1:

```
using System;


class Program
{
   // نعرف enum اسمه Weekdays
   enum Weekdays
   {
      Monday = 1,
     Tuesday,
      Wednesday,
      Thursday,
      Friday ,
   }


   static void Main()
   {
      foreach (Weekdays day in Enum.GetValues(typeof(Weekdays)))
      {
         Console.WriteLine($"{day} = {(int)day}");
      }
   }
}
```

- لأن الـ enum افتراضيًا بيبدأ من **0** ويزود +1 لكل عنصر.
- لكن ساعات في تطبيقات معينة) زي قواعد البيانات أو التعامل مع (API بيكون عندنا أرقام محددة لازم نطابقها مع الأيام.
  مثلاً:
  - لو الجدول في قاعدة البيانات مسجل "Monday = 1" لازم نخلي الـ enum يبدأ من 1 مش 0.
- كمان كتابة القيم صراحة بتخلي الكود أوضح، ومايبقاش فيه لبس لو أضفنا قيم جديدة في النص.

2:

```
using System;
class Program
{
   enum Grades : short
   {
      F = 1,
      D= 2,
      C= 3,
      B= 4,
      A= 5,
   }
   static void Main()
   {
      foreach (Grades grade in Enum.GetValues(typeof(Grades)))
      {
         Console.WriteLine($"{grade} = {(short)grade}");
      }
   }
}
```

- النوع `short` مداه من **32,768-لحد 32,767**.
- لو حاولنا ندي قيمة أكبر من كده (مثلاً 40,000)، الكومبايلر هيدي **Error** وقت الترجمة (compile-time error) ومش هيقبل الكود.
- يعني **#C بتتحقق من القيم أثناء الكومبايل مش في وقت التشغيل**.

3:

```
using System;
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Department { get; set; } // الخاصية الجديدة
}
class Program
{
    static void Main()
    {
        Person p1 = new Person();
        p1.Name = "Ahmed";
        p1.Age = 25;
        p1.Department = "IT";

        Person p2 = new Person();
        p2.Name = "Sara";
        p2.Age = 22;
        p2.Department = "HR";

        Console.WriteLine($"Name: {p1.Name}, Age: {p1.Age}, Department: {p1.Department}");
```

```
        Console.WriteLine($"Name: {p2.Name}, Age: {p2.Age}, Department: {p2.Department}");
    }
}
```

- الكلمة `virtual` معناها إن الخاصية دي ممكن تتعملها **Override** في أي كلاس وارث.(Derived Class)
- يعني لو عندنا كلاس أساسي (Base Class) فيه خاصية `virtual`، نقدر نعيد تعريفها (override) في كلاس الابن ونغير سلوكها.
- ده بيدينا **مرونة في الوراثة** وبيطبق مبدأ الـ) **Polymorphism** تعدد الأشكال(

4:

```
using System;

class Parent
{
    public virtual int Salary { get; set; } = 5000;
}

class Child : Parent
{
    public sealed override int Salary { get; set; } = 7000;

    public void DisplaySalary()
    {
        Console.WriteLine($"Salary: {Salary}");
    }
```

```
}

class Program
{
   static void Main()
   {
      Child c = new Child();
      c.DisplaySalary();
   }
}
```

- لما نستخدم **sealed** مع خاصية أو دالة، معناها إننا قفلنا الباب على أي كلاس تاني يرث ويعمل **override** لها.
- السبب: علشان نثبت السلوك ونمنع تغييره في الكلاسات الأبعد، وده بيدي أمان أكتر للكود وبيحافظ على القاعدة (base logic)

5:

```
using System;

class Utility
{
   public static int CalculatePerimeter(int length, int width)
   {
```

```
        return 2 * (length + width);
    }
}

class Program
{
    static void Main()
    {
        int result = Utility.CalculatePerimeter(10, 5);
        Console.WriteLine($"Perimeter: {result}");
    }
}
```

- **Static members**: مرتبطة بالكلاس نفسه مش بالكائن، وبتتندَه عليها مباشرة من غير ما نعمل `new`.

