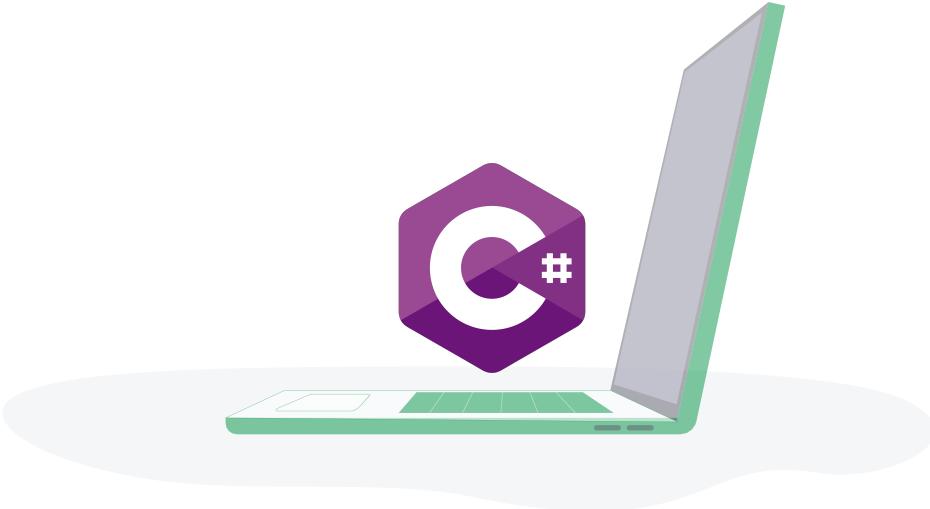


C# Object Oriented programming (OOP)

Education and Training Solutions 2022



- 1 Overview of OOP
- 2 OOP Structure
- 3 Encapsulation
- 4 Constructor
- 5 Constructor Overloading

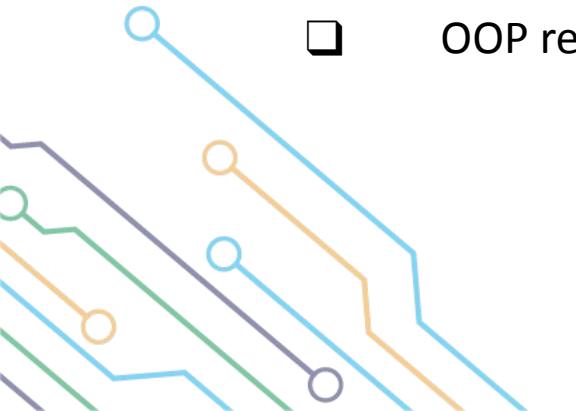






Overview of OOP

- The process of organizing software design around data or objects, rather than functions and logic is called Object Oriented Programming.
- Large, complex, and actively updated programs are a good fit for this method of development.
- OOP revolves around the **real-life** entities.



OOP programming languages

- C#
- Python
- C++
- PHP
- TypeScript



OOP benefits

- **Reusability:** means that you don't have to write code multiple times, Instead,
You can reuse it through inheritance.
- **Modularity:** it is about partitioning the code into modules, building them first
followed by linking and finally combining them to form a complete project.

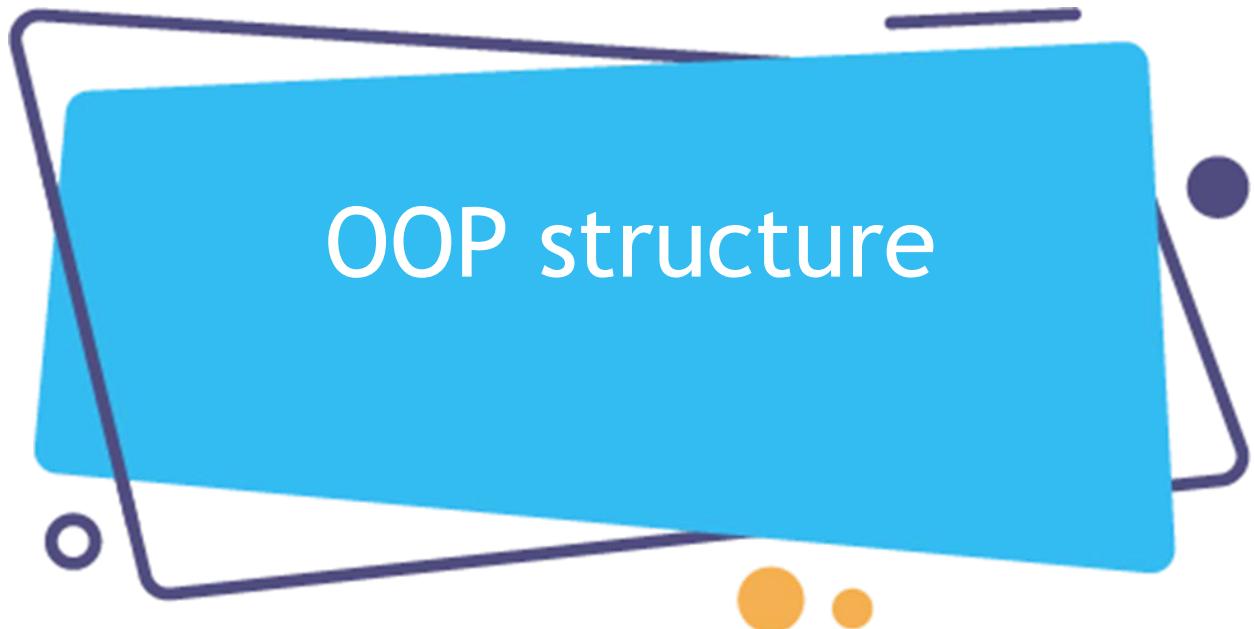




OOP benefits

- **Security:** as the complex code is hidden, software maintenance is easier and internet protocols are protected.
- **Productivity:** construct new programs quicker using multiple libraries and reusable code.

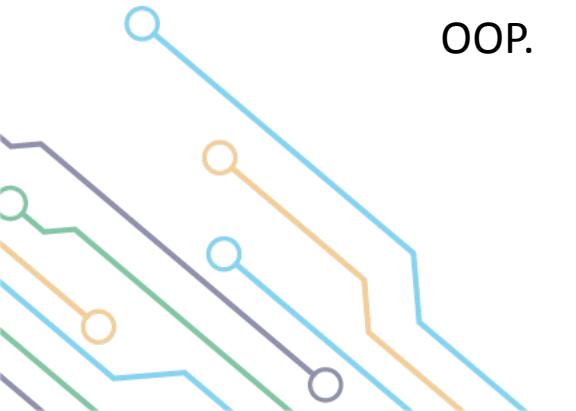






Main aspects of OOP

- **Class:** user-defined datatype act as the **blueprint** for the object, that contains functions to perform operations on the data and variables for storing data.
- **Object:** it is an instance of a class that specifies real-world data.
- The classes, objects, and their **attributes** and **methods** is the main component of OOP.





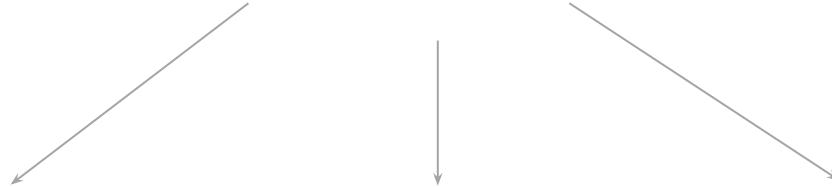
Class and object

In real life, a car is an object. The car has **attributes or field**, such as weight, size and color, and **methods**, such as drive and brake.



Objects

Class

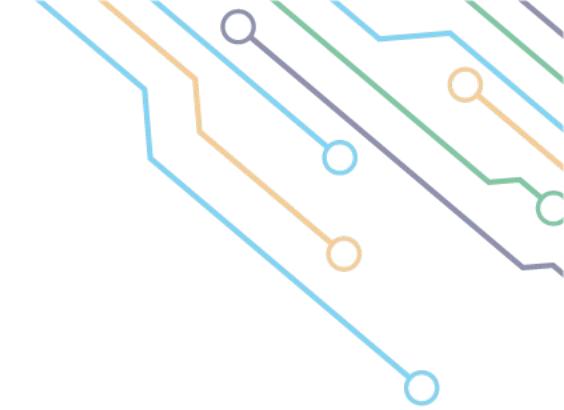




Class and object

- To create a class, we should use the **class** keyword while if we want to create an object of a class then we should use **new** keyword.
- You can access the class members once you create the object using this object name.



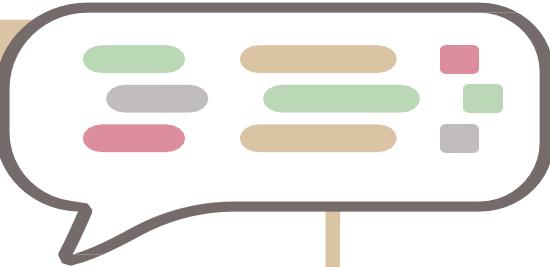


Let's Take an Example



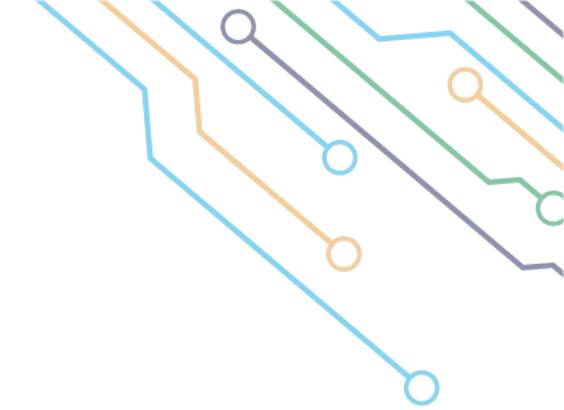
Define a class

```
public class Car
{
    string type;
    string weight;
    string color;
    public void Drive()
    { Console.WriteLine("Car is driving");
    }
    public void Brake()
    {
        Console.WriteLine("Car is
Breaking");
    }
}
// in main
Car car = new Car();
```



Console.WriteLine("Car is





Gift for you 😊

- When you want to name an object, variable or parameter use the **Camel case naming convention** (A word with the first letter lowercase, and the first letter of each subsequent word part capitalized. Ex: `customerName`, `employeeDetails`, `salary`).





Gift for you 😊

- As well as, if you want to name a class, method or property use **the Pascal case naming convention** (A word with the first letter capitalized, and the first letter of each subsequent word part capitalized. Ex: CustomerName, EmployeeDetails, Salary, etc.).





Fields

- It is a class variable; we can access it by creating an object from the class.
- It is private by default if you don't specify an access modifier.





Properties

- It is an extension of the class variable, it provides a mechanism to read, write or change the value of the class variable without affecting the external way of accessing it in our applications.





Properties

- In c#, properties can contain one or two code blocks called accessors get accessor and set accessor.
- By using get and set accessors, we can change the internal implementation of class variables and expose it based on our requirements.





Encapsulation

- One of the main concept of OOP, It is the process of making sure that the sensitive data are hidden from the users.
- This can be achieved by declaring variable as private, then use access modifier to read or write into it.



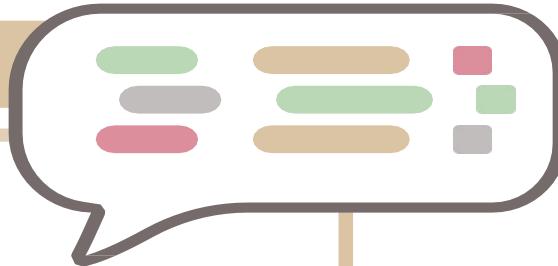


Let's Take an Example



This is how we can define a property

```
string type;
int weight;
string color;
public string Type
{ get=> type; set=> type=value; }
public int Weight
{ get=> weight; set=> weight=value; }
public string Color
{ get=> color; set=> color=value; }
```

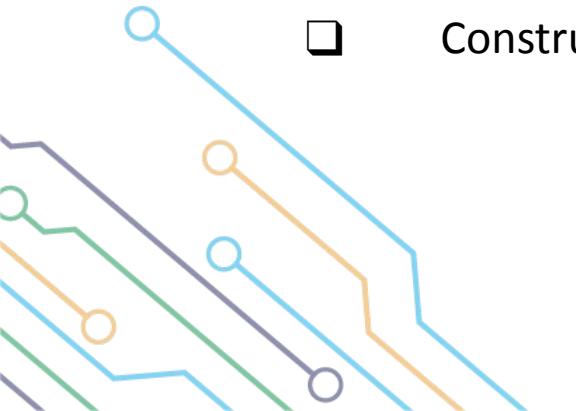


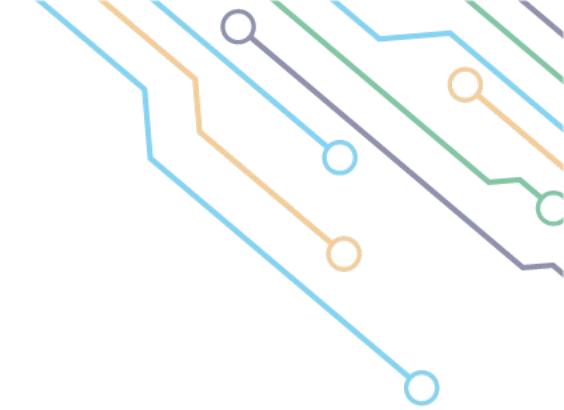




Instance constructor

- The instance constructor is special type of method that get executed when an instance of the class is created. Programmers usually use it to specify the fields that are declared in class for each object is created from. It is public by default and has no return type.
- Constructor usually used to Initialize the instance variables.



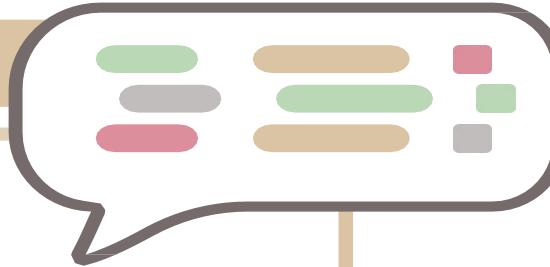


Let's Take an Example



Instance constructor

```
public Car(){  
    Console.WriteLine("This is  
constructor");  
}
```



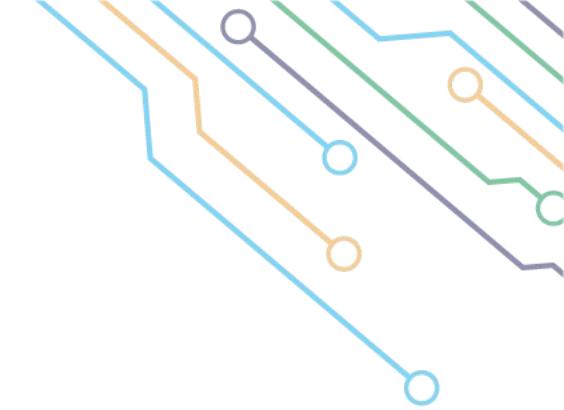
Constructor overloading



Constructor Overloading

- It is the ability to define a Constructor in different form.
- We can overload the constructor using same name with different parameters
(different type of parameters, different number of parameters or different order
of parameters).





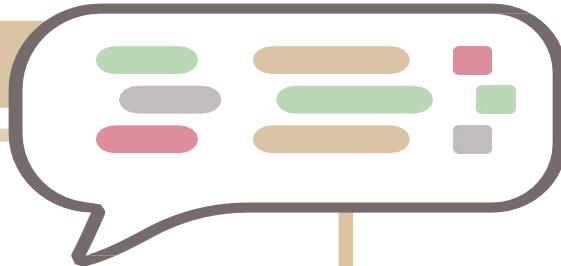
Let's Take an Example



Constructor overloading

```
public class Car {  
  
    public Car(){  
        constructor1");}  
  
    public Car(string color){  
        Console.WriteLine("This is constructor2");  
    }  
  
    public Car(string color, string type){  
        Console.WriteLine("This is constructor3");  
    }  
}
```

Console.WriteLine("This is







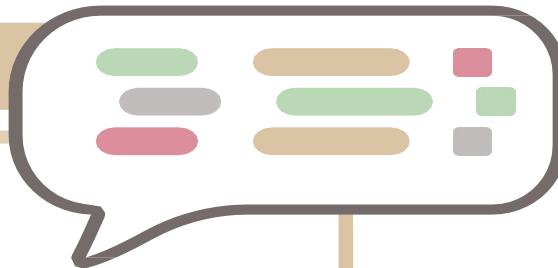
Example

Square and rectangle is an object, so we can represent “shape” as class of these object, it has height, width and color as **properties**, area and circumference as **methods**



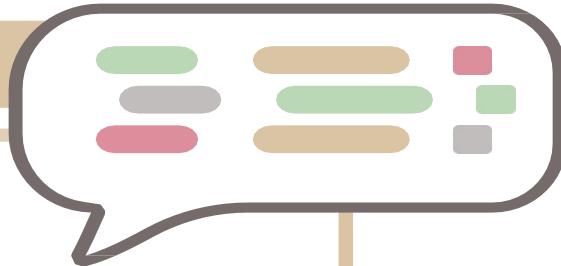
Define shape class

```
public class Shapes
{
    public Shapes(double shapeheight, double
shapewidth)
    {
        ShapeHeight = shapeheight;
        Shapewidth = shapewidth;
    }
    public double Shapewidth { get; set; }
    public double Shapeheight { get; set; }
    public int ShapeColor { get; set; }
    public double ShapeArea() {
        return Shapewidth * Shapeheight;
    }
}
```



Create object from shape class

```
static void Main(string[] args)
{
    Shapes Rectangle = new Shapes(5, 4);
    double area = Rectangle.ShapeArea();
    Console.WriteLine($"Rectangle area=
{area}");
}
```



1. Create a class called “Number” that have 2 int numbers as proprieties initially set to zero .
2. Create method called “calculate” that find the result of adding, subtracting, dividing and multiplying this 2 numbers.

