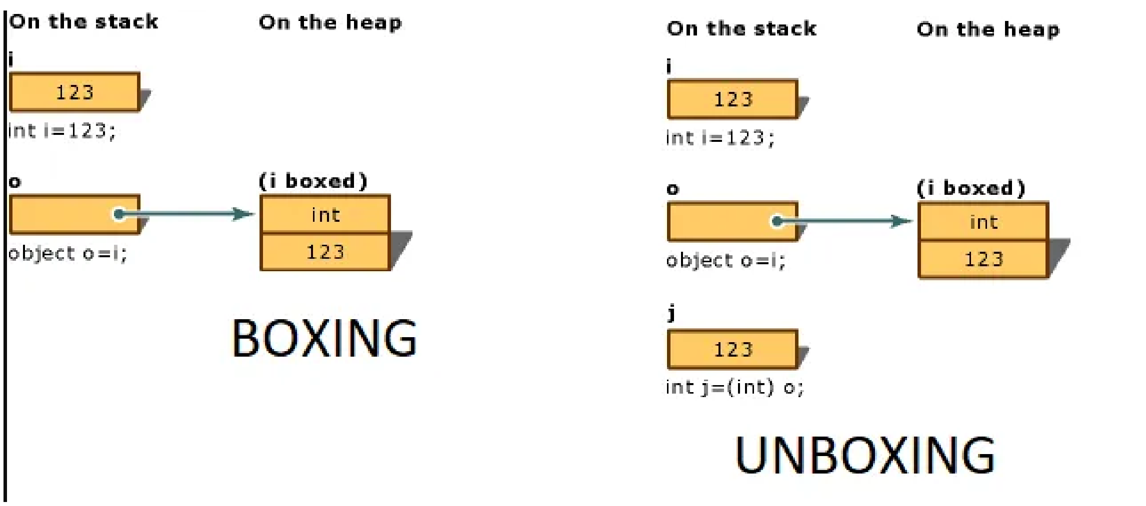
Day Three

▼ boxing Vs Unboxing , Nullable Value Type , Generics

▼ boxing Vs unboxing

- 

- In simple terms:
  - Boxing implies storing a value type as an object = box value type in System.Object
  - Unboxing = read the value from an object = unbox value from the object
- Now the official definitions:
  - 1. Boxing
    - Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type.
      .NET CLR boxes a value type i.e., wraps it in System.Object instance and stores it on the

    managed(garbage-collected) heap.
    Boxing is an implicit operation.

  - Unboxing
    - Unboxing is an explicit conversion from the type object to a value type or from an interface type to a value type that implements the interface. Unboxing extracts the value type from the object.
    - An unboxing operation consists of:
    - Checking the object instance to make sure that it is a boxed value of the given value type.
    - Copying the value from the instance into the value-type variable.
    - Attempting to unbox null causes a NullReferenceException.
    - Attempting to unbox a reference to an incompatible value type causes an InvalidCastException.
- **Problems with Boxing and Unboxing**
  - **Performance Overhead**: Boxing and unboxing involve additional memory allocation and CPU cycles.
  - **Type Safety**: Unboxing requires explicit casting, which can lead to runtime errors if the types don't match.
- Performance -Wise
  - Boxing and unboxing operations affect the performance as they are computationally expensive processes.
    - When a value type is boxed, a new object must be allocated and constructed. To a lesser degree, the cast required for unboxing is also expensive computationally.

```
Console.WriteLine("=== Boxing and Unboxing in Action ==

        // Step 1: Demonstrate Boxing
        int valueType = 42;
        object boxedValue = valueType; // Implicit boxin
        Console.WriteLine($"Boxed Value (object): {boxed

        // Step 2: Demonstrate Unboxing
        try
        {
            int unboxedValue = (int)boxedValue; // Expli
            Console.WriteLine($"Unboxed Value (int): {un
        }
        catch (InvalidCastException ex)
        {
            Console.WriteLine($"Error during unboxing: {
        }

        // Step 3: Incorrect Unboxing
        try
        {
            object boxedDouble = 42.5; // Boxed double
            int invalidUnboxing = (int)boxedDouble; // W
        }
        catch (InvalidCastException ex)
        {
            Console.WriteLine($"Type Mismatch Error: {ex
        }

        // Step 4: Performance Demonstration
        Console.WriteLine("\n=== Performance Impact ==="
        int iterations = 1_000_000;
        object box;
        var startTime = DateTime.Now;
```

```
        for (int i = 0; i < iterations; i++)
        {
            box = i; // Boxing
        }

        var endTime = DateTime.Now;
        Console.WriteLine($"Time taken for {iterations}

        // Step 5: Avoid Boxing with Generics
        Console.WriteLine("\n=== Avoiding Boxing with Ge
        PrintGenericValue(42); // No boxing
```

