Day One

▼ 1. What Is an Interface?

- **Definition**
  - An **interface** is a **contract** specifying a set of members (methods, properties, indexers, events).
  - It **does not** provide any implementation details (until C# 7).
  - Any class or struct that **implements** an interface **must** implement **all** interface members.
- **Benefits**
  - **Decoupling**: Write code against abstractions, not specific classes.
  - **Polymorphism**: Multiple classes can implement the same interface differently.
  - **Testability**: Interfaces make it easy to replace concrete implementations with mocks/stubs during testing.

▼ 2. Interface vs. Abstract Class

Aspect	Interface	Abstract Class
Primary Purpose	Define Behavioral Requirements (a Contract)	Serves as a *base class* with optional partial implementations
Implementation	Pre-C# 8: no default implementation; C# 8+ allows default methods sparingly	Can contain both *abstract* (unimplemented) and *concrete* (implemented) members
Fields	**Not** allowed to have instance fields	Can contain fields, constructors, destructors

Aspect	Interface	Abstract Class
Multiple Inheritance	A class can implement **multiple** interfaces	A class can only inherit **one** abstract class
When to Use	When you need a common contract across potentially *unrelated* classes	When you have a shared base of functionality or data

Key Takeaway

- **Interfaces** = "What must be done?"
- **Abstract Classes** = "Common base + partial implementation to build on."

▼ 3. Common .NET Interfaces

▼ 3.1 `IComparable<T>`

- **Purpose**: Enables custom sorting logic by implementing `CompareTo(T)`.
- **Usage**
  - **Sorting arrays** (or other collections).
  - `Array.Sort(myArray)` calls `CompareTo` for each pair of elements to determine their correct order.
- **Key Points**
  - `CompareTo` should return:
    - **< 0** if *this* is "less" than *other*.
    - **0** if *this* is "equal" to *other*.
    - **> 0** if *this* is "greater" than *other*.

▼ Array Example using `IComparable<T>`

-

```
using System;

public class Person : IComparable<Person>
{
    public string Name { get; set; }

    public Person(string name)
    {
        Name = name;
    }

    // IComparable<Person> implementation
    public int CompareTo(Person? other)
    {
        // Handle null references safely
        if (other == null) return 1;

        // Compare by Name alphabetically
        return string.Compare(this.Name, other.Name, Str
    }
}

public class Program
{
    public static void Main()
    {
        Person[] people =
        {
            new Person("Charlie"),
            new Person("Alice"),
            new Person("Bob")
        };

        Console.WriteLine("Before sorting by Name (IComp
        PrintArray(people);
```

```
        // Sort using CompareTo logic in the Person clas
        Array.Sort(people);

        Console.WriteLine("\nAfter sorting by Name (ICom
        PrintArray(people);
    }

    static void PrintArray(Person[] arr)
    {
        foreach (var person in arr)
        {
            Console.WriteLine($"  {person.Name}");
        }
    }
}
```

- `Array.Sort(people)` calls `people[i].CompareTo(people[j])` under the hood.
- The `CompareTo` method orders the `Person` objects by `Name` in ascending alphabetical order.

▼ 3.2 `IComparer<T>`

- **Purpose** : Defines an **external** comparison strategy in a separate class. This is useful if you want **multiple** ways to sort the same object type *without* altering the object's own CompareTo method.
- **Usage** : If you need different sorting logic for the **same** type (e.g., sort by Name or by Age, depending on the situation).

▼ Array Example using `IComparer<T>`

- we have two ways to sort the **same** `Person` array:

1. By **Name** (using `IComparable<Person>` in the class).

2. By **Age** (using a separate `AgeComparer` class that implements `IComparer<Person>`).

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

public class Person : IComparable<Person>
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    // Default sort by Name (alphabetically)
    public int CompareTo(Person? other)
    {
        if (other == null) return 1;
        return string.Compare(this.Name, other.Name, Str
    }
}

// Separate comparer to sort by Age
public class AgeComparer : IComparer<Person>
{
    public int Compare(Person? x, Person? y)
    {
        if (x == null && y == null) return 0;
        if (x == null) return -1;
        if (y == null) return 1;
```

```
        // Sort ascending by Age
        return x.Age.CompareTo(y.Age);
    }
}

public class Program
{
    public static void Main()
    {
        Person[] people =
        {
            new Person("Charlie", 25),
            new Person("Alice", 20),
            new Person("Bob", 30)
        };