Object members: محتاجة نعمل كائن (object) من الكلاس الأول علشان نقدر نستخدمها، وكل كائن بيكون ليه نسخة خاصة بيها .

6:

```
using System;

class ComplexNumber
{
    public int Real { get; set; }
    public int Imaginary { get; set; }
```

```
    public ComplexNumber(int real, int imaginary)
    {
        Real = real;
        Imaginary = imaginary;
    }

    public static ComplexNumber operator *(ComplexNumber c1, ComplexNumber c2)
    {
        int real = (c1.Real * c2.Real) - (c1.Imaginary * c2.Imaginary);
        int imaginary = (c1.Real * c2.Imaginary) + (c1.Imaginary * c2.Real);
        return new ComplexNumber(real, imaginary);
    }

    public override string ToString()
    {
        return $"{Real} + {Imaginary}i";
    }
}

class Program
{
    static void Main()
```

```
    {
        ComplexNumber n1 = new ComplexNumber(2, 3);
        ComplexNumber n2 = new ComplexNumber(4, 5);
        ComplexNumber result = n1 * n2;
        Console.WriteLine(result);
    }
}
```

- ا، مش كل الـ operators ممكن نعملها overloading في.C#
- في Operators معينة زي +, -, *, /, ==, !=, <, > ممكن نعملها.
- لكن Operators تانية زي (dot) .أو (ternary) ?: أو (assignment) =مش مسموح.
- السبب: بعض الـ operators مرتبطة بلغة C# نفسها وبالـ CLR ومش منطقي يتغير سلوكها .

7:

```
using System;

class Program
{
    enum GenderDefault
    {
        Male,
        Female
    }
```

```
    enum GenderByte : byte
    {
        Male,
        Female
    }

    static void Main()
    {
        Console.WriteLine($"Size of GenderDefault (int): {sizeof(GenderDefault)} bytes");
        Console.WriteLine($"Size of GenderByte (byte): {sizeof(GenderByte)} bytes");
    }
}
```

- نفكر نغير النوع الأساسي للـ enum لما:
  1. **عايزين نوفر في الذاكرة**، خصوصًا لو عندنا ملايين العناصر.
  2. **القيم صغيرة ومعروفة**، ومش محتاجين مدى كبير زي `int`.
  3. في حالة التعامل مع **أنظمة قديمة أو APIs** بتتطلب حجم معين) زي byte أو short).

8:

```
using System;

static class Utility
{
    public static double CelsiusToFahrenheit(double celsius)
```

```
    {
        return (celsius * 9 / 5) + 32;
    }

    public static double FahrenheitToCelsius(double fahrenheit)
    {
        return (fahrenheit - 32) * 5 / 9;
    }
}

class Program
{
    static void Main()
    {
        double f = Utility.CelsiusToFahrenheit(25);
        double c = Utility.FahrenheitToCelsius(77);

        Console.WriteLine($"25°C = {f}°F");
        Console.WriteLine($"77°F = {c}°C");
    }
}
```

- الـ **static class** مش ممكن يتعمله `new`، لأنه معمول علشان يحتوي **أعضاء static فقط** (methods, fields, properties).

- وجود **constructor عادي (instance constructor)** معناه إننا نقدر نعمل object من الكلاس، وده عكس فكرة الـ static class.
- عشان كده الـ C# بتمنع أي instance constructor في static class وتسمح بس بالـ **static constructor**)اللي بيتنادى مرة واحدة أوتوماتيك عند أول استخدام.(

9:

```
using System;


class Program
{
   enum Grades
   {
      F = 1,
      D,
      C,
      B,
      A
   }


   static void Main()
   {
      Console.Write("Enter grade (A, B, C, D, F): ");
      string input = Console.ReadLine();
```

```
        if (Enum.TryParse(input, true, out Grades grade))
        {
            Console.WriteLine($"Valid grade: {grade} ({(int)grade})");
        }
        else
        {
            Console.WriteLine("Invalid grade entered.");
        }
    }
}
```

Enum.TryParse •بيديك طريقة آمنة لتحويل النصوص لـ enum من غير ما يحصل Exception لو القيمة غلط.