▼ Nullable Value Types

- Definition : Value types in C# have an inherent limitation: by definition, they represent a value and therefore cannot be null. But what if you genuinely need to represent the absence of a value? This is where nullable value types come in, introduced with C# 2.0.
- **What are Nullable Value Types?**
    - In C#, value types (e.g., `int`, `double`, `bool`, `struct`) are types that have a value or are uninitialized. They cannot be set to null. Nullable value types bridge this gap by allowing a value type to be `null`.
    - **Declaring a Nullable Value Type**

    ```
    int? nullableInt = null;
    double? nullableDouble = 123.45;
    bool? flag = null;
    ```

    - **Value and HasValue Properties:**

- The `Nullable<T>` struct provides two handy properties:
  - • `HasValue`: Returns `true` if the nullable type contains a value; `false` if it contains `null`.
  - • `Value`: Returns the value of the nullable type if `HasValue` is `true`. If `HasValue` is `false`, accessing `Value` will throw an `InvalidOperationException`.
  - **Null Coalescing and Nullable Value Types**
    - The null coalescing operator (`??`) is particularly useful with nullable value types. It returns the left-hand operand if it's not null; otherwise, it returns the right operand.
    -

```
int? a = null;
int b = a ?? -1;  // b will be set to -1
```

  - **Null Conditional Operator**
    - The null conditional operator (`?.`) allows for short-circuiting when dealing with nullable value types:
  - Conversion
    - You can convert between a nullable value type and its underlying type:
  - Nullable value types are still stored on the stack, but with an additional flag indicating whether they have a value or are `null`.

```
int? a = 5; // Nullable int with value 5
int? b = null; // Nullable int with value null
```

```
int c = (int)a; // Unboxing a nullable value ty
```

- **Casting Nullable to Non-Nullable**:
    -

```
int? nullableInt = 5;
int nonNullableInt = nullableInt.Value; // Expl
```

- **Handling Null Values**:
    -

```
int? nullableInt = null;
int nonNullableInt = nullableInt ?? 0; // Use t
```

- Example

```
using System;

public class NullableValueTypesDemo
{
    public static void Main()
    {
        Console.WriteLine("=== Nullable Valu
e Types with Boxing/Unboxing ===");

        // Step 1: Declaring Nullable Value
Types
        int? nullableInt = null;
        double? nullableDouble = 123.45;
        bool? nullableBool = null;
```

```
        // Step 2: Checking HasValue and Val
ue Properties
        Console.WriteLine($"nullableInt has
value: {nullableInt.HasValue}");
        Console.WriteLine($"nullableDouble h
as value: {nullableDouble.HasValue}");

        if (nullableDouble.HasValue)
        {
            Console.WriteLine($"Value of nul
lableDouble: {nullableDouble.Value}");
        }

        // Step 3: Using the Null-Coalescing
Operator
        int defaultInt = nullableInt ?? -1;
// Assign -1 if nullableInt is null
        Console.WriteLine($"Value of default
Int: {defaultInt}");

        // Step 4: Null Conditional Operator
        int? calculatedValue = nullableInt?.
GetHashCode(); // Safely handle nullable typ
es
        Console.WriteLine($"Nullable int has
h code: {calculatedValue ?? 0}");

        // Step 5: Casting Nullable to Non-N
ullable
        int? nullableIntWithValue = 10;
        int nonNullableInt = nullableIntWith
Value ?? 0; // Use null-coalescing operator
        Console.WriteLine($"Non-nullable in
t: {nonNullableInt}");
```

```
        // Step 6: Boxing and Unboxing with
Nullable Types
        object boxedNullableInt = nullableIn
tWithValue; // Boxing
        Console.WriteLine($"Boxed nullable i
nt: {boxedNullableInt}");

        try
        {
            int unboxedValue = (int)boxedNul
lableInt; // Unboxing
            Console.WriteLine($"Unboxed valu
e: {unboxedValue}");
        }
        catch (InvalidCastException ex)
        {
            Console.WriteLine($"Error during
unboxing: {ex.Message}");
        }

        // Step 7: Handling Null Values Duri
ng Unboxing
        object boxedNull = nullableInt; // B
oxing a null value
        Console.WriteLine($"Boxed null: {box
edNull ?? "null"}");

        try
        {
            int unboxedNull = (int)boxedNul
l; // Attempting to unbox null
            Console.WriteLine($"Unboxed nul
l: {unboxedNull}");
        }
        catch (NullReferenceException ex)
        {
```

```
            Console.WriteLine($"Error during
unboxing null: {ex.Message}");
        }

        Console.WriteLine("=== End of Demo =
==");
    }
}
```