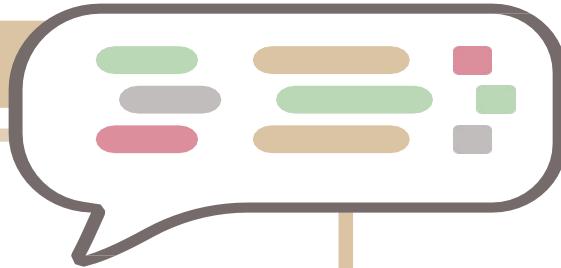


3. Create other method called “IsOdd” that return true if the 2 number is odd and false if one of them is even.



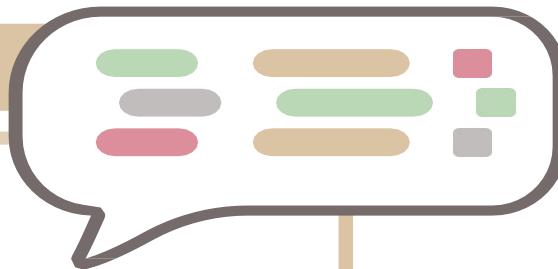
Exercise Solution 1

```
public class Number
{
    public int num1 { get; set; }
    public int num2 { get; set; }
    public Number(int num1, int num2)
    {
        this.num1 = num1;
        this.num2 = num2;
    }
}
```



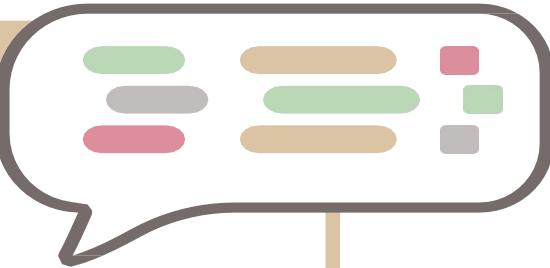
Exercise Solution 2

```
public void Calculate()
{
    Console.WriteLine($"num1 + num2 = {num1+
num2}");
    Console.WriteLine($"num1 * num2 = {num1*
num2}");
    Console.WriteLine($"num1 / num2 = {num1/ num2}");
    Console.WriteLine($"num1 - num2 = {num1-
num2}");
```



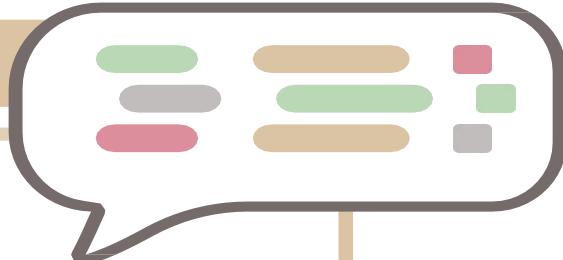
Exercise Solution 3

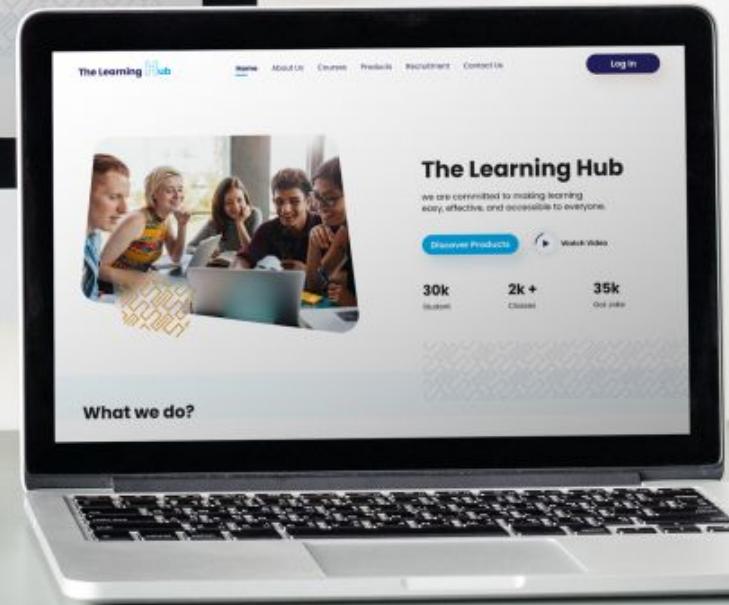
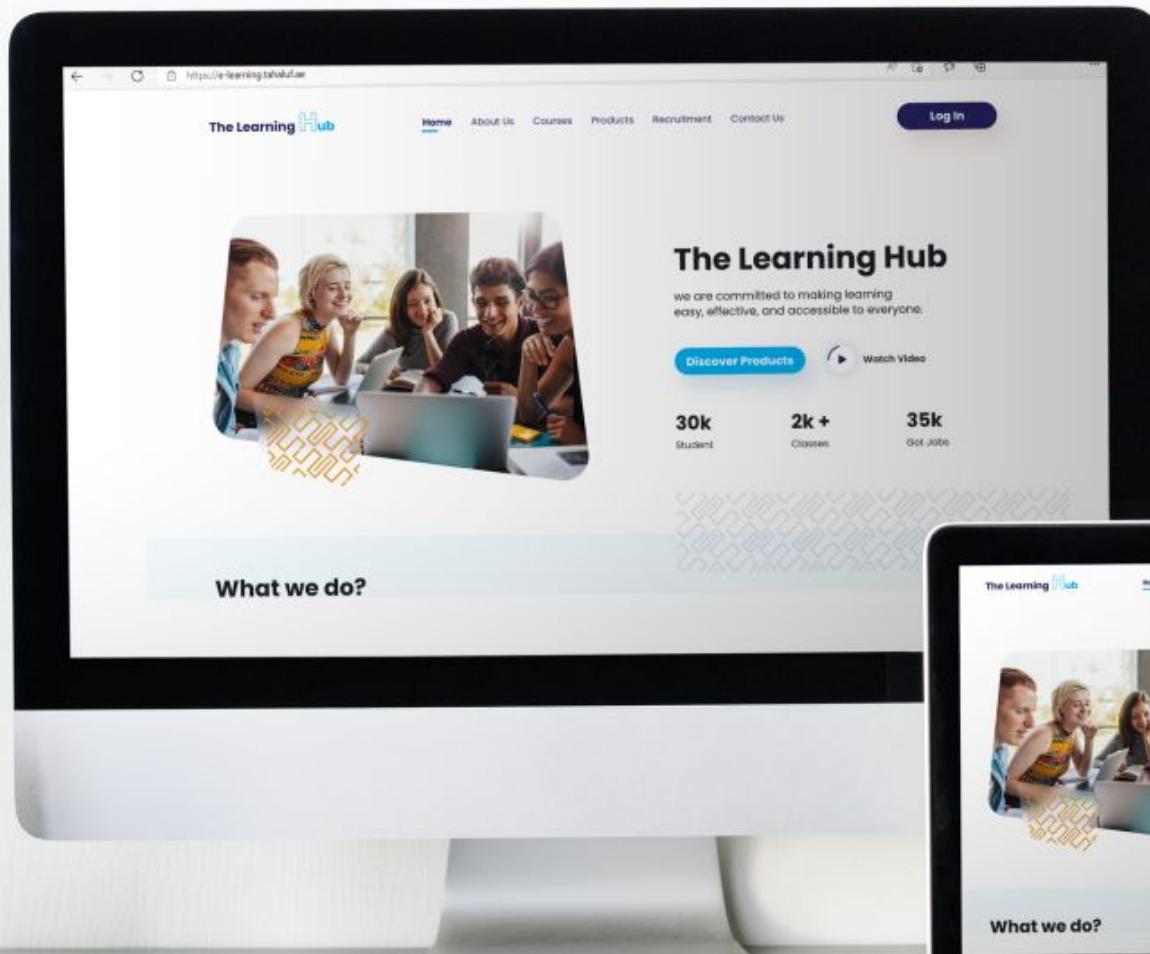
```
public bool IsOdd()
{
    if (num1 % 2 == 0)
    {
        return false;
    }
    else
        return true;
}
```



Exercise Solution 3 - main

```
static void Main(string[] args)
{
    Number number = new Number(2, 2);
    number.Calculate();
    if (number.IsOdd())
    {
        Console.WriteLine("The 2 numbers are
odd");
    }
    else {
        Console.WriteLine("One or both of
the 2 numbers are even");
    }
}
```





C# Object Oriented programming (OOP)

Education and Training Solutions 2022



- 1 Inheritance Overview
- 2 Single Inheritance
- 3 Multilevel Inheritance
- 4 Hierarchical Inheritance
- 5 Sealed Class
- 6 Access Modifiers





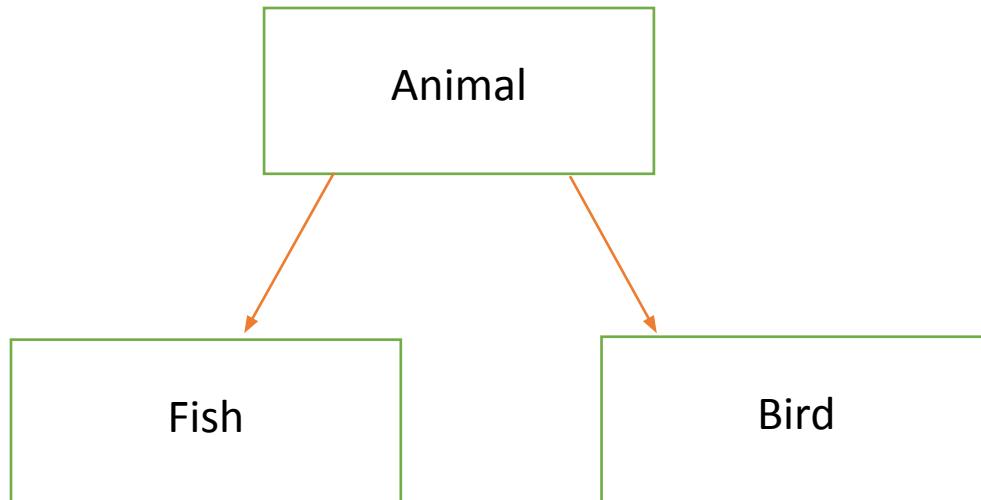


Inheritance Overview

One of the most primary characteristics in OOP is **Inheritance** which enables programmers not to repeat themselves while coding. As a result, using inheritance will reduce code redundancy which will reflect on whole program performance.



Inheritance Overview

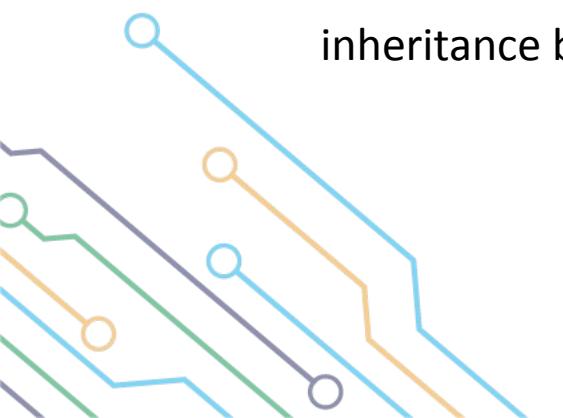




Inheritance Benefits

- Code reusability.
- Extend and modify the defined behavior.
- Create more specialized behavior.

Note that C# and .Net support single inheritance only and we can declare multiple inheritance by using interfaces as will be covered in the next chapter.

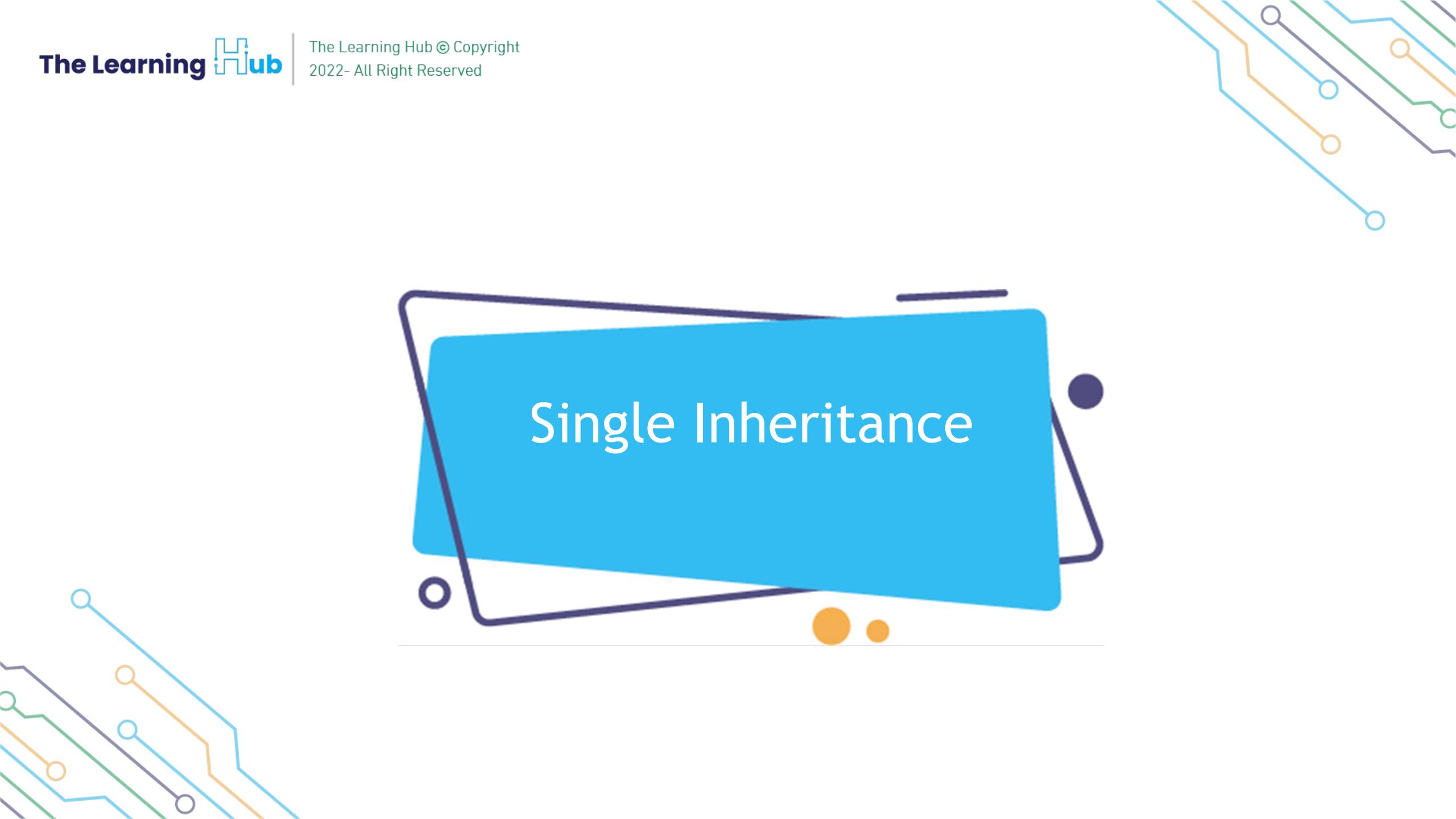




Points to Distinguish

- The more generalized class which its members are inherited based on access modifier is called **base** class or **parent**.
- The more specialized class which inherits these members is called **derived** class or **child**.

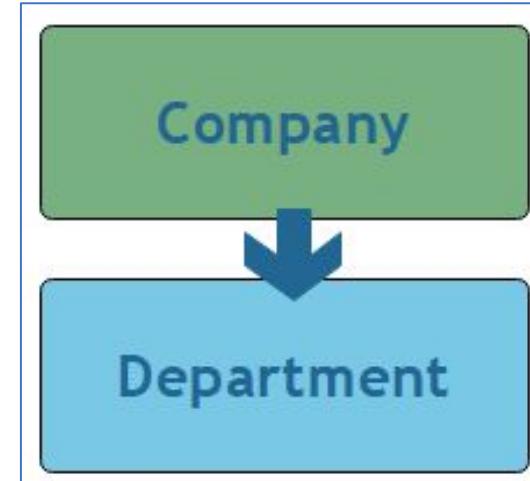




Single and Simple Inheritance

When there are a base class and one and only one derived class, this is what is meant by single inheritance.

Note that all members of base class will be inherited to derived class based on their access modifiers.