•مميزاته عن `int.Parse`أو `Enum.Parse`:

1. **ما بيرميش Exception** عند الخطأ ← أسرع وأكفأ.
2. بيرجع `bool`يوضح إذا التحويل نجح أو لأ.
3. بيسمح بخيارات زي تجاهل حالة الحروف.(case-insensitive)

•لكن `int.Parse`أو `Enum.Parse`هيرموا خطأ (Exception) لو الإدخال مش صحيح، وده محتاج.try/catch

10:

```
using System;

class Employee
{
```

```
    public int Id { get; set; }
    public string Name { get; set; }

    public override bool Equals(object obj)
    {
        if (obj is Employee other)
        {
            return this.Id == other.Id && this.Name == other.Name;
        }
        return false;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(Id, Name);
    }
}

class Helper2<T>
{
    public static int SearchArray(T[] array, T item)
    {
        for (int i = 0; i < array.Length; i++)
```

```
        {
            if (array[i].Equals(item))
            {
                return i;
            }
        }
        return -1;
    }
}

class Program
{
    static void Main()
    {
        Employee[] employees = new Employee[]
        {
            new Employee { Id = 1, Name = "Ali" },
            new Employee { Id = 2, Name = "Sara" },
            new Employee { Id = 3, Name = "Omar" }
        };

        Employee target = new Employee { Id = 2, Name = "Sara" };
```

```
        int index = Helper2<Employee>.SearchArray(employees, target);


        if (index != -1)
            Console.WriteLine($"Employee found at index {index}");
        else
            Console.WriteLine("Employee not found");
    }
}
```

- **Overriding Equals**:
  - بيحدد إزاي نقارن كائنين من نفس النوع بناءً على القيم.(Value equality)
  - افتراضيًا في الكلاسات، `Equals`بيعمل مقارنة بالـ Reference (هل نفس العنوان في الذاكرة). لما نعمل overrideنقدر نخليه يقارن بالمحتوى.

- **Overloading ==**:
  - لازم نعمله صراحة لو عاوزين نقارن الكائنات بالقيم.
  - افتراضيًا للكلاسات، ==بيقارن المرجع.(Reference equality)
  - للـstructs ، ==مش متعرف بشكل افتراضي (لازم نعرفه إحنا).

```
        11:
        using System;


        class Helper
        {
            public static T Max<T>(T a, T b) where T : IComparable<T>
            {
```

```
        return a.CompareTo(b) >= 0 ? a : b;
    }
}

class Program
{
    static void Main()
    {
        Console.WriteLine(Helper.Max(5, 10));
        Console.WriteLine(Helper.Max(3.14, 2.71));
        Console.WriteLine(Helper.Max("Ali", "Sara"));
    }
}
```

- 1. `where T : struct` → لازم يكون النوع **قيمة** **(Value Type)**.
  2. `where T : class` → لازم يكون النوع **مرجع** **(Reference Type)**.
  3. `where T : new()` → لازم يكون ليه **constructor افتراضي**.
  4. `where T : BaseClass` → لازم يكون وارث من كلاس معين.
  5. `where T : IInterface` → لازم يطبق إنترفيس معين.