- Example

▼ Generics

- **Introduction**
    - **Generics**: Allow you to define classes, methods, delegates, and interfaces with a placeholder for the data type.
- Benefits of Generics:
    - **Type Safety**: Compile-time checking, reducing runtime errors.
    - **Performance**: Avoids boxing/unboxing as the types are specified at compile time.
    - **Code Reusability**: Write a method or class once and reuse it with different data types.
    - Example

```
public class Box<T>
{
    private T _value;
    public Box(T value)
    {
        _value = value;
    }
```

```
    public T GetValue()
    {
        return _value;
    }
}

Box<int> intBox = new Box<int>(123);
Box<string> strBox = new Box<string>("Hello");

public T Add<T>(T a, T b) where T : struct
{
    return a + b; // Assume operator + is defined for T
}
```

- Generic Constraints
    - `where` **Clause**
        - `where T : struct` (T must be a value type)
        - `where T : class` (T must be a reference type)
        - `where T : new()` (T must have a parameterless constructor)
        - `where T : SomeBaseClass` (T must inherit from `SomeBaseClass`)
    - Example

```
public class GenericExample<T> where T : class, new()
{
    public T CreateInstance()
    {
        return new T(); // Requires T to have a paramete
    }
}
```

- **Memory-Wise**:

- **No Boxing/Unboxing**: Since types are determined at compile time, generics avoid the overhead associated with boxing/unboxing.
- **Performance**: Generics are optimized at runtime, providing both type safety and performance benefits.

- Without generics
  - Using System.Object

```
using System;

public class ObjectBox
{
    private object _value;

    public ObjectBox(object value)
    {
        _value = value;
    }

    public object GetValue()
    {
        return _value;
    }
}

class Program
{
    static void Main()
    {
        ObjectBox intBox = new ObjectBox(123); //
Boxing
        ObjectBox strBox = new ObjectBox("Hello");

        // Retrieving values
```

```
        Console.WriteLine($"Integer: {intBox.GetVa
lue()}");
        Console.WriteLine($"String: {strBox.GetVal
ue()}");

        // Type safety issue
        try
        {
            int value = (int)strBox.GetValue(); //
Runtime InvalidCastException
        }
        catch (InvalidCastException ex)
        {
            Console.WriteLine($"Error: {ex.Messag
e}");
        }
    }
}

Problems:
//Lack of type safety: No compile-time checks; all
type validation happens at runtime.
//Performance overhead: Boxing/unboxing when deali
ng with value types.
//Developer errors: Misusing the wrong type (e.g.,
casting string to int).
```

- Array List

```
using System;
using System.Collections;

class Program
{
    static void Main()
```

```
    {
        ArrayList list = new ArrayList();
        list.Add(1); // Boxing
        list.Add("Hello");

        // Type safety issue
        foreach (object item in list)
        {
            try
            {
                int value = (int)item; // Runtime
InvalidCastException
                Console.WriteLine($"Value: {valu
e}");
            }
            catch (InvalidCastException ex)
            {
                Console.WriteLine($"Error: {ex.Mes
sage}");
            }
        }
    }
}

//ArrayList:
//Holds objects of any type, but no compile-time t
ype checking.
//Example Issue: Mixing integers and strings cause
s runtime errors during type casting.
```