Let's Take an Example



Company class as base class

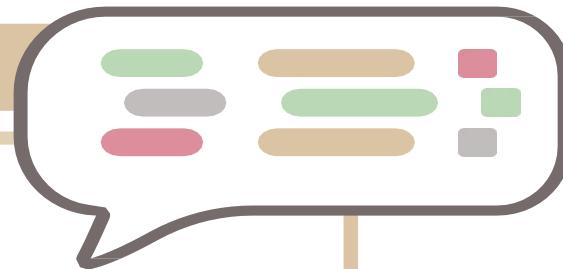
```
class Company
{
    public int Id { get; set; }
    public string Name { get ; set; }
    public string Location { get; set; }
    public string Description { get; set; }
    public void CompanyInfo() =>
        Console.WriteLine($"{Name} Company with {Id} id " +
            $"which is located at{Location}
    provide {Description}");
}
```



Department class as derived class

```
class Department:Company
{
    public int DepartmentId { get; set; }
    public string DepartmentName { get; set; }
    public int NumberOfEmployees { get; set; }
    public string ManagerName { get; set; }

    public void DepartmentInfo()=>
        Console.WriteLine($"{DepartmentName}
with {DepartmentId} id" +
                    $" under {ManagerName} management
has {NumberOfEmployees} employees");
}
```

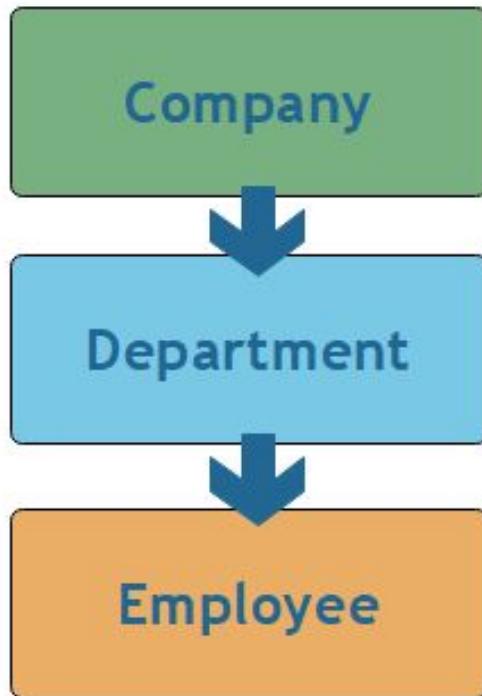




How can multilevel inheritance be achieved?

When a derived class inherits its members to another class, this is what is calling by multilevel inheritance.

- ❑ Here, the members of base class and the first derived class are inherited to the second derived class.



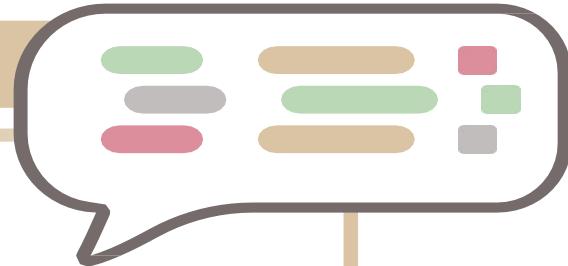


Let's Take an Example



Employee class inherits Department

```
class Employee:Department
{
    public int EmployeeId { get; set; }
    public string EmployeeName { get; set; }
    public double Salary { get; set; }
    public double Age { get; set; }
    public void EmployeeInfo()
    {
        Console.WriteLine($"{EmployeeName} with {EmployeeId}
id" +
                    $" is {Age} years old works at
{DepartmentName} in {Name} company" +
                    $" and receives {Salary} JOD ");
    }
}
```





Call base class constructor

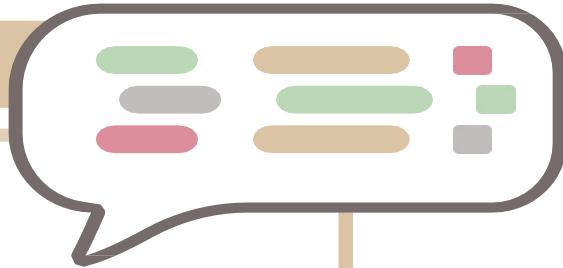
Programmers can call base class constructor using `base` keyword, which sometimes will be optional if the base class has overloading constructors, and sometimes will be mandatory when it has one constructor with parameters.

`Syntax public Derived(): base()`



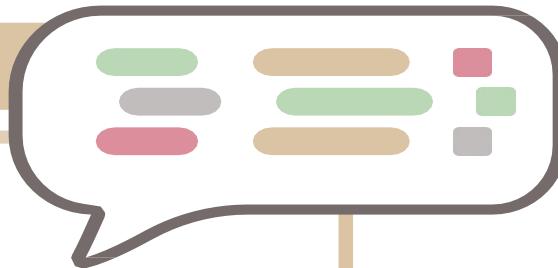
Call base class constructor

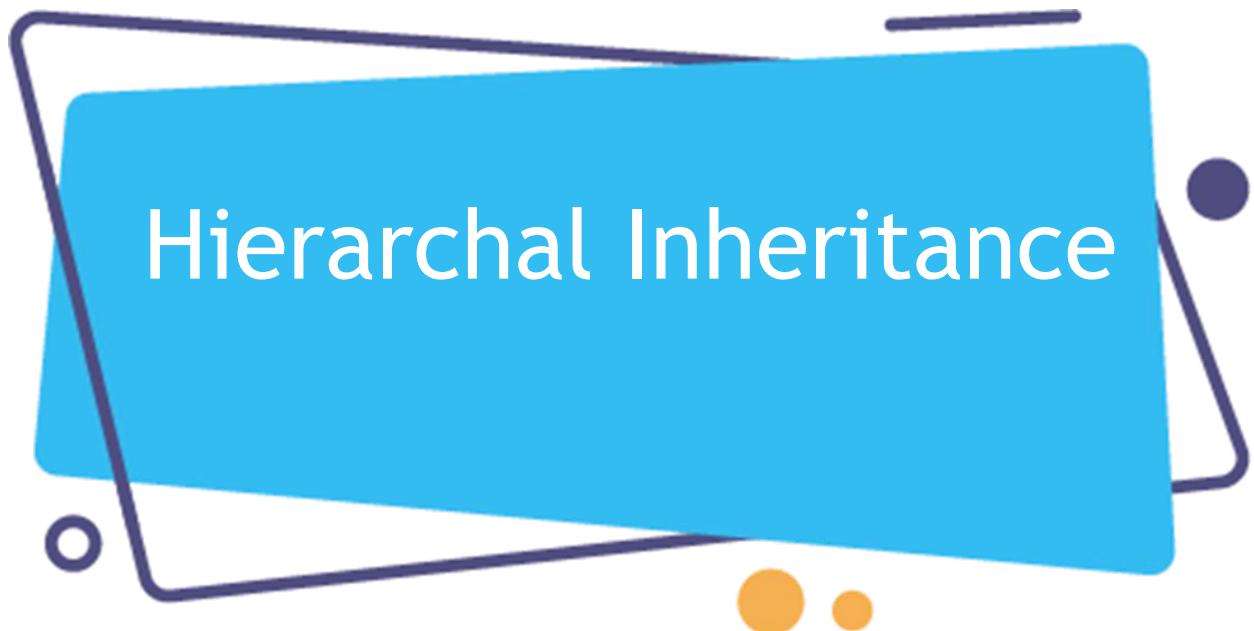
```
class Company
{
    public Company(int id)
    {
        Id= id;
    }
}
class Department: Company
{
    public Department(int companyId,int depId) :
base(companyId)
    {
        DepartmentId = depId;
    }
}
```



Continue..

```
class Employee:Department
{
    public Employee(int companyId,int depId,int
empId) : base(companyId, depId)
    {
        EmployeeId = empId;
    }
}
```

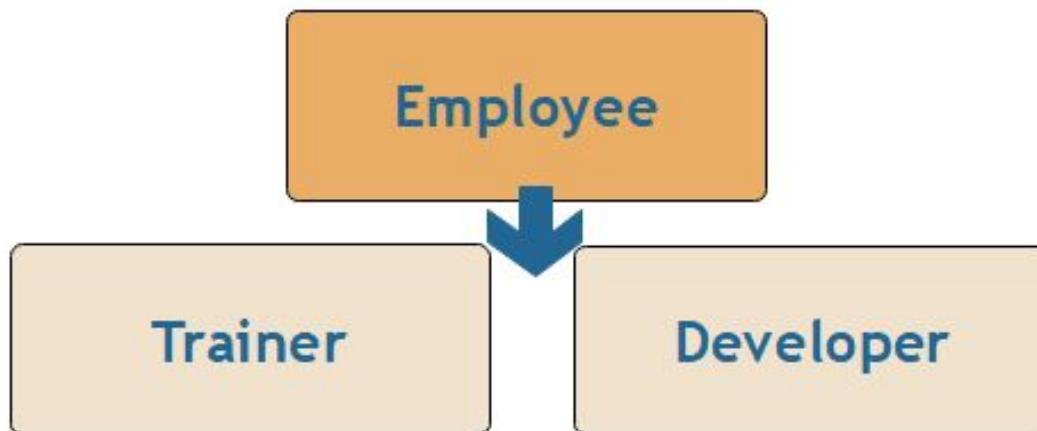


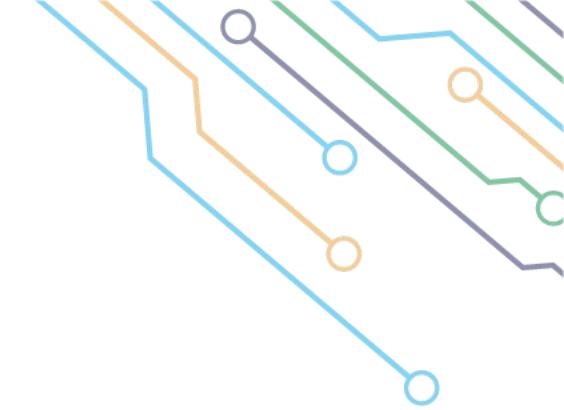


Hierarchical Inheritance

Hierarchical Inheritance

When some features or members are needed to be inherited to more than one derived class, programmers use hierarchical inheritance.





Let's Take an Example



Employee class as base class

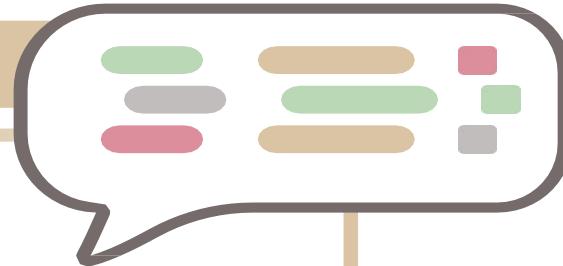
```
class Employee
{
    public int EmployeeId { get; set; }
    public string EmployeeName { get; set; }
    public double Salary { get; set; }
    public double Age { get; set; }
    public void EmployeeInfo()
    {
        Console.WriteLine($"{EmployeeName} with {EmployeeId}
id" +
                     $" is {Age} years old receives
{Salary} JOD");
    }
}
```

\$" is {Age} years old receives



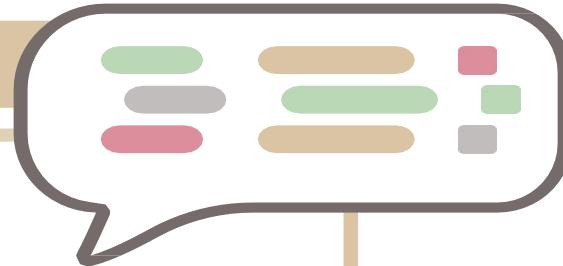
Trainer class inherits Employee class

```
class Trainer : Employee
{
    public string CourseName { get; set; }
    public int NumberOfStudents { get; set; }
    public DateTime StartDate { get; set; }
    public DateTime EndDate { get; set; }
    public void TrainerInfo(){
        Console.WriteLine($"{EmployeeName} presents
{CourseName}\n " +
                    $" to {NumberOfStudents} students
which starts at\n" +
                    $" {StartDate} and completes at
{EndDate}");
    }
}
```



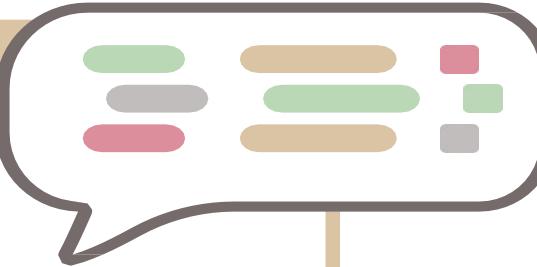
Developer class inherits Employee class

```
class Developer : Employee
{
    public string Specialty { get; set; }
    public string Position { get; set; }
    public void DeveloperInfo()
    {
        Console.WriteLine($"{EmployeeName} is
working in {Specialty} +
$" as {position}");
    }
}
```



Main

```
Trainer trainer = new Trainer();
trainer.CourseName = "OOP";
trainer.NumberOfStudents = 33;
DateTime date = new DateTime(2022, 10, 10);
TimeSpan duration = new TimeSpan(8,0,0,0);
trainer.StartDate = date;
trainer.EndDate = date.Add(duration);
trainer.TrainerInfo();
```







What if programmers want to prevent inheritance?

In some cases, programmers need to make some classes non-heritable, they can accomplish this by declaring classes as **sealed** ones.

Note that programmers still can declare an object from sealed class.



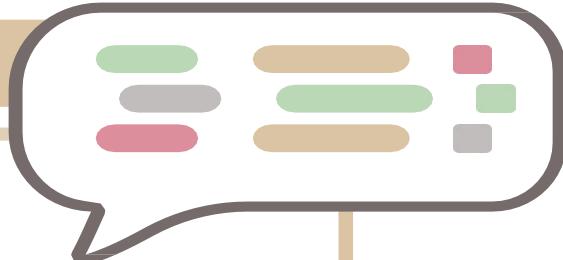


Let's Take an Example

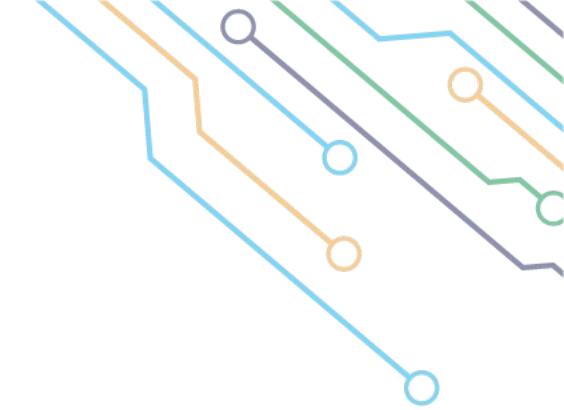


Sponsors class

```
sealed class Sponsors
{
    public string Id { get; set; }
    public string Name { get; set; }
    public void SponsorInfo()
    {
        Console.WriteLine($"{Name} with id {Id}");
    }
}
//in main
Sponsors sponsor = new Sponsors();
sponsor.Id = 1;
sponsor.Name = "Sir";
sponsor.SponsorInfo();
```







Overview of Access Modifier

By declaring access modifier, we determine the visibility and accessibility level of class and its members from outside the class in the same assembly or from another one references it, note that all types and members have an accessibility level in program.



Access modifiers

Public members can be accessed from whole program in the same assembly or another assembly that references it.

Private members can be accessed just by the owner class or struct itself.

Internal members can be accessed by the owner class surely and within current assembly in derived classes or as instances from.



Access modifiers

Protected members can be accessed by the owner class and from within any derived classes in the same assembly or in another assembly that references it.

Private Protected members can be accessed by the owner class and from within derived classes in the same assembly.



Access modifiers

Protected Internal members combine the accessibility level from both internal and protected access modifiers. Which mean they can be accessed by the owner class and within the same assembly in derived classes or as instances from. Additionally, they can be accessed from within derived classes in another assembly.



Inherited Members

What about access modifiers of members in base class? As it mentioned in the previous slide, derived class inherits the base class members based on these access modifiers.

- Now, let's add some members in our base class with all explained access modifiers and see the differences.



Summary table

Caller's location	public	protected internal	protected	internal	private protected	private
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

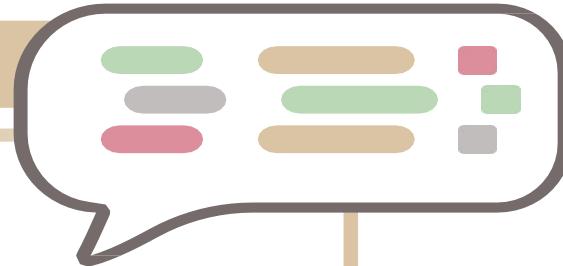


Let's make some changes



Add these properties in Company class

```
class Company
{
    .....
    private double SerialNumber { get; set; }
    internal int NumberOfEmployees{ get; set; }
    protected string Owner { get; set; }
    protected internal string Sponsor{ get; set; }
    private protected double Cost { get; set; }
}
```

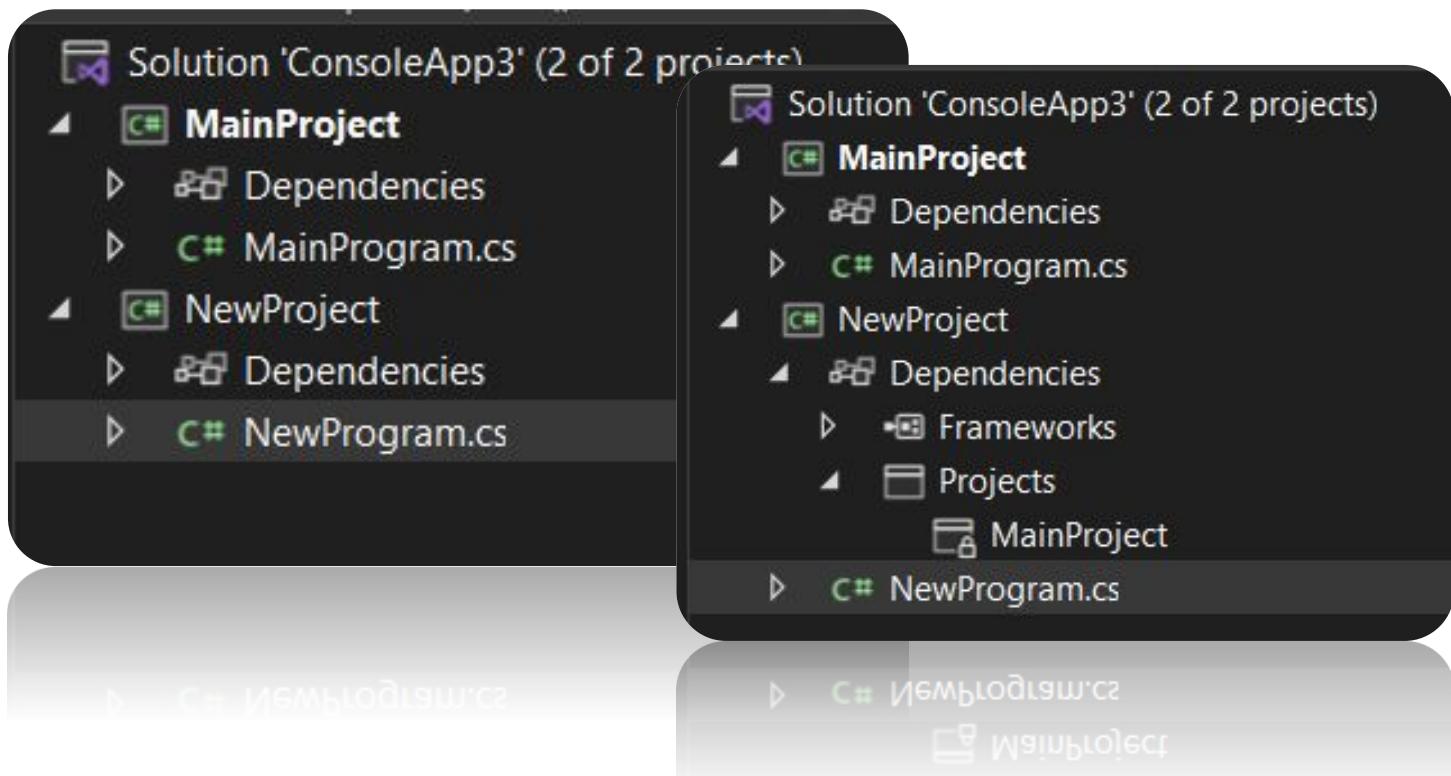


Add New Assembly

After creating a new assembly, make a reference between them and add derived class from Company class and see the inherited members based on access modifiers in these two situations:

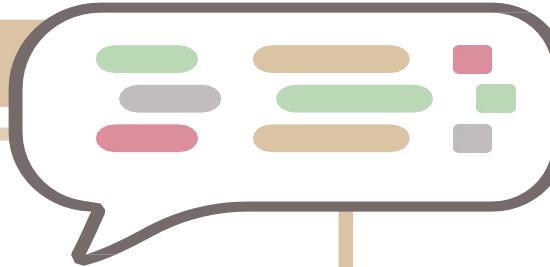
- In derived class.
- When an instance is created from company class.