        Console.WriteLine("People array (unsorted):");
        PrintArray(people);

        // 1) Sort by default (Name) - uses IComparable<
        Array.Sort(people);
        Console.WriteLine("\nSorted by Name (default Com
        PrintArray(people);

        // 2) Sort by Age - uses IComparer<Person>
        Array.Sort(people, new AgeComparer());
        Console.WriteLine("\nSorted by Age (IComparer<Pe
        PrintArray(people);
    }

    static void PrintArray(Person[] arr)
    {
        foreach (var person in arr)
        {
            Console.WriteLine($"  {person.Name}, Age: {p
        }
```

```
    }
}
```

- **How it works**:
  1. `Array.Sort(people)` (without a comparer) uses **`Person.CompareTo`** and sorts by **Name**.
  2. `Array.Sort(people, new AgeComparer())` uses **`AgeComparer.Compare`** and sorts by **Age**.

Why Use `IComparable<T>` and `IComparer<T>` With Arrays?

- **Flexibility**
  - If your only data structure is an **array**, you can still leverage powerful sorting operations without writing custom loops.
  - `Array.Sort` is built-in and efficient; just supply the appropriate logic.
- **Maintainable Code**
  - If your sorting logic changes (e.g., from sorting by Name to Age), you don't have to rewrite array manipulation code—just change the `CompareTo` method or swap in a different `IComparer`.
- **Reusability**
  - You can keep different comparers in separate classes—no need to edit the `Person` class whenever you want a new sorting rule.
- **Performance**
  - Built-in `Array.Sort` is highly optimized.
  - `IComparable<T>` and `IComparer<T>` are recognized by .NET's sorting algorithms for better performance than manual iterations.

3.3 `ICloneable`

- **Purpose**: Standard mechanism to create an object's copy via `Clone()`.
- **Usage**:
  - Quickly duplicate an object.
  - **Note**: Doesn't specify *deep* vs. *shallow* copy (you decide).
  - Often used in **prototype** or **templating** scenarios where you need to quickly duplicate objects.

Why Might You Need a Deep Copy?

- Prevent Shared References
  - If you have **nested objects** (e.g., a `Person` with an `Address` object), a *shallow* copy only duplicates **top-level fields**, leaving references to the **same** sub-objects.
  - A **deep** copy duplicates *all* nested objects, ensuring the clone is fully independent.
- Temporary State Manipulation
  - You may want a "sandbox" copy to safely modify data (like a "what-if" scenario) without affecting the original object.

Deep Copy with `ICloneable` Code Example

```
using System;

namespace DeepCloneDemo
{
    // Represents an Address, which will be deeply co
    public class Address
```

```
    {
        public string Street { get; set; } = string.En
        public string City { get; set; } = string.Empt

        // Copy constructor (an easy way to help with
        public Address(Address other)
        {
            Street = other.Street;
            City = other.City;
        }

        // Parameterless constructor for convenience
        public Address() { }
    }

    // Person class that contains an Address object
    public class Person : ICloneable
    {
        public string Name { get; set; } = string.Empt
        public Address HomeAddress { get; set; } = new

        // Parameterless constructor for convenience
        public Person() { }

        // Constructor that accepts name and address
        public Person(string name, Address address)
        {
            Name = name;
            HomeAddress = address;
        }

        // ICloneable implementation: deep copy
        public object Clone()
        {
            // 1) Create a new Person
            var clone = new Person();
```

```
        // 2) Copy primitive fields (Name, etc.)
        clone.Name = this.Name;

        // 3) Create a new Address instance instea
        clone.HomeAddress = new Address(this.HomeA

        // Return the fully-cloned Person
        return clone;
    }
}

public class Program
{
    public static void Main()
    {
        Console.WriteLine("=== Deep Clone Demo (IC

        // Original person
        var original = new Person("Alice", new Add

        // Clone the person
        var cloned = (Person) original.Clone();

        // Change data in the cloned object
        cloned.Name = "Alicia";
        cloned.HomeAddress.City = "Mirrorland";

        Console.WriteLine("Original Person:");
        Console.WriteLine($"   Name: {original.Nar

        Console.WriteLine("\nCloned Person:");
        Console.WriteLine($"   Name: {cloned.Name]

        // Notice that the Original object's Addre
        // proving we have a deep copy.
```