12:

```
using System;

class Helper2<T>
{
```

```
    public static void ReplaceArray(T[] array, T oldValue, T newValue)
    {
        for (int i = 0; i < array.Length; i++)
        {
            if (array[i].Equals(oldValue))
            {
                array[i] = newValue;
            }
        }
    }
}

class Program
{
    static void Main()
    {
        int[] numbers = { 1, 2, 3, 2, 4 };
        Helper2<int>.ReplaceArray(numbers, 2, 9);
        Console.WriteLine(string.Join(", ", numbers));

        string[] names = { "Ali", "Sara", "Ali", "Omar" };
        Helper2<string>.ReplaceArray(names, "Ali", "Hana");
        Console.WriteLine(string.Join(", ", names));
```

```
    }
}
```

- **Generic Methods**:

- بتخلي الميثود نفسها عامة (generic) من غير ما الكلاس يكون.Generic
- النوع بيتحدد عند استدعاء الميثود.
- مرنة أكتر لو عندنا كلاس عادي لكن محتاجين ميثود تشتغل مع أنواع مختلفة.

- **Generic Classes**:

- الكلاس كله بيشتغل مع نوع محدد يتم تحديده وقت الإنشاء.
- كل الميثودز والخصائص جوا الكلاس بتشتغل على نفس النوع.
- مفيدة لما يكون عندنا بيانات أو عمليات مرتبطة بنوع واحد طول عمر الكائن (زي(<List<T

13:

```
using System;


struct Rectangle
{
   public int Length { get; set; }
   public int Width { get; set; }


   public Rectangle(int length, int width)
   {
      Length = length;
      Width = width;
   }
```

```
    public override string ToString()
    {
        return $"Length: {Length}, Width: {Width}";
    }
}

class Program
{
    static void Swap(ref Rectangle r1, ref Rectangle r2)
    {
        Rectangle temp = r1;
        r1 = r2;
        r2 = temp;
    }

    static void Main()
    {
        Rectangle rect1 = new Rectangle(5, 10);
        Rectangle rect2 = new Rectangle(7, 14);

        Console.WriteLine($"Before Swap: rect1 = {rect1}, rect2 = {rect2}");
```

```
        Swap(ref rect1, ref rect2);


        Console.WriteLine($"After Swap: rect1 = {rect1}, rect2 = {rect2}");
    }
}
```

- لو عملنا **custom Swap method** لكل نوع) زي (... Rectangle, Circle, Employee هنلاقي نفس الكود بيتكرر مع اختلاف النوع بس ← ده بيخلي الكود طويل وصعب الصيانة.
- باستخدام **generic swap method** نكتب الكود مرة واحدة ويشتغل مع أي نوع.
- ده بيحقق:
  1. **إعادة الاستخدام** .(Code Reusability)
  2. **تقليل الأخطاء** لأن الكود الموحد أسهل في الاختبار والصيانة.
  3. **مرونة** بحيث أي نوع جديد يقدر يستفيد من نفس الميثود .

14:

```
using System;


class Department
{
    public int DeptId { get; set; }
    public string DeptName { get; set; }


    public override bool Equals(object obj)
    {
        if (obj is Department other)
        {
```

```
            return this.DeptId == other.DeptId && this.DeptName == other.DeptName;
        }
        return false;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(DeptId, DeptName);
    }

    public override string ToString()
    {
        return $"{DeptName} ({DeptId})";
    }
}

class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Department Department { get; set; }

    public override string ToString()
```

```
    {
        return $"{Name} - {Department}";
    }
}

class Helper2<T>
{
    public static int SearchArray(T[] array, T item)
    {
        for (int i = 0; i < array.Length; i++)
        {
            if (array[i].Equals(item))
                return i;
        }
        return -1;
    }
}

class Program
{
    static void Main()
    {
        Department it = new Department { DeptId = 1, DeptName = "IT" };
```

```
        Department hr = new Department { DeptId = 2, DeptName = "HR" };


        Employee[] employees = new Employee[]
        {
            new Employee { Id = 1, Name = "Ali", Department = it },
            new Employee { Id = 2, Name = "Sara", Department = hr },
            new Employee { Id = 3, Name = "Omar", Department = it }
        };


        int index = Helper2<Department>.SearchArray(
            new Department[] { employees[0].Department, employees[1].Department,
employees[2].Department },
            new Department { DeptId = 1, DeptName = "IT" }
        );


        if (index != -1)
            Console.WriteLine($"Department found at index {index}");
        else
            Console.WriteLine("Department not found");
    }
}
```