- With generic
    -

```
using System;

```

```
public class Box<T>
{
    private T _value;

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

    public T GetValue()
    {
        return _value;
    }
}

class Program
{
    static void Main()
    {
        Box<int> intBox = new Box<int>(123);
        Box<string> strBox = new Box<string>("Hell
o");

        // Type safety
        Console.WriteLine($"Integer: {intBox.GetVa
lue()}");
        Console.WriteLine($"String: {strBox.GetVal
ue()}");

        // No need for explicit casting
        int value = intBox.GetValue(); // Safe and
no runtime exceptions
        Console.WriteLine($"Safe Value: {value}");
    }
}
```

- Generic Constraint
  -

```
using System;

public class Factory<T> where T : class, new()
{
    public T CreateInstance()
    {
        return new T(); // Requires parameterless constructor
    }
}

class Program
{
    static void Main()
    {
        Factory<SampleClass> factory = new Factory<SampleClass>();
        SampleClass instance = factory.CreateInstance();
        Console.WriteLine($"Instance created: {instance.GetType().Name}");
    }
}

public class SampleClass
{
}
```

▼ Inventory Management System

- **problem** :

You are tasked with building an inventory management system to handle multiple types of products (e.g., Electronics, Groceries, and Furniture). The system should:

1. Store products in an array.
2. Allow adding, retrieving, and displaying product details.
3. Calculate the total cost of all products in the inventory.

- **Constraints:**
    - Without Generics, you must rely on `System.Object` to handle various product types.
    - This approach will showcase the difficulties and type safety issues.

▼ Without Generics

- product
    - class

```
public class Product
{
    public string Id { get; set; }
    public string Name { get; set; }
    public double Price { get; set; }

    public Product(string id, string name, double p
rice)
    {
        Id = id;
        Name = name;
        Price = price;
    }
```

```
    public virtual string GetDetails()
    {
        return $"ID: {Id}, Name: {Name}, Price: {Pr
ice}";
    }
}
```

- Electronics
    - class

```
public class Electronics : Product
{
    public int WarrantyYears { get; set; }

    public Electronics(string id, string name, doub
le price, int warrantyYears)
        : base(id, name, price)
    {
        WarrantyYears = warrantyYears;
    }

    public override string GetDetails()
    {
        return base.GetDetails() + $", Warranty: {W
arrantyYears} years";
    }
}
```

- Groceries
    - Class

```
public class Groceries : Product
{
    public string ExpiryDate { get; set; }
```

```
    public Groceries(string id, string name, double
price, string expiryDate)
        : base(id, name, price)
    {
        ExpiryDate = expiryDate;
    }

    public override string GetDetails()
    {
        return base.GetDetails() + $", Expiry Date:
{ExpiryDate}";
    }
}
```

- Furniture
  - Class

```
public class Furniture : Product
{
    public string Material { get; set; }

    public Furniture(string id, string name, double
price, string material)
        : base(id, name, price)
    {
        Material = material;
    }

    public override string GetDetails()
    {
        return base.GetDetails() + $", Material: {M
aterial}";
```

```
    }
}
```

- Inventory Manager Without Generics
    -

```
public class InventoryManager
{
    private object[] inventory; // Using System.Obj
ect for all product types
    private int index;

    public InventoryManager(int capacity)
    {
        inventory = new object[capacity];
        index = 0;
    }

    public void AddProduct(object product)
    {
        if (index >= inventory.Length)
        {
            Console.WriteLine("Inventory is ful
l.");
            return;
        }

        inventory[index++] = product;
    }

    public void DisplayAllProducts()
    {
        foreach (object item in inventory)
        {
            if (item is Product product)
```

```
            {
                Console.WriteLine(product.GetDetail
s());
            }
            else
            {
                Console.WriteLine("Invalid item in
inventory.");
            }
        }
    }

    public double CalculateTotalValue()
    {
        double totalValue = 0;

        foreach (object item in inventory)
        {
            if (item is Product product)
            {
                totalValue += product.Price;
            }
            else
            {
                Console.WriteLine("Invalid item in
inventory.");
            }
        }

        return totalValue;
    }
}
```