```
            Console.WriteLine("\nPress any key to exit
            Console.ReadKey();
        }
    }
}
```

- ## Step-by-Step Explanation
  1. **`Address` Class**
     - Has two properties: `Street` and `City`.
     - Provides a **copy constructor** `public Address(Address other)` to help with the **deep** copy operation.
  2. **`Person` Class**
     - Implements `ICloneable` and overrides `Clone()` to create a new `Person`.
     - **Deep copy** logic:
       - Duplicates the **simple** field (`Name`).
       - Creates a **new** `Address` by calling `new Address(this.HomeAddress)`.
       - This ensures that `Person.HomeAddress` is **not** the same reference as the original.
  3. **Deep vs. Shallow**
     - If we had just used `MemberwiseClone()`, the `Address` object would be **shared** between the original and cloned `Person`.
     - By manually creating a new `Address`, we ensure changes to the cloned address **do not** affect the original.
  4. **Test**

- After cloning, we modify `cloned.HomeAddress.City`.
- Observe that the original `Address` remains **unchanged**, confirming a proper deep copy.

▼ When Might I Use This?

1. **Prototype or Templating**
   - If you frequently create new objects based on an “existing template,” deep cloning can be simpler than re-initializing from scratch.
2. **Isolation of Changes**
   - In data processing or business logic, you might want to “try out” transformations on a cloned object. If something goes wrong, you **discard the clone** and keep the safe original.
3. **Simulation / ‘What-If’ Scenarios**
   - Cloning allows you to modify a copy while leaving the original data intact, enabling back-testing or scenario analysis.
4. **Legacy or Shared Code Requirements**
   - Some frameworks or libraries might expect `ICloneable`, so implementing it can make your code more compatible.

- **Note**: In modern C#, many developers prefer **copy constructors**, **factory methods**, or even **records** for simpler copying semantics. Still, `ICloneable` can be a quick standard mechanism if you explicitly define how deep or shallow the clone is.

▼ 3.4 `IEquatable<T>`

- **Why Is It Better Than `object.Equals` Alone?**

1. **No Casting**: With `object.Equals`, you often have to *cast* or *pattern match* to your specific type. `IEquatable<T>` is already strongly typed.
2. **Performance**: Avoid boxing and extra runtime checks; `.NET` can call `Equals(T other)` directly.
3. **Generics**: Data structures like `List<T>` and `Dictionary<TKey,TValue>` can use `IEquatable<T>` to compare elements **more efficiently**.

- **Where Do We Use** `IEquatable<T>`
    - **Domain Models:** If you have a `Customer` class and you consider two `Customer` instances "equal" if they have the same `CustomerId`, `IEquatable<Customer>` can encode that domain-specific equality logic.

▼ How `IEquatable<T>` Works with Dictionaries

- Hash-Based Lookup Process
    - When you insert an object (key) into a **`Dictionary<TKey, TValue>`** or **`HashSet<T>`**:
        - **`GetHashCode()`** is called to determine which **bucket** the key should go into.
        - If multiple keys have the **same** hash code (collision), the data structure calls **`Equals()`** to decide if they are truly the same object.
    - **Key Point**: **`Dictionary<TKey, TValue>`** or **`HashSet<T>`** always calls **`GetHashCode()`** first, then **`Equals()`** if it finds a collision in that hash bucket.

- **Ensuring Consistency** :
    - You **must** ensure that if **two objects are equal** by `Equals(T other)`, they **return the same** hash code in `GetHashCode()`. Otherwise, dictionary lookups (and other equality-based operations) break or produce unexpected behavior.

▼ Full Implementation Steps

To fully and properly implement **`IEquatable<T>`**:

1. **Implement the `bool Equals(T other)` Method**
    - Compare the fields/properties you consider relevant for equality.
    - Decide whether `null` checks, case-insensitivity, etc., are important.
2. **Override `bool Equals(object obj)`**
    - Typically, call your `Equals(T other)` method after casting or pattern matching.
    - This ensures that **all** equality checks, even those via a `base object` reference, are consistent.
3. **Override `int GetHashCode()`**
    - Must be consistent with the logic used in `Equals`.
    - For multi-field objects, prefer `HashCode.Combine(...)` or another robust method.
4. **Consider** Marking the Type as `sealed` or `record`
    - If the class is **not** meant to be inherited, `sealed` can avoid complex equality issues in subclasses.
    - If you use a **record** (C# 9+), the compiler can auto-generate equality members, but you can still tweak

them as needed.

▼ **Example**