- لو ما عملناش **override** لـ **Equals** في `Department`، المقارنة هتكون بالـ reference (العنوان في الذاكرة).
- ده معناه إن حتى لو عندنا اتنين `Department` بنفس `DeptId` و`DeptName`، مش هيعتبَروا متساويين لو اتعملوا بإنشاء منفصل.

- لما نعمل **Equals-override** ، المقارنة هتبقى بالقيم(value equality) ، وده بيخلي البحث أوضح وأدق، خصوصًا لما نتعامل مع بيانات جاية من مصادر مختلفة.(DB, API...)

15:

```
using System;


struct CircleStruct
{
    public int Radius { get; set; }
    public string Color { get; set; }


    public CircleStruct(int radius, string color)
    {
        Radius = radius;
        Color = color;
    }


    public override bool Equals(object obj)
    {
        if (obj is CircleStruct other)
        {
            return this.Radius == other.Radius && this.Color == other.Color;
        }
```

```
        return false;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(Radius, Color);
    }
}

class CircleClass
{
    public int Radius { get; set; }
    public string Color { get; set; }

    public CircleClass(int radius, string color)
    {
        Radius = radius;
        Color = color;
    }

    public override bool Equals(object obj)
    {
        if (obj is CircleClass other)
```

```
        {
            return this.Radius == other.Radius && this.Color == other.Color;
        }
        return false;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(Radius, Color);
    }
}

class Program
{
    static void Main()
    {
        CircleStruct s1 = new CircleStruct(5, "Red");
        CircleStruct s2 = new CircleStruct(5, "Red");

        Console.WriteLine($"Struct Equals: {s1.Equals(s2)}");
        // Console.WriteLine(s1 == s2); // مش متاح افتراضيًا للـ struct

        CircleClass c1 = new CircleClass(5, "Red");
```

```
        CircleClass c2 = new CircleClass(5, "Red");


        Console.WriteLine($"Class Equals: {c1.Equals(c2)}");

        Console.WriteLine($"Class == : {c1 == c2}");

    }

}
```

- في **structs**، الـ ==مش متعرف بشكل افتراضي لأن:
  1. الـ struct ممكن يحتوي حقول كتيرة جدًا، والمقارنة bit by bit مش منطقية دائمًا.
  2. مش كل structs محتاجة مقارنة بالقيمة.(Value Equality)
  3. C#بتسيب القرار للمطور لو عايز يعرّف ==بشكل يناسب struct بتاعه.
- أما **Equals**فهو متاح وبيتم عمل override له علشان نقارن القيم.

في الكلاسات:

- ==بيقارن المرجع (Reference equality) إلا لو عملنا له.override
- `Equals`ممكن نعيد تعريفه علشان يقارن بالمحتوى.

Wafaa Mohammed - أنت
الآن ·

عمرك سألت نفسك هو ليه في البرمجة عندنا أنواع كتير من الكلاسات (Classes) ومش مجرد نوع واحد وخلاص؟ 🤔
السر هنا إن كل نوع من الكلاسات معمول عشان يخدم سيناريو مختلف ويسهل على المبرمج شغله. تعالا نوضحهم بشكل بسيط:

🧑‍🏫 أنواع الكلاسات:

1️⃣ Concrete Class
ده الكلاس العادي اللي بننشن منه Objects ونشتغل بيه على طول.
مثال:

```
class Car
{
  public string Model { get; set; }
}
```

2️⃣ Abstract Class
زي كأنك عامل خطة عامة، فيها دوال مش متطبقة بالكامل، وبتجبر أي كلاس يرثها إنه يكمل الباقي.

```
abstract class Shape
{
  public abstract double Area();
}
```

