent class (like builder classes, or private internal helpers).
- **Memory Trick:** "Nested = Class Inside, Belongs to Outer!"

Summary of C# Class Types

Type	Keyword / Feature	Key Rule	Where / Why to Use
Normal Class	class	Nothing special	For general-purpose objects (like models, services, etc.).
Static Class	static class	Only static members, no instances	Utility / helper classes (like MathHelper, StringUtils). Shared functionality without needing an object.
Abstract Class	abstract class	Cannot instantiate directly	Base class when you want partial implementation. Forces derived classes to implement missing pieces. Good for shared behaviors.
Sealed Class	sealed class	Cannot inherit	When you want to prevent others from extending your class for security, stability, or design reasons.
Partial Class	partial class	Defined across files	Large classes split into multiple files for better organization (common in auto-generated code, WinForms, or big models).
Record Class	record	Immutable, value-based equality	When you need data-focused objects (like DTOs, API models) that care about values, not references.
Generic Class	class<T>	Works with any type	Reusable logic for multiple data types

			(e.g., Collections, Services, Repositories).
Nested Class	class inside class	Inner class accessible via outer	Helper types that are only meaningful inside their parent class (like builder classes, or private internal helpers).

Nullable Types (?)

- Allows value types (and reference types implicitly) to be assigned a value of null.
- Requires appending ? to the value type (e.g., int? x = null;).

Default Keyword

- Used to get the default value for a type.

Type	Default Value Example
int	int defaultInt = default(int);
bool	bool defaultBool = default(bool);
string	string defaultString = default;

Enum (Enumeration)

- Defines a set of named constant values.
- Useful when you have a fixed set of options.
- Example:

```
enum DaysOfWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}
```

Var Keyword

- Allows the compiler to infer the type of a local variable implicitly.
- Similar to auto in C++.
- Example:

```
var name = "Alice"; // Compiler infers 'string'
var age = 25;       // Compiler infers 'int'
var numbers = new List<int> { 1, 2, 3 }; // Compiler infers 'List<int>'
```

Dynamic Keyword

- Allows the type of a variable to be determined at runtime.
- Bypasses compile-time type checking.
- Example:

```
dynamic value = "Hello, world!";
value = 42; // No compile-time error, type changes at runtime
```

Collections

Stack

- Represents a last-in, first-out (LIFO) collection of objects.
- Can be Generic (Stack<T>) for type safety or Non-Generic (Stack) (Non-Generic is generally not recommended).

Here are examples of using both generic and non-generic Stacks:

```
// Generic Stack Example (Recommended - provides type safety)
Console.WriteLine("--- Stack<T> (Generic) Example ---");
var genericStack = new Stack<string>();
genericStack.Push("First");
// genericStack.Push(1); // This would cause a compile-time error because the stack is for strings
genericStack.Push("Third");

Console.WriteLine($"Popped: {genericStack.Pop()}"); // Output: Popped: Third

foreach (var item in genericStack)
{
    Console.WriteLine(item); // Output: First
}

Console.WriteLine(); // Add a blank line for separation

// Non-Generic Stack Example (Not Recommended - lacks type safety, requires casting)
Console.WriteLine("--- Stack (Non-Generic) Example ---");
Stack nonGenericStack = new Stack();
nonGenericStack.Push("First");
nonGenericStack.Push(2); // Can push different types
nonGenericStack.Push(3.14);

Console.WriteLine($"Popped: {nonGenericStack.Pop()}"); // Output: Popped: 3.14 (returns object, may need casting)

foreach (var item in nonGenericStack)
```

```
{
    Console.WriteLine(item); // Output: 2, First (order depends on pop)
}
```

HashSet

- Stores a collection of unique elements.
- Optimized for fast lookups, additions, and removals.