```
using System;

namespace EquatableDemo
{
    public sealed class Product : IEquatable<Product>
    {
        public int Id { get; init; }
        public string Name { get; init; } = string.Emp
        public string Manufacturer { get; init; } = st
        public decimal Price { get; init; }
        public DateTime ReleaseDate { get; init; }
        public string Category { get; init; } = string

        public Product(int id, string name, string mar
        {
            Id = id;
            Name = name;
            Manufacturer = manufacturer;
            Price = price;
            ReleaseDate = releaseDate;
            Category = category;
        }

        // 1) IEquatable<Product> implementation:
        public bool Equals(Product? other)
        {
            // A) Check if we're comparing with the sa
            if (ReferenceEquals(this, other)) return t

            // B) If 'other' is null, they're not equa
            if (other is null) return false;
```

```
            // C) Since this class is sealed, a type c
            //    But to illustrate best practice:
            if (this.GetType() != other.GetType()) ret

            // D) Compare each field that matters for
            //    We'll do a case-sensitive match here
            return (this.Id == other.Id)
                && (this.Name == other.Name)
                && (this.Manufacturer == other.Manufac
                && (this.Price == other.Price)
                && (this.ReleaseDate == other.ReleaseD
                && (this.Category == other.Category);
        }

        // 2) Override object.Equals(object?)
        public override bool Equals(object? obj)
        {
            // Pattern match to the strongly typed Equ
            return obj is Product product && Equals(pr
        }

        // 3) Override GetHashCode to include all fiel
        public override int GetHashCode()
        {
            // Combine all fields to avoid collisions
            // The order of combining doesn't matter,
            return HashCode.Combine(
                Id,
                Name,
                Manufacturer,
                Price,
                ReleaseDate,
                Category
            );
        }
    }
```

```
public class Program
{
    public static void Main()
    {
        // Create an array of products
        var products = new Product[]
        {
            new Product(
                id: 101,
                name: "Super Phone",
                manufacturer: "GizmoCorp",
                price: 699.99m,
                releaseDate: new DateTime(2025, 01
                category: "Mobile"
            ),
            new Product(
                id: 102,
                name: "Noise Cancelling Headphones
                manufacturer: "Soundify",
                price: 199.99m,
                releaseDate: new DateTime(2025, 02
                category: "Audio"
            ),
            new Product(
                id: 103,
                name: "Smart Watch",
                manufacturer: "WearableTech",
                price: 299.99m,
                releaseDate: new DateTime(2025, 03
                category: "Wearables"
            )
        };

        // Search for a product with the same data
        // Change some text if you want to see how
```

```
                var searchProduct = new Product(
                    id: 101,
                    name: "Super Phone",
                    manufacturer: "GizmoCorp",
                    price: 699.99m,
                    releaseDate: new DateTime(2025, 01, 01
                    category: "Mobile"
                );

                // Array.IndexOf uses Equals() under the h
                int index = Array.IndexOf(products, search

                if (index >= 0)
                {
                    Console.WriteLine($"Product found at i
                }
                else
                {
                    Console.WriteLine($"Product with Id={s
                }
            }
        }
}
```

- **Consistency**
    - Always ensure `Equals(T other)` and `Equals(object?)` **agree**.
    - If `obj is T typedObj`, then `Equals(typedObj)` should yield the same result.
- **Stable Hash Code**
    - Include **all** the fields in `GetHashCode()` that are used in `Equals()`.
    - Changing any of those fields should theoretically produce a different hash code

(though collisions can still happen, that’s normal).

- **Immutability (Recommended)**
    - If the fields used in equality can **change** after object creation, the hash code can become **invalid** in hash-based collections.
    - Consider making your objects **immutable** (e.g., use `init` or read-only properties).
    - Alternatively, avoid storing mutable objects as keys in dictionaries or sets.
- **Use `record`** for Simple Scenarios
    - If your class is just storing data (DTO, small entity), records (C# 9+) automatically handle equality, hashing, and immutability.
    - But if you need custom logic, you can still override the generated members or implement `IEquatable<T>` explicitly.
- **Document the Equality Semantics**
    - Others reading your code should know which fields matter for equality.
    - If partial matching or case-insensitivity are important, note it clearly in the doc comments.