Our Two Projects



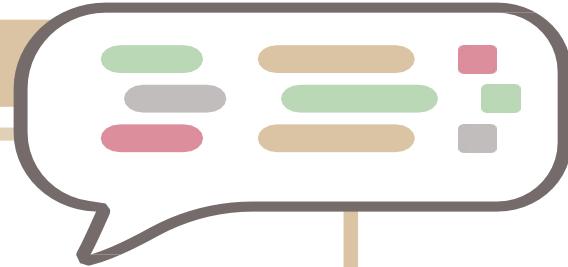
Add these properties in Company class

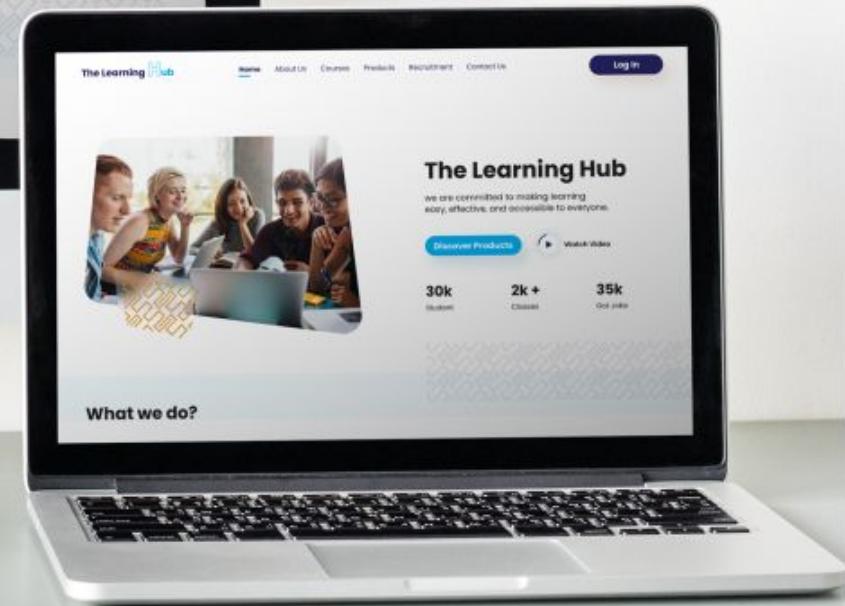
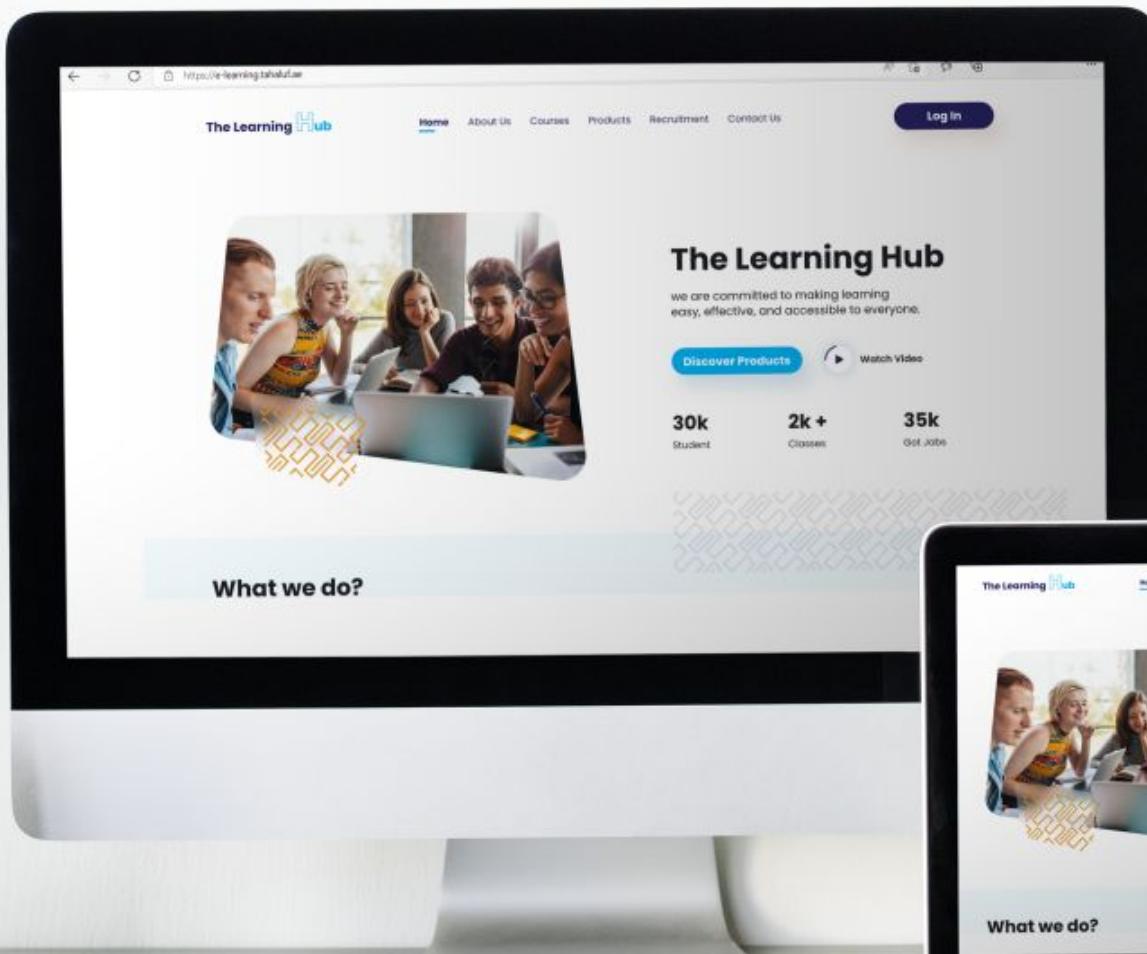
```
public class Company1:Company
{
    public Company1()
    {
        Id = 222;
        Name = "Harmony IT Solutions";
        Owner = "Sir";
        Sponsor = "Sir";
    }
}
```



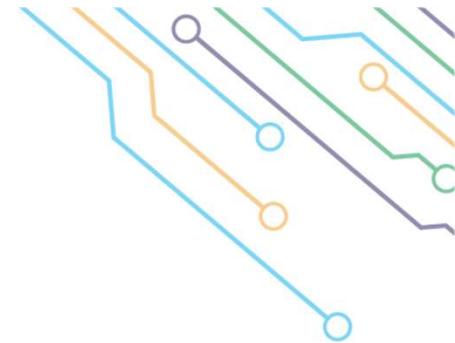
Continue..

```
public void PrintInheritedMembers()
{
    Console.WriteLine($"{base.Id}-{base.Name}: the owner
is {base.Owner} +
                    $"and it is sponsored by
{base.Sponsor}");
}
```



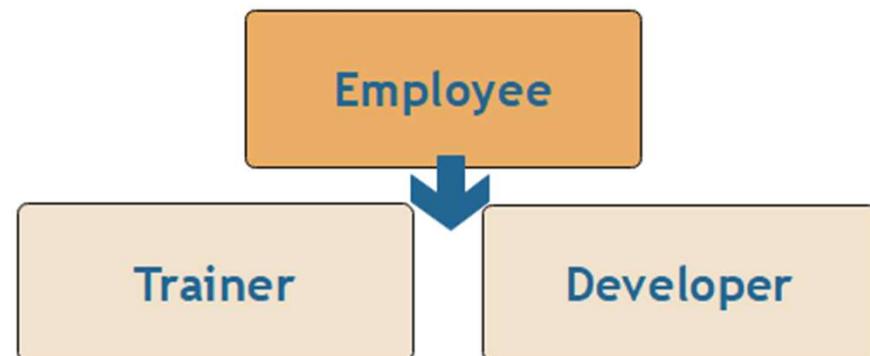


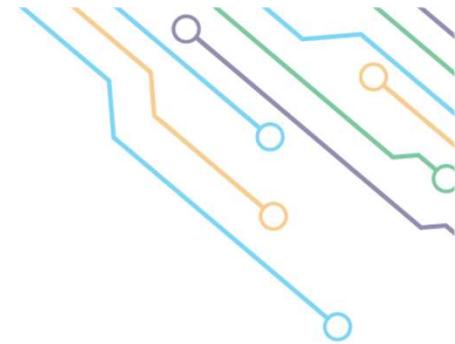
Hierarchal Inheritance



Hierarchical Inheritance

When some features or members are needed to be inherited to more than one derived class, programmers use hierarchical inheritance.





Let's Take an Example

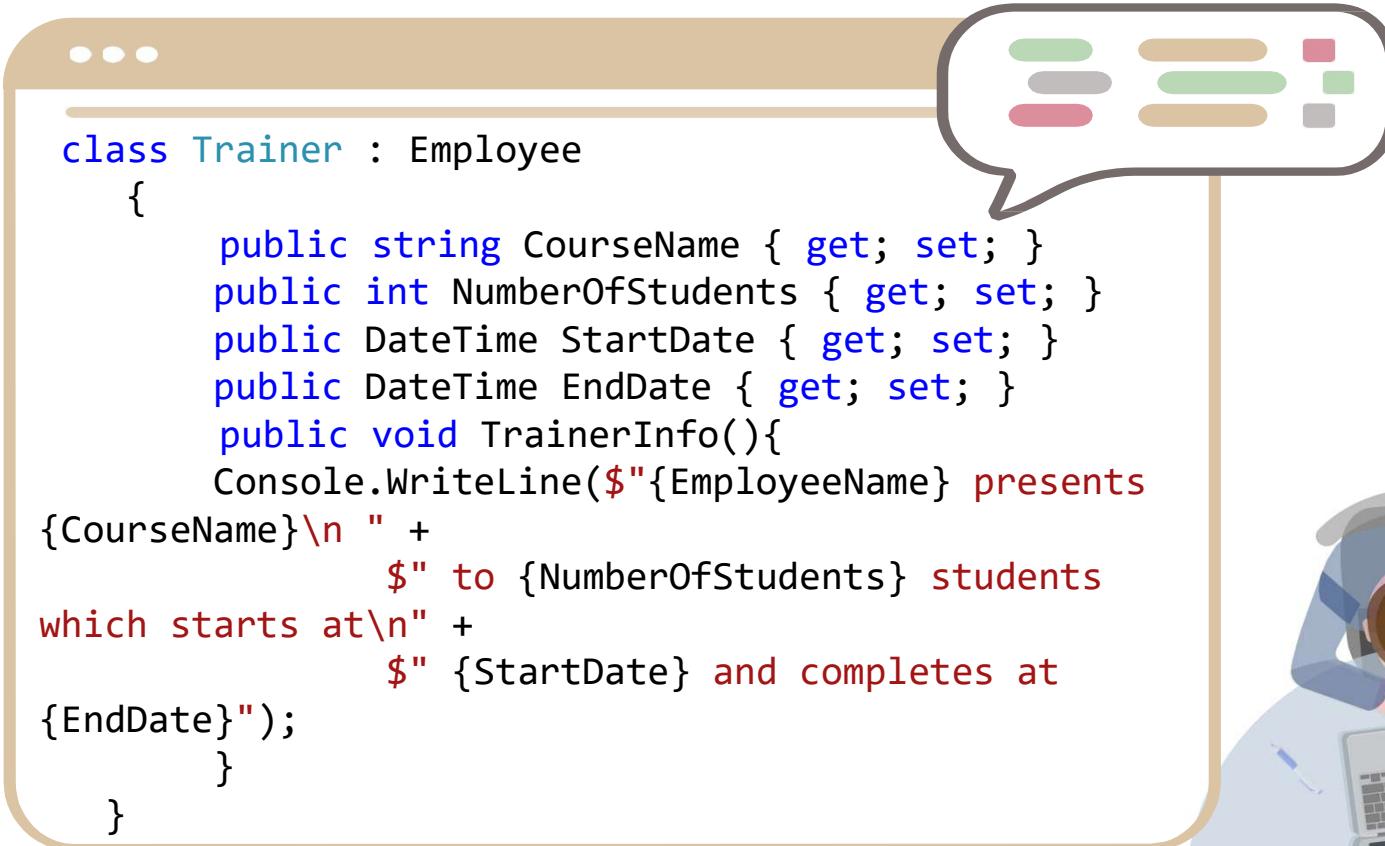


Employee class as base class

```
class Employee
{
    public int EmployeeId { get; set; }
    public string EmployeeName { get; set; }
    public double Salary { get; set; }
    public double Age { get; set; }
    public void EmployeeInfo()
    {
        Console.WriteLine($"{EmployeeName} with {EmployeeId}
id" +
                    $" is {Age} years old receives
{Salary} JOD");
    }
}
```



Trainer class inherits Employee class



```
class Trainer : Employee
{
    public string CourseName { get; set; }
    public int NumberOfStudents { get; set; }
    public DateTime StartDate { get; set; }
    public DateTime EndDate { get; set; }
    public void TrainerInfo(){
        Console.WriteLine($"{EmployeeName} presents
{CourseName}\n" +
                     $" to {NumberOfStudents} students
which starts at\n" +
                     $" {StartDate} and completes at
{EndDate}");
    }
}
```

A speech bubble contains the output of the TrainerInfo() method:

{EmployeeName} presents
{CourseName}
to {NumberOfStudents} students
which starts at
{StartDate} and completes at
{EndDate});



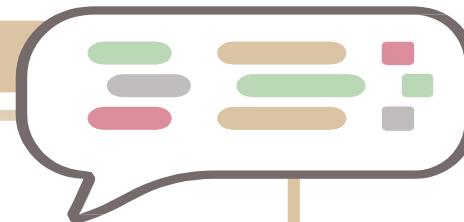
Developer class inherits Employee class

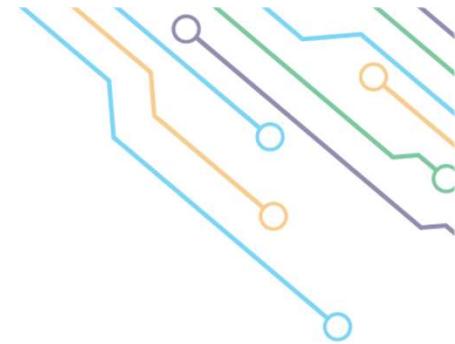
```
class Developer : Employee
{
    public string Specialty { get; set; }
    public string Position { get; set; }
    public void DeveloperInfo()
    {
        Console.WriteLine($"{EmployeeName} is
working in {Specialty} +
$" as {position}");
    }
}
```

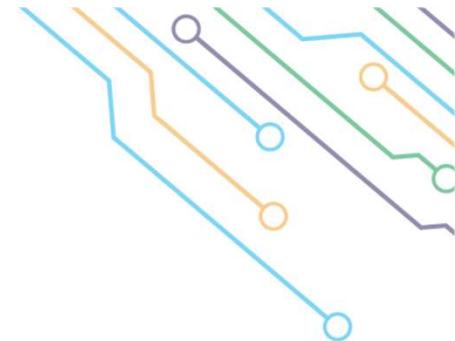


Main

```
Trainer trainer = new Trainer();
trainer.CourseName = "OOP";
trainer.NumberOfStudents = 33;
DateTime date = new DateTime(2022, 10, 10);
TimeSpan duration = new TimeSpan(8,0,0,0);
trainer.StartDate = date;
trainer.EndDate = date.Add(duration);
trainer.TrainerInfo();
```





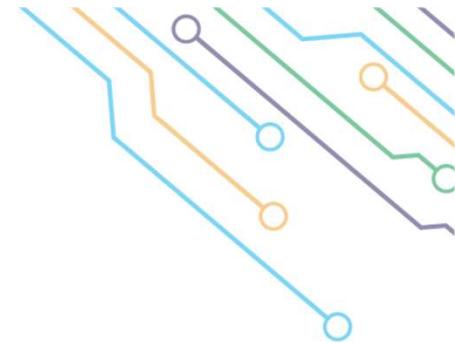


What if programmers want to prevent inheritance?