Here is a simple example of a HashSet:

```
Console.WriteLine("--- HashSet Example ---");
var uniqueNumbers = new HashSet<int>();

uniqueNumbers.Add(1);
uniqueNumbers.Add(2);
uniqueNumbers.Add(3);
uniqueNumbers.Add(2); // Adding 2 again has no effect

Console.WriteLine($"Count: {uniqueNumbers.Count}"); // Output: Count: 3

Console.WriteLine($"Contains 2: {uniqueNumbers.Contains(2)}"); // Output: Contains 2: True
Console.WriteLine($"Contains 5: {uniqueNumbers.Contains(5)}"); // Output: Contains 5: False

foreach (var number in uniqueNumbers)
{
    Console.WriteLine(number); // Output: 1, 2, 3 (order not guaranteed)
}
```

Dictionary (Dictionary<TKey, TValue>)

- Represents a collection of key-value pairs.
- Keys must be unique.
- Example:

```
Console.WriteLine("--- Dictionary<TKey, TValue> Example ---");
var capitals = new Dictionary<string, string>
{
    {"USA", "Washington, D.C."},
    {"France", "Paris"},
    {"Japan", "Tokyo"}
};
foreach (var kvp in capitals)
{
    Console.WriteLine($"{kvp.Key}: {kvp.Value}");
}
```

ArrayList (Non-Generic)

- Stores a resizable array of objects.
- **Note:** Generally less recommended than generic collections (List<T>) due to lack of type safety and potential for boxing/unboxing overhead.
- Example:

```
Console.WriteLine("--- ArrayList Example ---");
ArrayList arrayList = new ArrayList();
arrayList.Add(1);
arrayList.Add("Hello");
arrayList.Add(3.14);
arrayList.Add(true);
foreach (var item in arrayList)
{
    Console.WriteLine(item);
}
```

Collections Controlling Thread (Thread-Safe)

Concurrent Collections (e.g., ConcurrentDictionary<TKey, TValue>)

- Located in the System.Collections.Concurrent namespace.
- Designed for use in multi-threaded scenarios.
- They provide built-in locking mechanisms to safely access and modify the collection from multiple threads without explicit locking required by the user.

Here is an example of a ConcurrentDictionary:

```
Console.WriteLine("--- ConcurrentDictionary<TKey, TValue> Example ---");
var concurrentDict = new ConcurrentDictionary<int, string>();

// Add key-value pairs
concurrentDict.TryAdd(1, "One");
concurrentDict.TryAdd(2, "Two");

// Update existing or add new
// If key 2 exists, update its value to "UpdatedTwo"
// If key 3 does not exist, add it with value "Three"
concurrentDict.AddOrUpdate(2, "Second", (key, oldValue) => "UpdatedTwo");
concurrentDict.AddOrUpdate(3, "Three", (key, oldValue) => "UpdatedThree");

// Trying to add key 1 again will have no effect as TryAdd only adds if the key doesn't exist
concurrentDict.TryAdd(1, "One Again");

Console.WriteLine($"Count: {concurrentDict.Count}"); // Output: Count: 3

// Accessing values
if (concurrentDict.TryGetValue(2, out string value))
{
    Console.WriteLine($"Value for key 2: {value}"); // Output: Value for key 2: UpdatedTwo
}

// Iterating through the dictionary (order is not guaranteed)
foreach (var kvp in concurrentDict)
{
    Console.WriteLine($"Key: {kvp.Key}, Value: {kvp.Value}");
}
```

Immutable Collections (e.g., ImmutableList<T>)

- Located in the System.Collections.Immutable namespace.
- Once created, their contents cannot be modified.
- Any "modification" operation (like adding an element) actually returns a *new* immutable collection with the change, leaving the original unchanged.
- Useful in multi-threaded scenarios as they are inherently thread-safe once created.