3️⃣ Sealed Class
كلاس متقفول 🔒، يعني مينفعش أي كلاس يرثه.

```
sealed class Logger { }
```

4️⃣ Static Class
الكلاس ده مينفعش نعمل منه Object، كله static methods، زي الأدوات الجاهزة اللي تستدعيها على طول.

```
static class MathHelper
{
  public static int Add(int a, int b) => a + b;
}
```

5️⃣ Partial Class
لما الكود بتاعك يطول، تقدر تقسم تعريف الكلاس على كذا ملف عشان يبقى أسهل في التنظيم.

6️⃣ Nested Class
كلاس جواه كلاس تاني، وده بيكون مفيد لو الكلاس التاني ليه علاقة مباشرة بالأول.

📌 كل نوع من الكلاسات معمول لغرض، والمبرمج الشاطر هو اللي يعرف يستخدم الأنسب في الوقت الصح.

Generalization Concept using Generics

إحنا في البرمجة ساعات بنعمل دوال أو كلاسات بتكرر نفس الفكرة بس بتشتغل على أنواع مختلفة (int, string, double… إلخ.)

من غير **Generics** كنا هنضطر نكتب نفس الكود كذا مرة لكل نوع.
لكن بالـ **Generics** نقدر نكتب الكود مرة واحدة بشكل **عام (Generalized)** ويشتغل مع أي نوع بيانات.

مثال من غير Generics:

```
public class IntRepository
{
public int Value { get; set; }
}

public class StringRepository
{
public string Value { get; set; }
}
```

هنا عندنا كلاس للـ int وكلاس تاني للـ string → تكرار !

نفس المثال باستخدام Generics (Generalization):

```
public class Repository<T>
{
public T Value { get; set; }
}
```

دلوقتي الكلاس ده ممكن يستخدم مع أي نوع:

```
var intRepo = new Repository<int> { Value = 10 };
var stringRepo = new Repository<string> { Value = "Hello" };
```

- **Generalization** = نخلي الكود عام يشتغل على أكتر من نوع.
- **Generics** = الأداة اللي بتخلينا نحقق ده في C#.

What we mean by *Hierarchy Design* in real business?

Hierarchy designيعني ببساطة "التصميم الهرمي" أو "الهيكلية" جوّه أي بيزنس أو سيستم.
الفكرة إنك بترتب الكيانات (Entities) أو الناس أو الأقسام في شكل هرمي فيه مستويات(Levels) ، وكل مستوى ليه علاقة بالمستوى اللي فوقه أو تحته.

في البيزنس الحقيقي:

- عندك **CEO**فوق الهرم.
- تحته .**Managers**
- تحتهم .**Team Leads**
- تحتهم .**Employees**

ده اسمه **Organizational Hierarchy**، بيوضح مين بيراقب مين ومين مسؤول عن إيه.

في تصميم السوفتوير:(Business Systems)

إنت بتعكس نفس الفكرة دي في **الـ Database أو الـ.Classes**
مثلاً:

- كلاس **Employee**
- كلاس **Manager**يرث من) Employee بس ليه صلاحيات أكتر.(
- كلاس **Director**يرث من.Manager

يبقى عندك **hierarchy of classes**زي بالظبط الهيكل الإداري في الشركة.

الهدف من الـ:Hierarchy Design

1. تنظيم العلاقة بين الكيانات.
2. تسهيل الـ) **Authorization**مين ليه صلاحيات يعمل إيه.(
3. إن السيستم يعكس الواقع بتاع الشركة أو البيزنس.

🔷يعني لما نقول → **Hierarchy Design in real business**نقصد إننا بنعمل تصميم يعكس الواقع الإداري أو الوظيفي في شكل هرمي، سواء كان **Organizational Structure**أو **Inheritance**في الـ.**OOP**