In some cases, programmers need to make some classes non-heritable, they can accomplish this by declaring classes as **sealed** ones.

Note that programmers still can declare an object from sealed class.



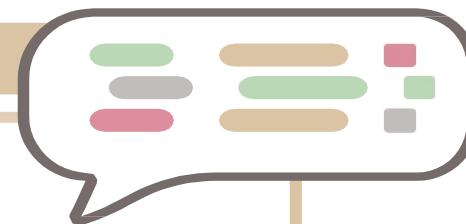


Let's Take an Example

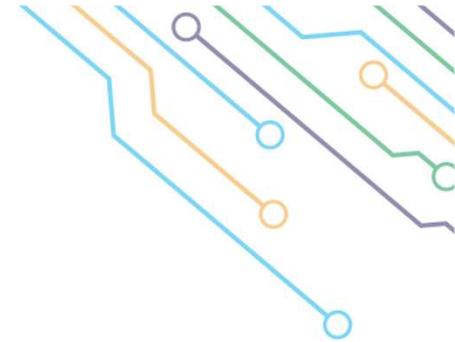


Sponsors class

```
sealed class Sponsors
{
    public string Id { get; set; }
    public string Name { get; set; }
    public void SponsorInfo()
    {
        Console.WriteLine($"{Name} with id {Id}");
    }
}
//in main
Sponsors sponsor = new Sponsors();
sponsor.Id = 1;
sponsor.Name = "Sir";
sponsor.SponsorInfo();
```



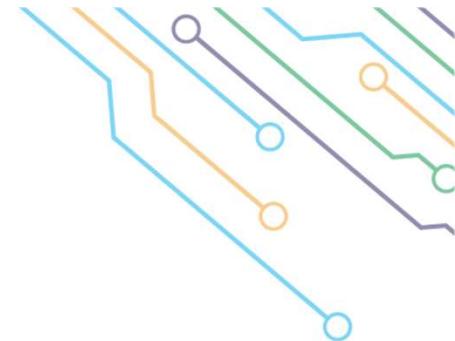




Overview of Access Modifier

By declaring access modifier, we determine the visibility and accessibility level of class and its members from outside the class in the same assembly or from another one references it, note that all types and members have an accessibility level in program.



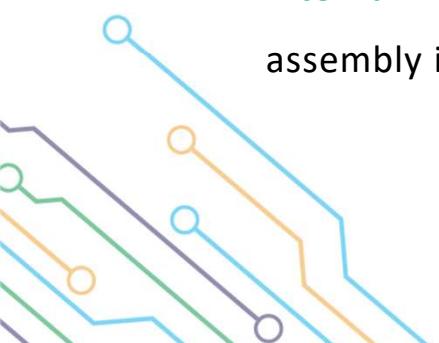


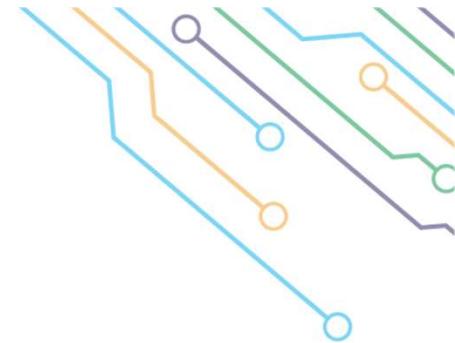
Access modifiers

Public members can be accessed from whole program in the same assembly or another assembly that references it.

Private members can be accessed just by the owner class or struct itself.

Internal members can be accessed by the owner class surely and within current assembly in derived classes or as instances from.



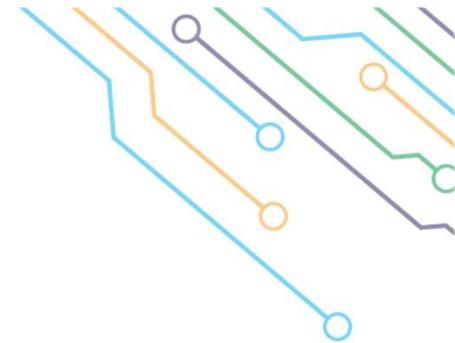


Access modifiers

Protected members can be accessed by the owner class and from within any derived classes in the same assembly or in another assembly that references it.

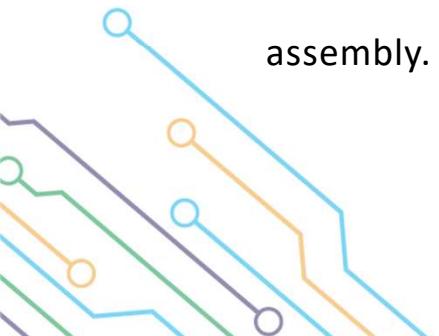
Private Protected members can be accessed by the owner class and from within derived classes in the same assembly.

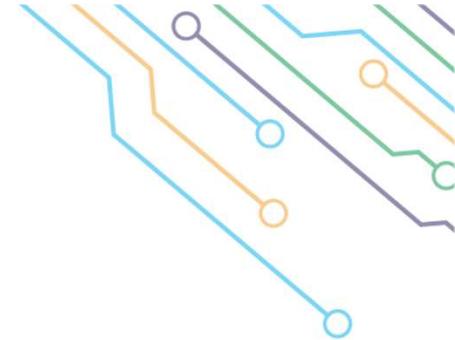




Access modifiers

Protected Internal members combine the accessibility level from both internal and protected access modifiers. Which mean they can be accessed by the owner class and within the same assembly in derived classes or as instances from. Additionally, they can be accessed from within derived classes in another assembly.

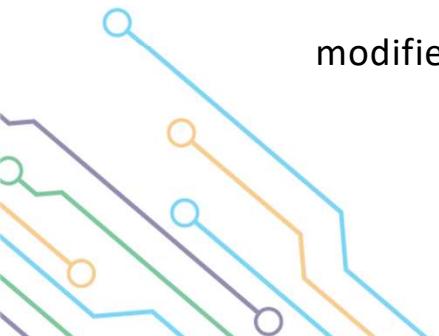




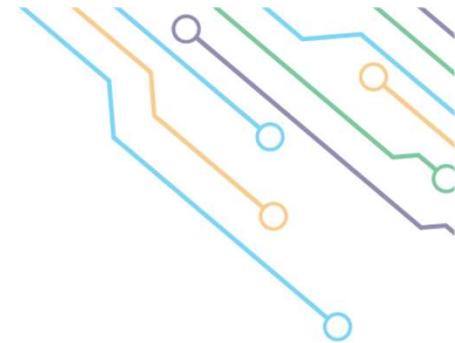
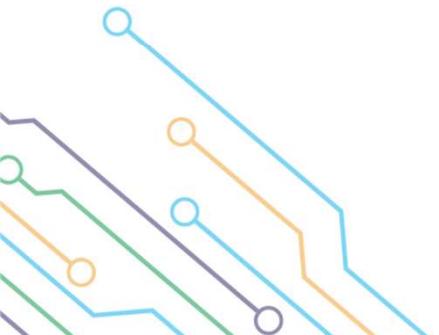
Inherited Members

What about access modifiers of members in base class? As it mentioned in the previous slide, derived class inherits the base class members based on these access modifiers.

- ❑ Now, let's add some members in our base class with all explained access modifiers and see the differences.

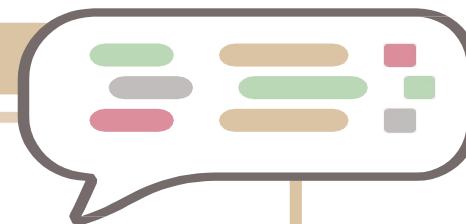


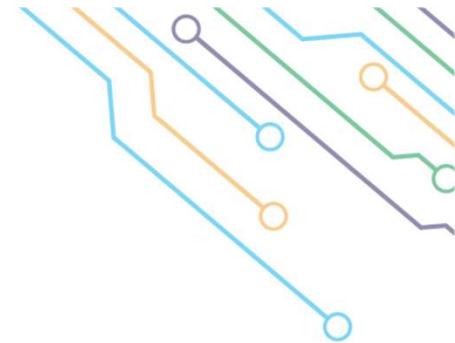
Let's make some changes 



Add these properties in Company class

```
class Company
{
    .....
    private double SerialNumber { get; set; }
    internal int NumberOfEmployees{ get; set; }
    protected string Owner { get; set; }
    protected internal string Sponsor{ get; set; }
    private protected double Cost { get; set; }
    .....
}
```



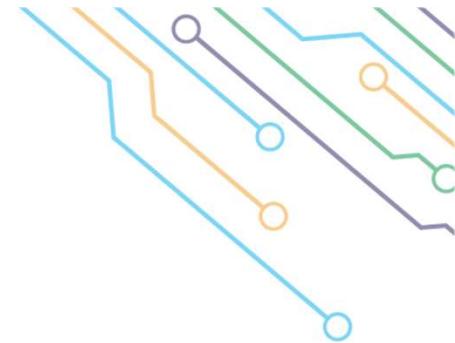


Add New Assembly

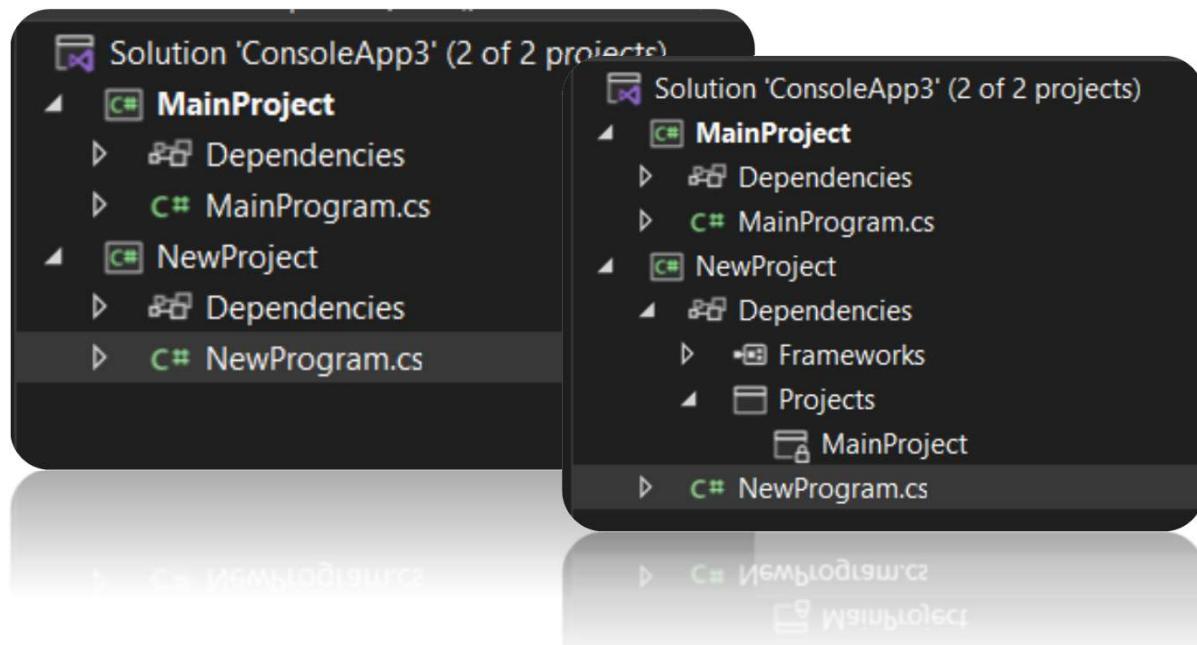
After creating a new assembly, make a reference between them and add derived class from Company class and see the inherited members based on access modifiers in these two situations:

- In derived class.
- When an instance is created from company class.



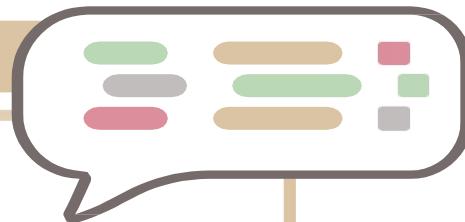


Our Two Projects



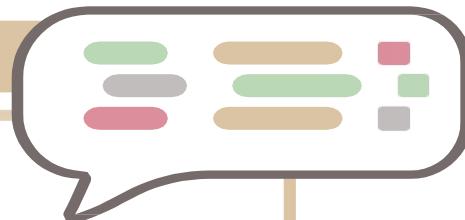
Add these properties in Company class

```
public class Company1:Company
{
    public Company1()
    {
        Id = 222;
        Name = "Harmony IT Solutions";
        Owner = "Sir";
        Sponsor = "Sir";
    }
}
```



Continue..

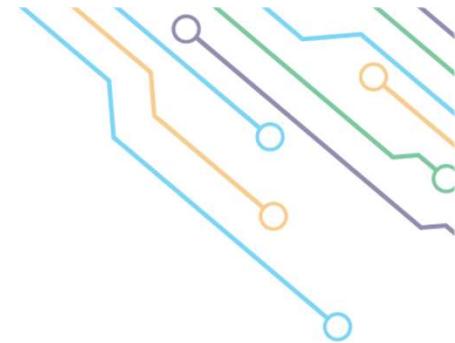
```
public void PrintInheritedMembers()
{
    Console.WriteLine($"{base.Id}-
{base.Name}: the owner is {base.Owner} " +
                    $"and it is sponsored by
{base.Sponsor}");
}
```



- 1 Polymorphism
- 2 Abstraction
- 3 Interface
- 4 Dependency Injection





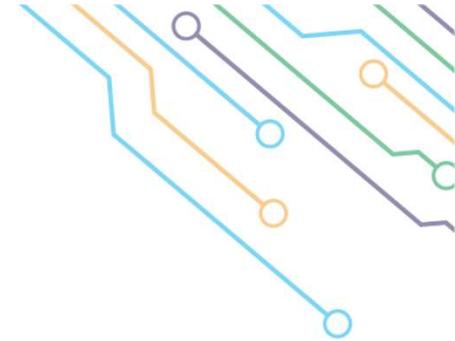


Polymorphism (Many shaped)

One of the main concept of OOP, it refers to the ability of providing more than one form either for a method or property, it has two types:

- Overloading.
- Overriding.

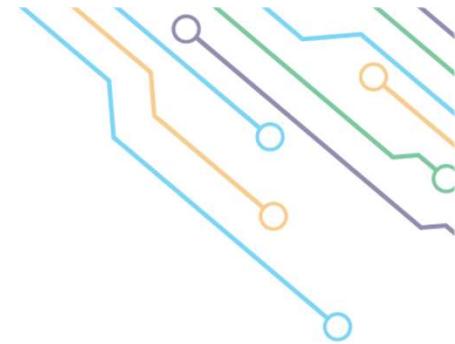




Overload

- Same method name with different parameter occurs at the same class.
- Different parameter means different type or number of parameters.
- The binding of overloading happens at compile time.