- Main Program
    -

```
public class Program
{
    public static void Main()
    {
        InventoryManager manager = new InventoryMan
ager(5);

        // Adding products
        manager.AddProduct(new Electronics("E001",
"Laptop", 1500, 2));
        manager.AddProduct(new Groceries("G001", "M
ilk", 3.5, "2025-01-01"));
        manager.AddProduct(new Furniture("F001", "C
hair", 120, "Wood"));
        manager.AddProduct("InvalidItem"); // Delib
erate error

        // Display all products
        Console.WriteLine("=== Inventory ===");
        manager.DisplayAllProducts();

        // Calculate total value
        double totalValue = manager.CalculateTotalV
alue();
        Console.WriteLine($"Total Inventory Value:
{totalValue}");
    }
}
```

▼ Refactored Solution With Generics

- Generic Inventory Manager

```
public class InventoryManager<T> where T : Product
{
    private T[] inventory;
```

```
    private int index;

    public InventoryManager(int capacity)
    {
        inventory = new T[capacity];
        index = 0;
    }

    public void AddProduct(T product)
    {
        if (index >= inventory.Length)
        {
            Console.WriteLine("Inventory is full.");
            return;
        }

        inventory[index++] = product;
    }

    public void DisplayAllProducts()
    {
        foreach (T product in inventory)
        {
            if (product != null)
            {
                Console.WriteLine(product.GetDetails
());
            }
        }
    }

    public double CalculateTotalValue()
    {
        double totalValue = 0;

        foreach (T product in inventory)
```

```
        {
            if (product != null)
            {
                totalValue += product.Price;
            }
        }

        return totalValue;
    }
}
```

- Main Class

```
public class Program
{
    public static void Main()
    {
        InventoryManager<Product> manager = new InventoryManager<Product>(5);

        // Adding products
        manager.AddProduct(new Electronics("E001", "Laptop", 1500, 2));
        manager.AddProduct(new Groceries("G001", "Milk", 3.5, "2025-01-01"));
        manager.AddProduct(new Furniture("F001", "Chair", 120, "Wood"));
        // manager.AddProduct("InvalidItem"); // Compile-time error!

        // Display all products
        Console.WriteLine("=== Inventory ===");
        manager.DisplayAllProducts();

        // Calculate total value
```

```
        double totalValue = manager.CalculateTotalValue();
        Console.WriteLine($"Total Inventory Value: {totalValue}");
    }
}
```

▼ Exploring Generics and IClonable Interface

- **Problem Statement:** You are tasked with creating a library management system that allows managing a collection of books. The system should:
  1. Use a **generic repository** to store, retrieve, and filter books by various criteria.
  2. Implement a `Book` class that supports cloning through the `IClonable` interface, allowing deep and shallow copies of book instances.
  3. Use the generic repository to demonstrate adding, retrieving, and filtering books.
  4. Include unit tests or a demonstration that validates the repository and cloning functionality.

- Requirements:
  - **Part 1: Implement the `Book` Class**
    - Create a Book class with the following properties:
      - `Id` (string)
      - `Title` (string)
      - `Author` (string)
      - `Price` (decimal)
      - `Genres` (string[])

            - Implement the `ICloneable` interface:
                - **Shallow Clone:** Create a shallow copy of the `Book` instance.
                - **Deep Clone:** Create a deep copy of the `Book` instance, including the `Genres` Array.
    - **Part2: Generic Repository**
        - Create a generic `Repository<T>` class that supports
            - Adding items (`Add(T item)`).
            - Removing items (`Remove(T item)`).
            - Retrieving all items (`GetAll()`).
    - **Part 3: Demonstration**
        - Add a `Program.cs` file to
            - Create a list of `Book` objects.
            - Add them to the generic repository.
            - Clone a book instance and show both shallow and deep cloning in action.

<u>Assignment (Generic Currency Range) :</u>