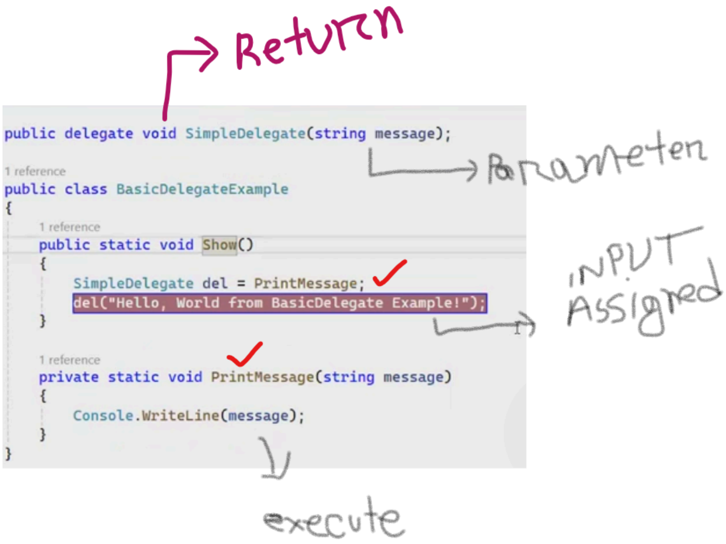
Here is an example of an ImmutableList:

```
// Example of an ImmutableList
Console.WriteLine("--- ImmutableList<T> Example ---");

// Create an initial immutable list
var list = ImmutableList.Create<int>(1, 2, 3);

// Adding an element creates a *new* list; the original remains unchanged
var newList = list.Add(4);

Console.WriteLine("Original List:");
foreach (var number in list)
{
    Console.WriteLine(number); // Output: 1, 2, 3
}

Console.WriteLine("New List:");
foreach (var number in newList)
{
    Console.WriteLine(number); // Output: 1, 2, 3, 4
}

// Removing an element also creates a *new* list
var listAfterRemoval = newList.Remove(2);

Console.WriteLine("List After Removal of 2:");
foreach (var number in listAfterRemoval)
{
    Console.WriteLine(number); // Output: 1, 3, 4
}

// The original 'list' and 'newList' are still unchanged
Console.WriteLine("Original List (still unchanged):");
foreach (var number in list)
{
    Console.WriteLine(number); // Output: 1, 2, 3
}

Console.WriteLine("New List (still unchanged):");
foreach (var number in newList)
{
    Console.WriteLine(number); // Output: 1, 2, 3, 4
}
```

Delegate & Event

Delegate

- A type that holds a reference to a method.
- Allows methods to be passed as arguments (callback mechanisms).



```
public delegate void SimpleDelegate(string message);

1 reference
public class BasicDelegateExample
{
    1 reference
    public static void Show()
    {
        SimpleDelegate del = PrintMessage;
        del("Hello, World from BasicDelegate Example!");
    }

    1 reference
    private static void PrintMessage(string message)
    {
        Console.WriteLine(message);
    }
}
```

Events

- A mechanism used to notify subscribers when something happens.
- Events in C# are based on delegates; they use delegates to define the signature of the event handler methods.
- Think of a button click: the button *publishes* an event, and any code that needs to react to the click *subscribes* to that event using a delegate.

Multicast Delegates

- A delegate instance that can hold references to multiple methods.
- When the delegate is invoked, all the methods it references are executed in the order they were added.
- Example:

```
public delegate void Notify(string message);

// Methods to be called by the delegate
private static void ShowMessage(string msg)
{
    Console.WriteLine($"Message: {msg}");
}

private static void ShowUpperMessage(string msg)
{
    Console.WriteLine($"Uppercase Message: {msg.ToUpper()}");
}

// Creating a multicast delegate
Notify notify = ShowMessage;
notify += ShowUpperMessage; // Add another method

// Invoking the delegate calls both methods
notify.Invoke("Multicast Delegate Example");
```

Publisher & Subscriber Model (Pub-Sub)

- A system design pattern where "publishers" send messages (events) without knowing who will receive them, and "subscribers" receive messages without knowing who sent them.
- Delegates and Events in C# are a common way to implement the Pub-Sub pattern within an application.

Essential Topics You Need to Explore

- Class Code: Event & Delegate Github repo [Must check]
- Pub & Sub Model
- Delegate multiple ways [anonymous function & lambda function]
- Class Code: C# file operations[note kri nai]