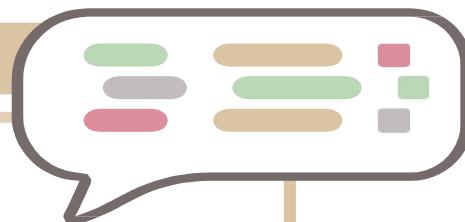


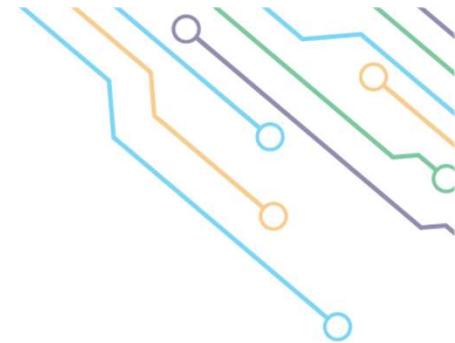
Let's Take an Example



Overloading Example

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public void GetName(string name)
    {
        Console.WriteLine("Name: "+ name);
    }
    public void GetName(int Id ,string name)
    {
        Console.WriteLine("Name: "+ name);
    }
}
```

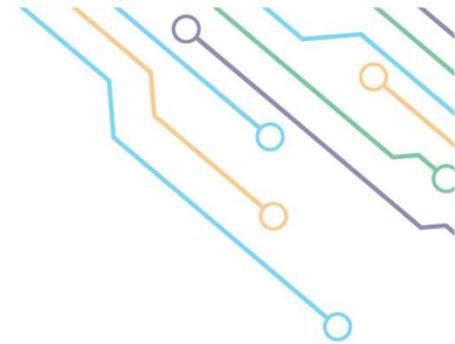




Override

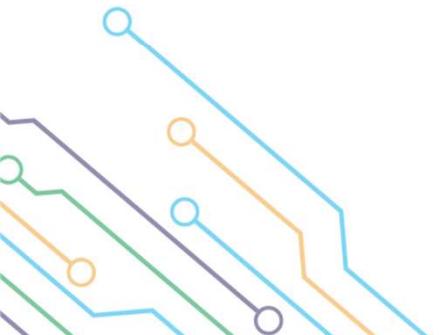
- Happens between base class and derived class.
- The implementation of the method which already implemented at the base class is also provided at the derived class.
- The binding of overriding happens at run time.

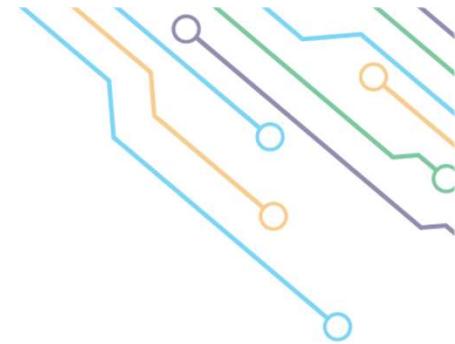




Override

- ❑ Use keyword “virtual” at the base class, “override” keyword at the derived class, as in the next slide...



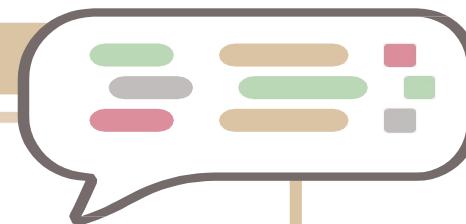


Let's Take an Example



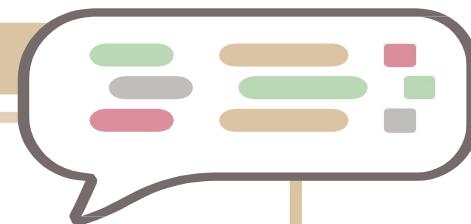
Overriding Example

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public virtual void GetName(string name)
    {
        Console.WriteLine("Name: " + name);
    }
}
```



Continue..

```
public class Student:Person
{
    public override void GetName(string name)
    {
        Console.WriteLine("Student Name: " +
name);
    }
}
```



Overriding Example (Main & result) (continued..):

A diagram illustrating a programming example. On the left, a large brown rounded rectangle contains a code editor window. Inside the code editor, the following C# code is shown:

```
Person student = new Student();
student.GetName("Amal");
```

Below the code editor is a terminal window titled "cmd.exe" with the path "C:\Windows\system32\cmd.exe". The terminal displays the output of the code execution:

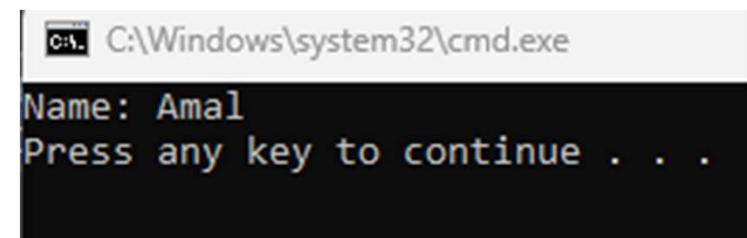
```
C:\Windows\system32\cmd.exe
Student Name: Amal
Press any key to continue . . .
```

On the right side of the diagram, there is a stylized illustration of a person sitting at a desk, viewed from behind. The person is wearing a blue hoodie and headphones, and is looking down at a laptop. A thought bubble originates from the person's head and points towards the code editor. The thought bubble contains several colored rectangular blocks of varying sizes and colors (green, orange, red, grey).

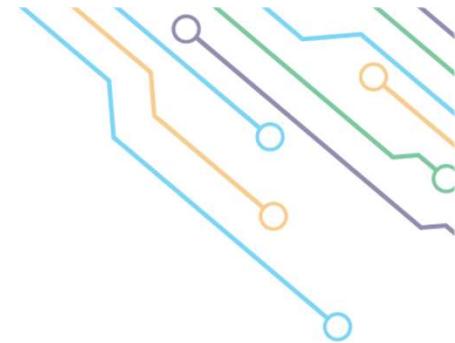


Note 1

If we didn't use virtual and override keywords, the method (`GetName()`) which implemented in the base class will be executed; because we referred student object in main to the parent class. And we'll get this result:



```
C:\Windows\system32\cmd.exe
Name: Amal
Press any key to continue . . .
```

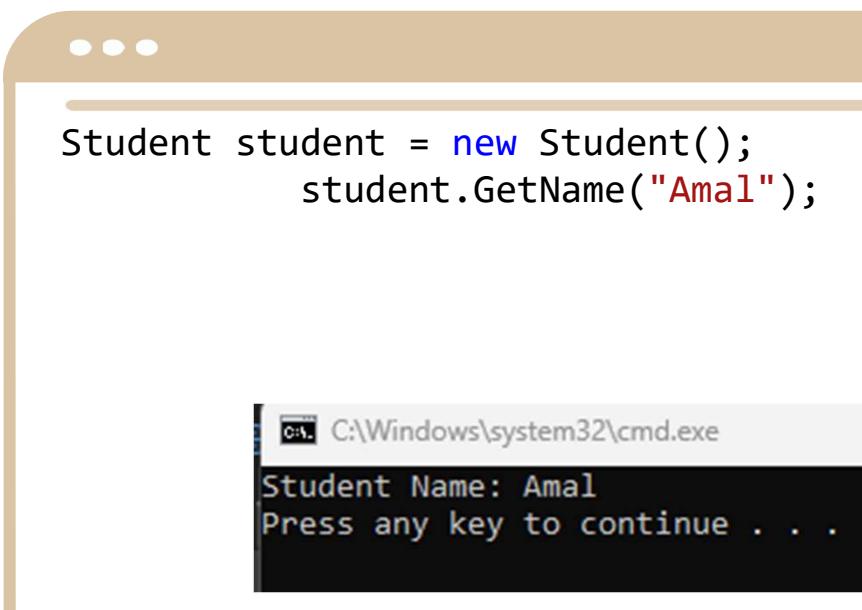


Note 2

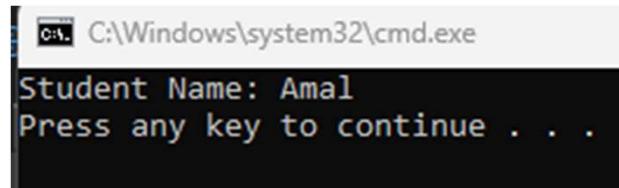
If we didn't use virtual and override keywords and referred the object student to Student class, the method (GetName()) which implemented in the child class will be executed. As in the next slide..



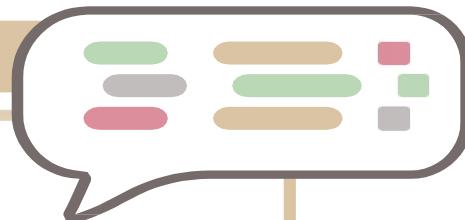
We'll get this result:

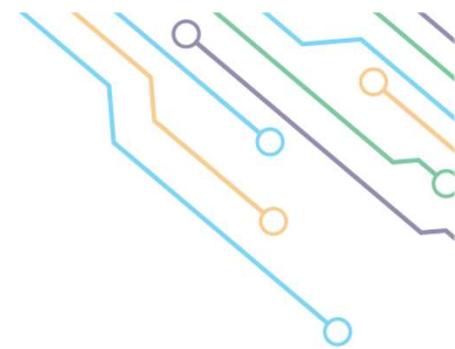


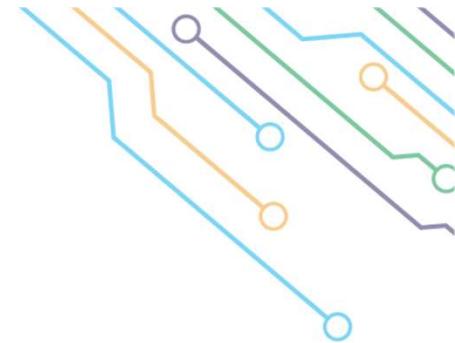
```
Student student = new Student();
student.GetName("Amal");
```



```
C:\Windows\system32\cmd.exe
Student Name: Amal
Press any key to continue . . .
```



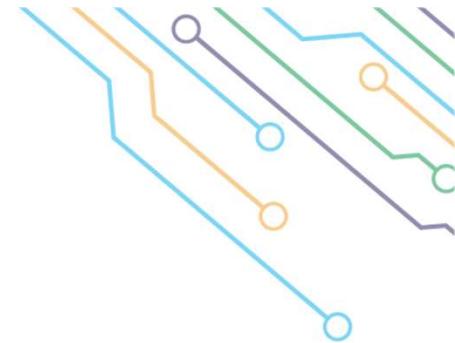




Abstraction

- ❑ One of the main concept of OOP, it is the process of data abstraction is to hide non-essential information, and only show the important details.
- ❑ Data abstraction improves data security by using abstract class or interface.



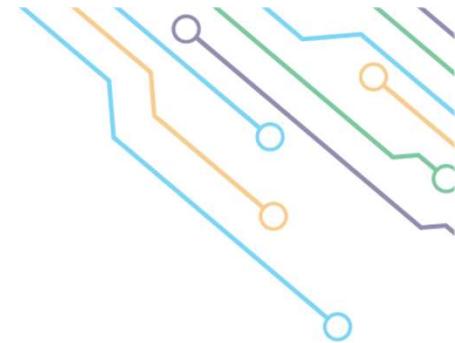


Use abstract keyword

We can use abstract keyword with:

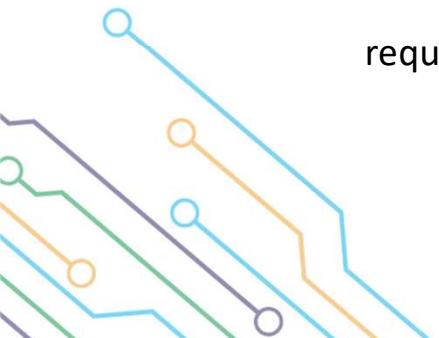
1. **Class.**
2. **Method.**
3. **Property.**

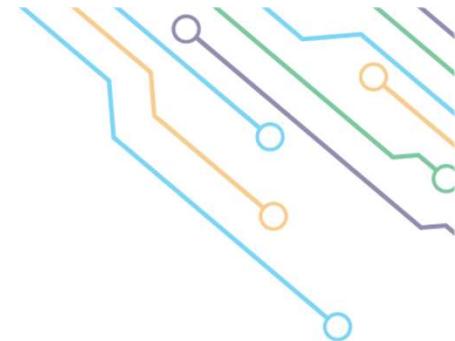




Abstract class

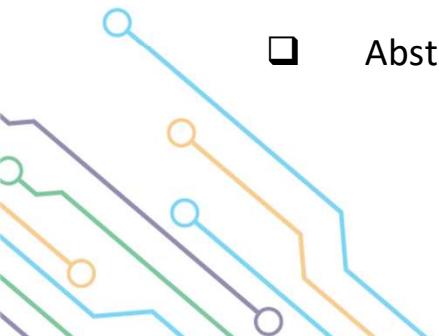
- ❑ A class that can't be used for creating an object.
- ❑ It contains abstract or non-abstract methods and properties.
- ❑ We can't use abstract with sealed modifier as they have opposite meanings
(sealed modifier prevent the class from being inherited, abstract modifier required the class to be inherited).

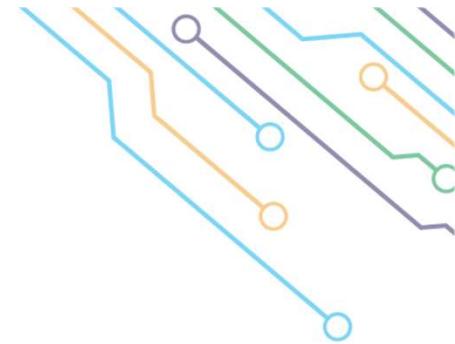




Abstract method

- ❑ Abstract modifier can be used with methods in abstract class only.
- ❑ It has no body.
- ❑ All abstract methods must be implemented in the inherited non-abstract class.
- ❑ Abstract method can't be in non-abstract class.



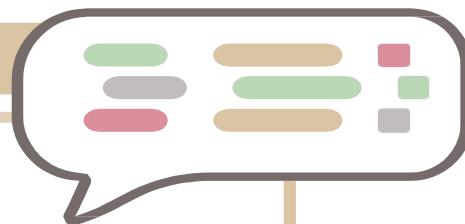


Let's Take an Example



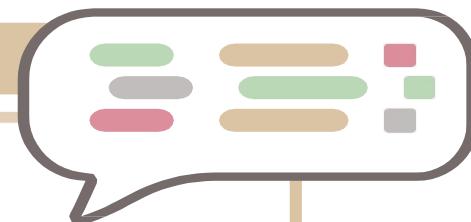
Abstract class with abstract method and abstract property

```
public abstract class Person
{
    public Person()
    {
        Console.WriteLine("This is Person");
    }
    public abstract int Id { get; set; }
    public string Name { get; set; }
```



Abstract class with abstract method and abstract property

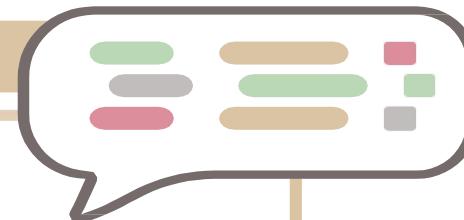
```
public DateTime Birthdate { get; set; }
    public void GetInfo()
    {
        Console.WriteLine($"Id: {Id}\nName:
{Name}\nBirthdate: {Birthdate}");
    }
    public abstract void GetAge();
}
```



The Derived Class

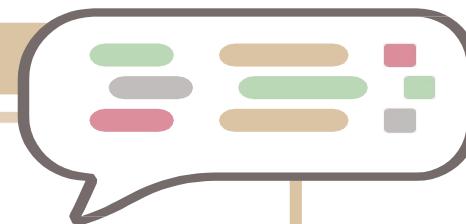
```
public class Student : Person
{
    public Student()
    {
        Console.WriteLine("This is student");
    }
    public override int Id { get; set; }

    public override void GetAge()
    {
        Console.WriteLine($"Student Age:
{DateTime.Now.Year-Birthdate.Year} ");
    }
}
```

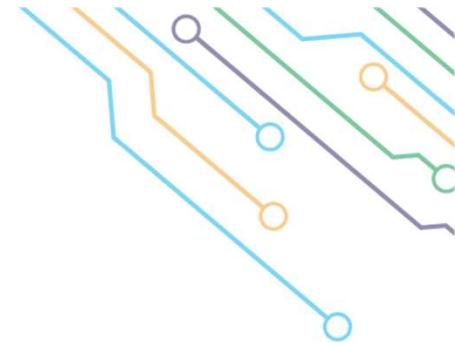


Main

```
static void Main(string[] args)
{
    Student student = new Student();
    student.Id = 1;
    student.Name = "Amal";
    student.Birthdate = new DateTime(1999, 8,
    student.GetAge();
}
```



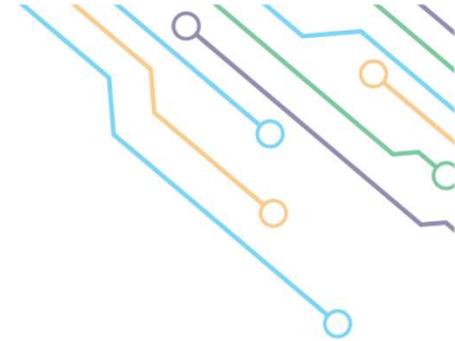




Interface

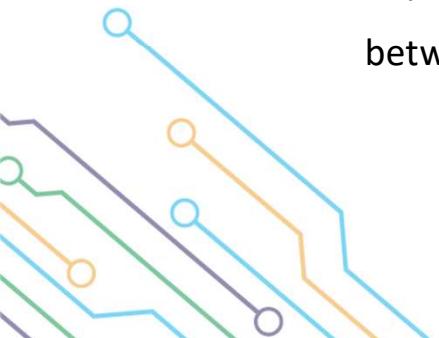
- It is completely an abstract class with some differences.
- We can achieve abstraction by using interfaces which enhance data security.
- It contains abstract properties and method, but it doesn't contain fields/variables.

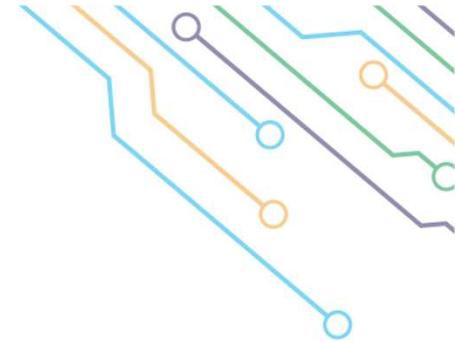




Interface

- ❑ Interface is like a contract, where any class who inherit from it must follow its rule and implement all its members.
- ❑ C# does not support multiple inheritance between classes; However, we can implement multiple interfaces in one class (Multiple inheritance is supported between class and interfaces)

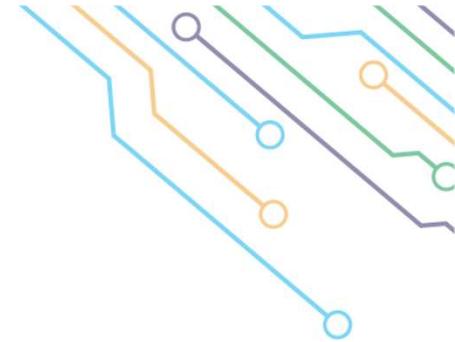




Interface

- Interface can't be instantiated (we can't create an object from it).
- All interface members should be implemented/override in the derived class.
- Method in any interface has no body, and it is abstract by default and public.
- Interface has no constructor.

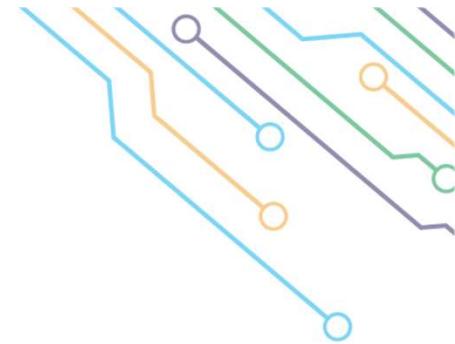




Interface

- When you name an interface use the **letter 'I'** then use pascal case to name it
(Ex: **IPerson**, **IShape**, **IEmployeeSalary**)
- Use the keyword **interface** to declare an interface.



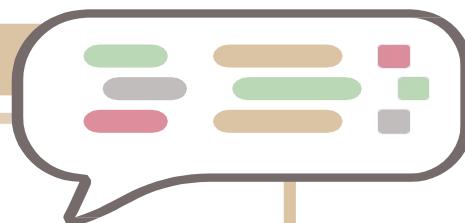


Let's Take an Example



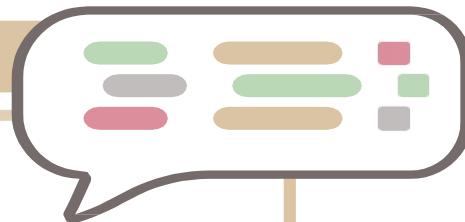
Interface

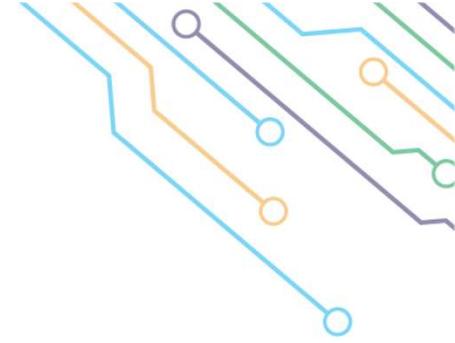
```
interface IFirst
{
    void FirstMethod(); // method
}
interface ISecond
{
    void SecondMethod(); // method
}
class FirstSecondClass : IFirst, ISecond
//Multiple inheritance
```



Continue..

```
public void FirstMethod() //Override
{
    Console.WriteLine("Method
implementation");
}
public void SecondMethod() //Override
{
    Console.WriteLine("Method
implementation");
}
```





Interface

- Create IStudent Interface with the following methods

- 1- SetStudentID()
- 2- GetStudentID()
- 3- SetName()
- 4- GetName
- 5- SetAVG()
- 6- GetAVG()
- 7- DisplayInfo()
- 8- SetMajor()

Then create three classes with university names and implement the IStudent Interface



C# Object Oriented programming (OOP)

Education and Training Solutions 2022



- 1 Polymorphism
- 2 Abstraction
- 3 Interface
- 4 Dependency Injection







Polymorphism (Many shaped)

One of the main concept of OOP, it refers to the ability of providing more than one form either for a method or property, it has two types:

- Overloading.
- Overriding.





Overload

- Same method name with different parameter occurs at the same class.
- Different parameter means different type or number of parameters.
- The binding of overloading happens at compile time.



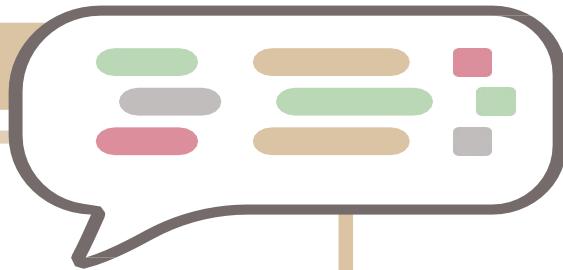


Let's Take an Example



Overloading Example

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public void GetName(string name)
    {
        Console.WriteLine("Name: " + name);
    }
    public void GetName(int Id ,string name)
    {
        Console.WriteLine("Name: " + name);
    }
}
```





Override

- Happens between base class and derived class.
- The implementation of the method which already implemented at the base class is also provided at the derived class.
- The binding of overriding happens at run time.

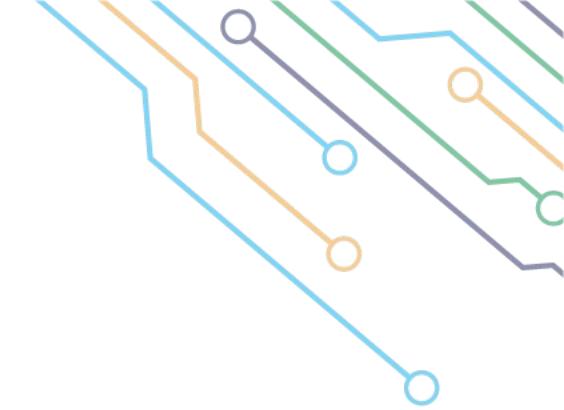




Override

- Use keyword “virtual” at the base class, “override” keyword at the derived class,
as in the next slide...



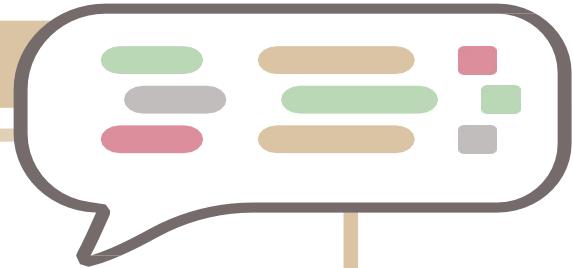


Let's Take an Example



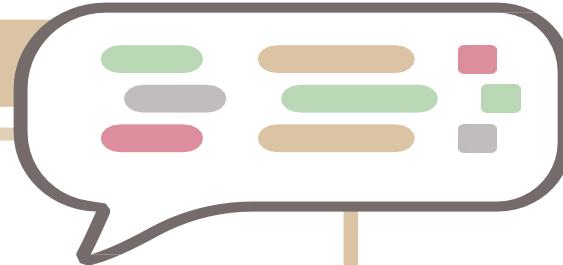
Overriding Example

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public virtual void GetName(string name)
    {
        Console.WriteLine("Name: " + name);
    }
}
```

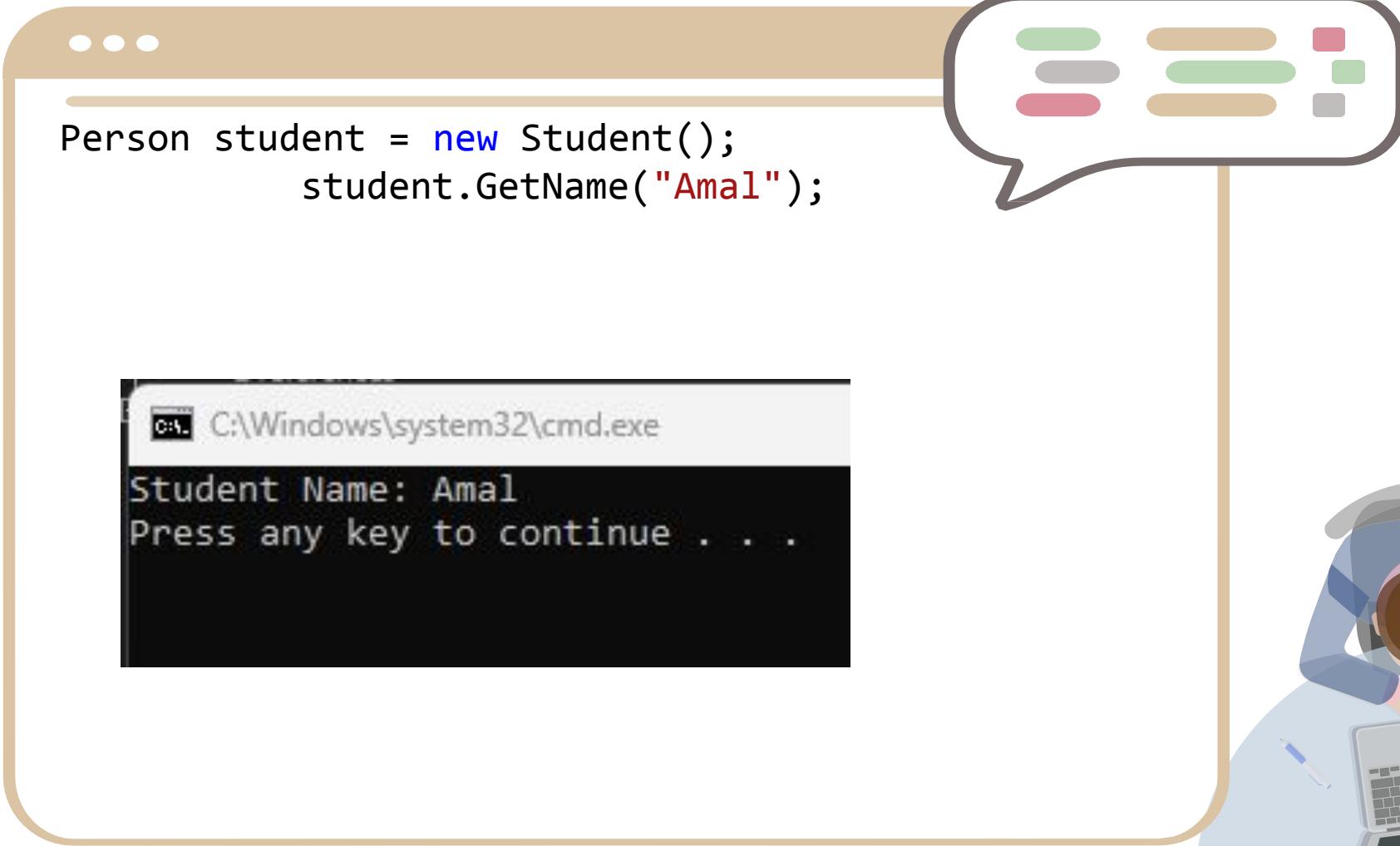


Continue..

```
public class Student:Person
{
    public override void GetName(string name)
    {
        Console.WriteLine("Student Name: " +
name);
    }
}
```



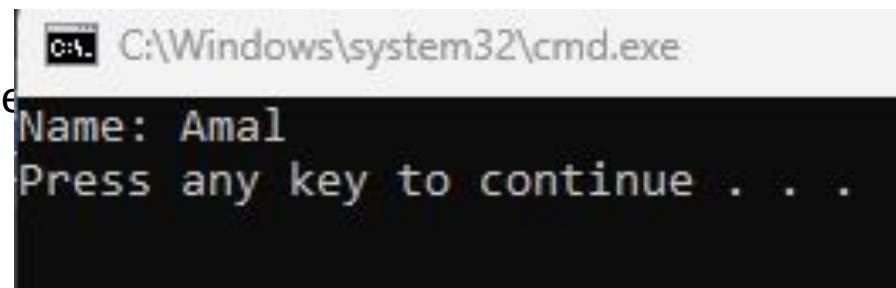
Overriding Example (Main & result) (continued..):





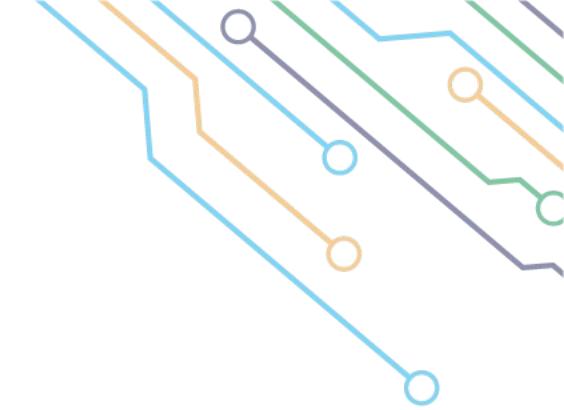
Note 1

If we didn't use virtual and override keywords, the method (GetName()) which implemented in the base class will be executed because we referred student object in main to the parent class. And we'll get this result:



```
C:\Windows\system32\cmd.exe
Name: Amal
Press any key to continue . . .
```



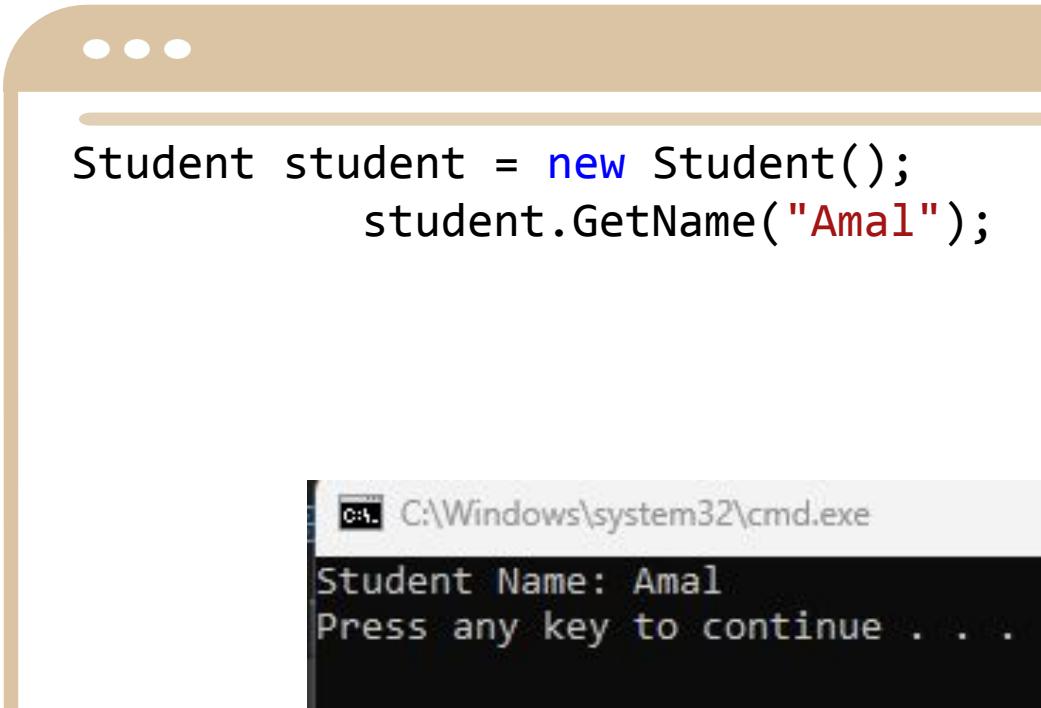


Note 2

If we didn't use virtual and override keywords and referred the object student to Student class, the method (GetName()) which implemented in the child class will be executed. As in the next slide..



We'll get this result:

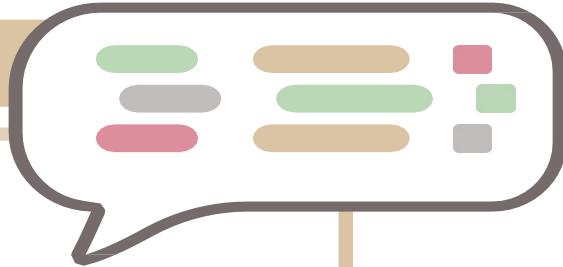


A mobile phone screen displays Java code and its execution result. The code is:

```
Student student = new Student();
student.GetName("Amal");
```

The execution result shows the output of the `GetName` method:

```
C:\Windows\system32\cmd.exe
Student Name: Amal
Press any key to continue . . .
```







Abstraction

- One of the main concept of OOP, it is the process of data abstraction is to hide non-essential information, and only show the important details.
- Data abstraction improves data security by using abstract class or interface.



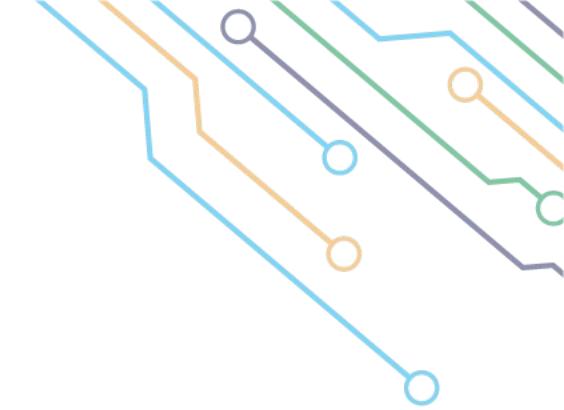


Use abstract keyword

We can use abstract keyword with:

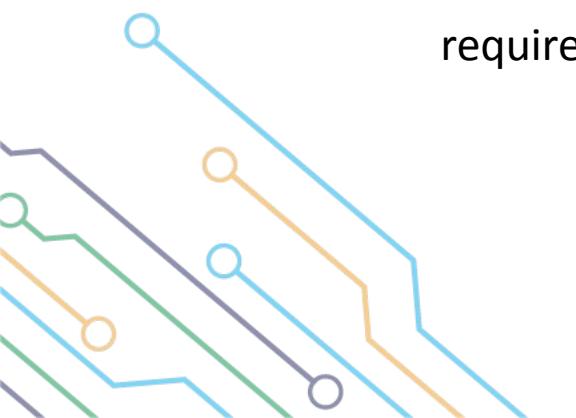
- 1. Class.**
- 2. Method.**
- 3. Property.**





Abstract class

- A class that can't be used for creating an object.
- It contains abstract or non-abstract methods and properties.
- We can't use abstract with sealed modifier as they have opposite meanings
(sealed modifier prevent the class from being inherited, abstract modifier required the class to be inherited).





Abstract method

- Abstract modifier can be used with methods in abstract class only.
- It has no body.
- All abstract methods must be implemented in the inherited non-abstract class.
- Abstract method can't be in non-abstract class.



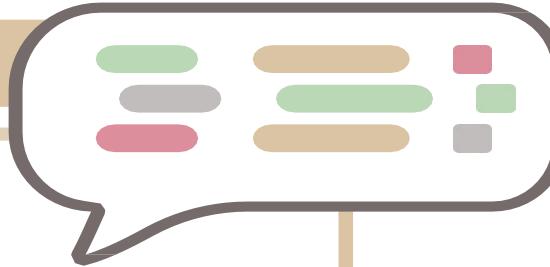


Let's Take an Example



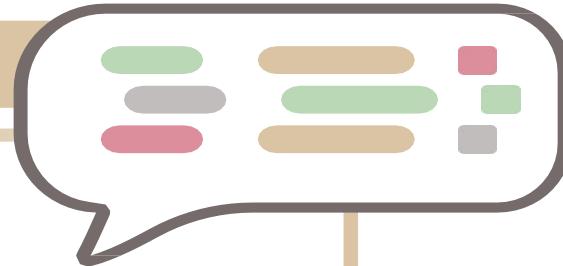
Abstract class with abstract method and abstract property

```
public abstract class Person
{
    public Person()
    {
        Console.WriteLine("This is Person");
    }
    public abstract int Id { get; set; }
    public string Name { get; set; }
```



Abstract class with abstract method and abstract property

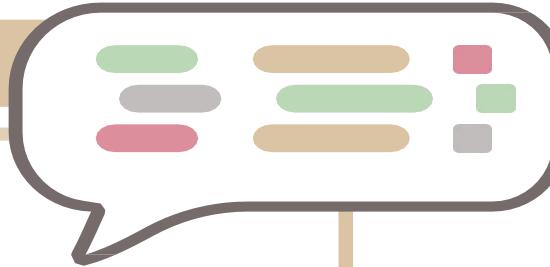
```
public DateTime Birthdate { get; set; }
    public void GetInfo()
    {
        Console.WriteLine($"Id: {Id}\nName:
{Name}\nBirthdate: {Birthdate}");
    }
    public abstract void GetAge();
}
```



The Derived Class

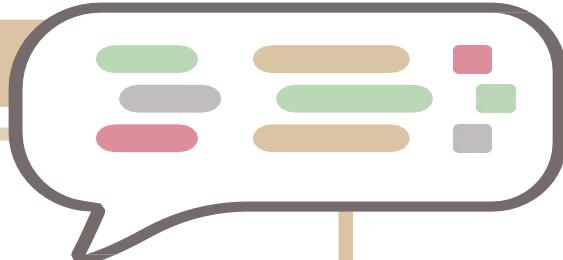
```
public class Student : Person
{
    public Student()
    {
        Console.WriteLine("This is student");
    }
    public override int Id { get; set; }

    public override void GetAge()
    {
        Console.WriteLine($"Student Age:
{DateTime.Now.Year-Birthdate.Year} ");
    }
}
```



Main

```
static void Main(string[] args)
{
    Student student = new Student();
    student.Id = 1;
    student.Name = "Amal";
    student.Birthdate = new DateTime(1999, 8,
    7);
    student.GetAge();
}
```



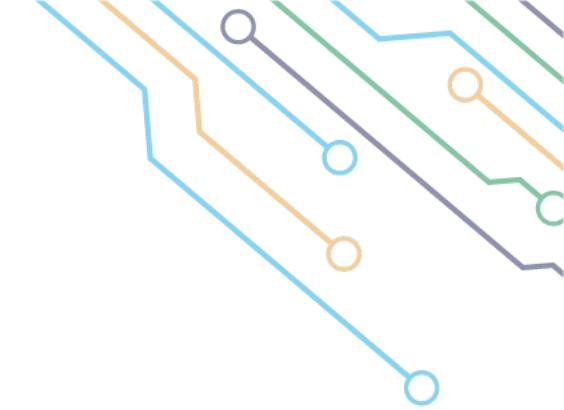




Interface

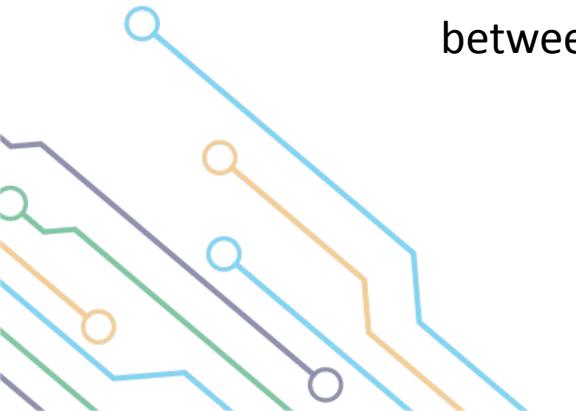
- It is completely an abstract class with some differences.
- We can achieve abstraction by using interfaces which enhance data security.
- It contains abstract properties and method, but it doesn't contain fields/variables.





Interface

- Interface is like a contract, where any class who inherit from it must follow its rule and implement all its members.
- C# does not support multiple inheritance between classes; However, we can implement multiple interfaces in one class (Multiple inheritance is supported between class and interfaces)





Interface

- Interface can't be instantiated (we can't create an object from it).
- All interface members should be implemented/override in the derived class.
- Method in any interface has no body, and it is abstract by default and public.
- Interface has no constructor.

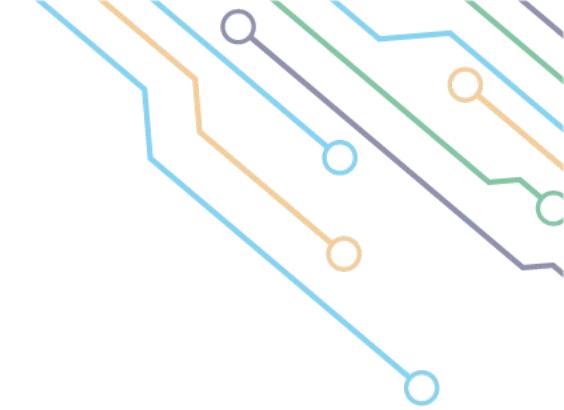




Interface

- When you name an interface use the **letter 'I'** then use pascal case to name it
(Ex: **IPerson**, **IShape**, **IEmployeeSalary**)
- Use the keyword **interface** to declare an interface.



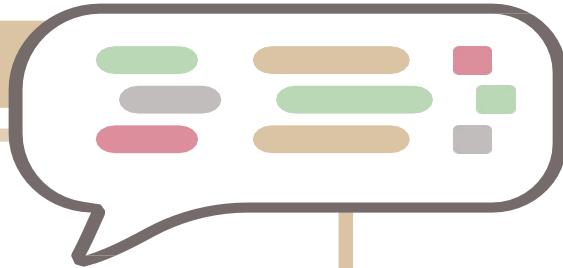


Let's Take an Example



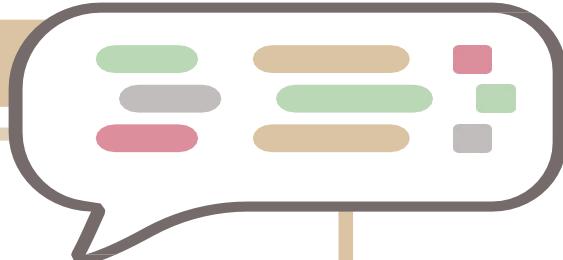
Interface

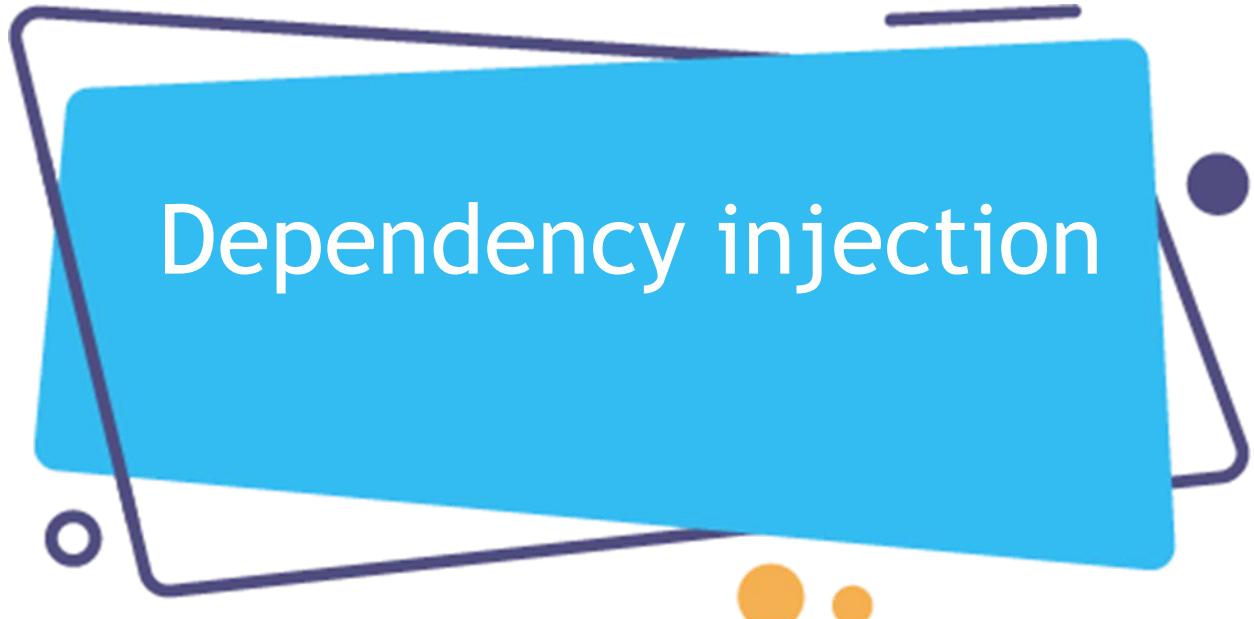
```
interface IFirst
{
    void FirstMethod(); // method
}
interface ISecond
{
    void SecondMethod(); // method
}
class FirstSecondClass : IFirst, ISecond
//Multiple inheritance
```



Continue..

```
{public void FirstMethod() //Override
{
    Console.WriteLine("Method
implemintation");
}
public void SecondMethod() //Override
{
    Console.WriteLine("Method
implemintation");
}
```

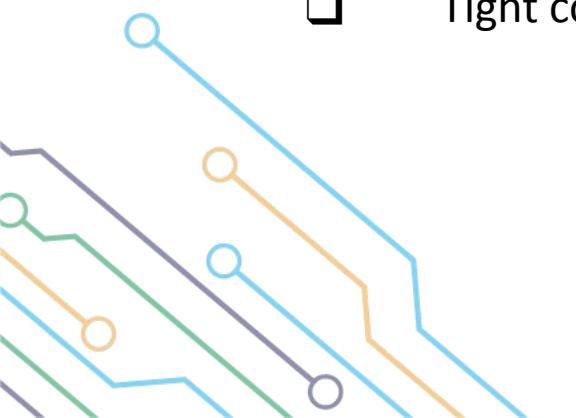


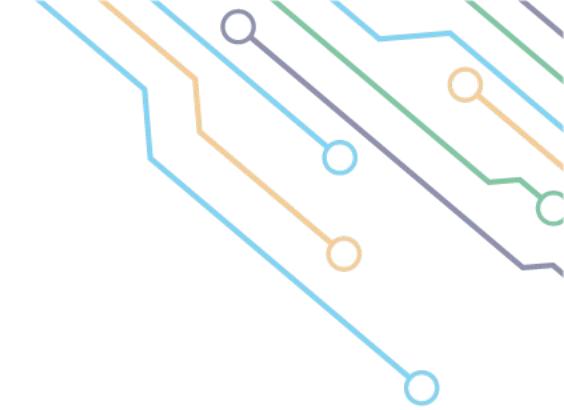




Dependency Injection (DI)

- It is software design pattern, which enables the development of **loosely coupled** code.
- Loose coupling: it is about reducing the dependencies between classes who depends on each other.
- Tight coupling: a group of classes are directly dependent on each other.



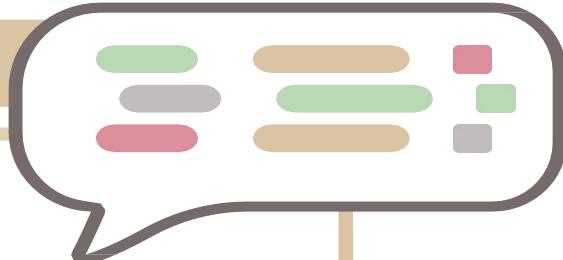


Let's Take an Example



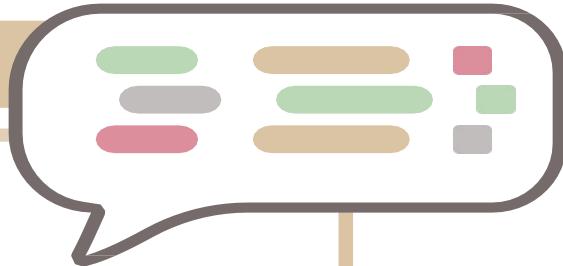
Tight coupling example (TightClass2 directly depends on TightClass1):

```
public class TightClass1
{
    public void print()
    {
        Console.WriteLine("Method...");
    }
}
public class TightClass2
{
    private TightClass1 tightClass1;
    public TightClass2(TightClass1 tightClass1)
    {
        this.tightClass1 = tightClass1;
    }
}
```



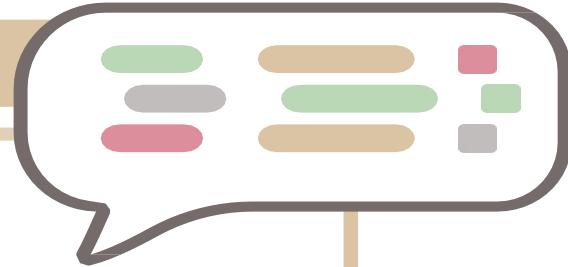
Tight coupling example (TightClass2 directly depends on TightClass2):

```
public void PrintInfo()
{
    this.tightClass1.print();
}
// main
TightClass1 tightClass1 = new TightClass1();
TightClass2 tightClass2 = new TightClass2(tightClass1);
tightClass2.PrintInfo();
```



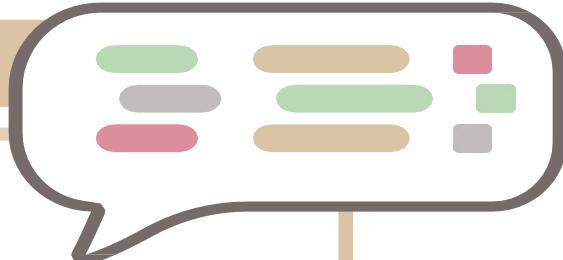
Loose coupling example

```
public interface ILooselyInterface
{
    void print();
}
public class LooselyClass1 : ILooselyInterface
{
    public void print()
    {
        Console.WriteLine("Method...");
    }
}
```



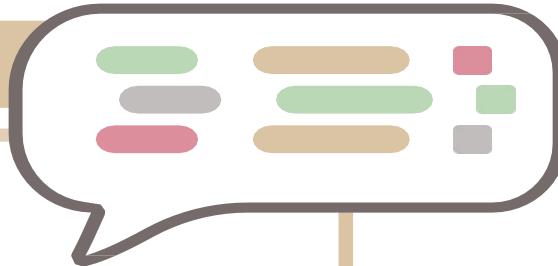
Continue..

```
public class LooselyClass2
{
    private ILooselyInterface looselyInterface;
    public LooselyClass2(ILooselyInterface
looselyInterface)
    {
        this.looselyInterface = looselyInterface;
    }
    public void printMethod()
    {
        this.looselyInterface.print();
    }
}
```



Continue..

```
// main
ILooselyInterface looselyInterface = new
LooselyClass1();
LooselyClass2 looselyClass2=new
LooselyClass2(looselyInterface);
looselyClass2.printMethod();
```





Dependency Injection (DI)

- You can reduce tight coupling between software component through DI by using interface rather than class.
- Loosely coupling means less dependency.

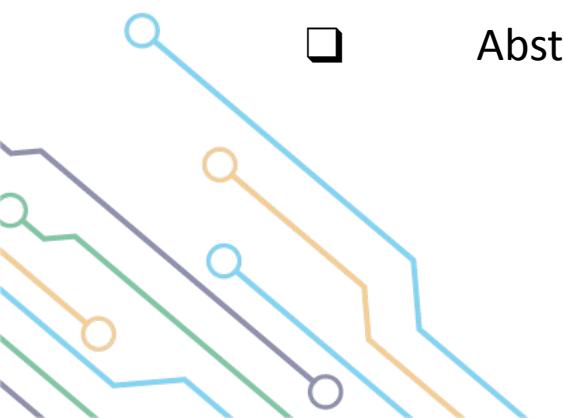


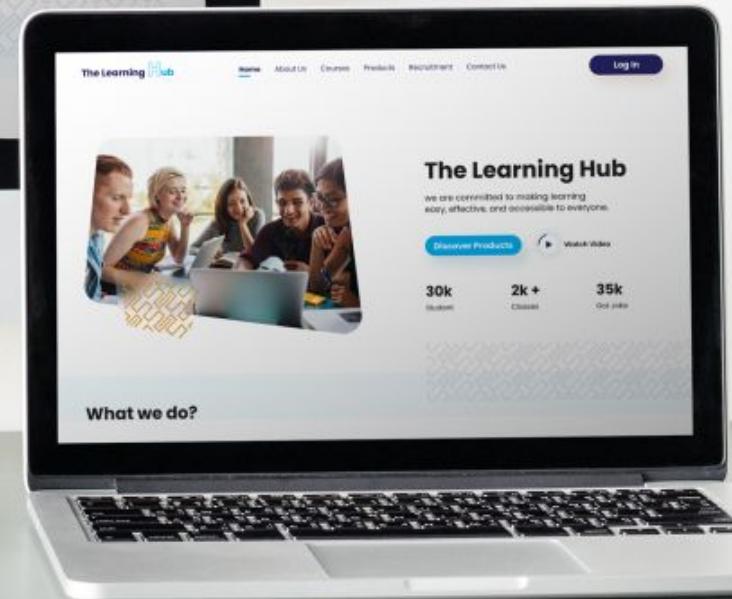
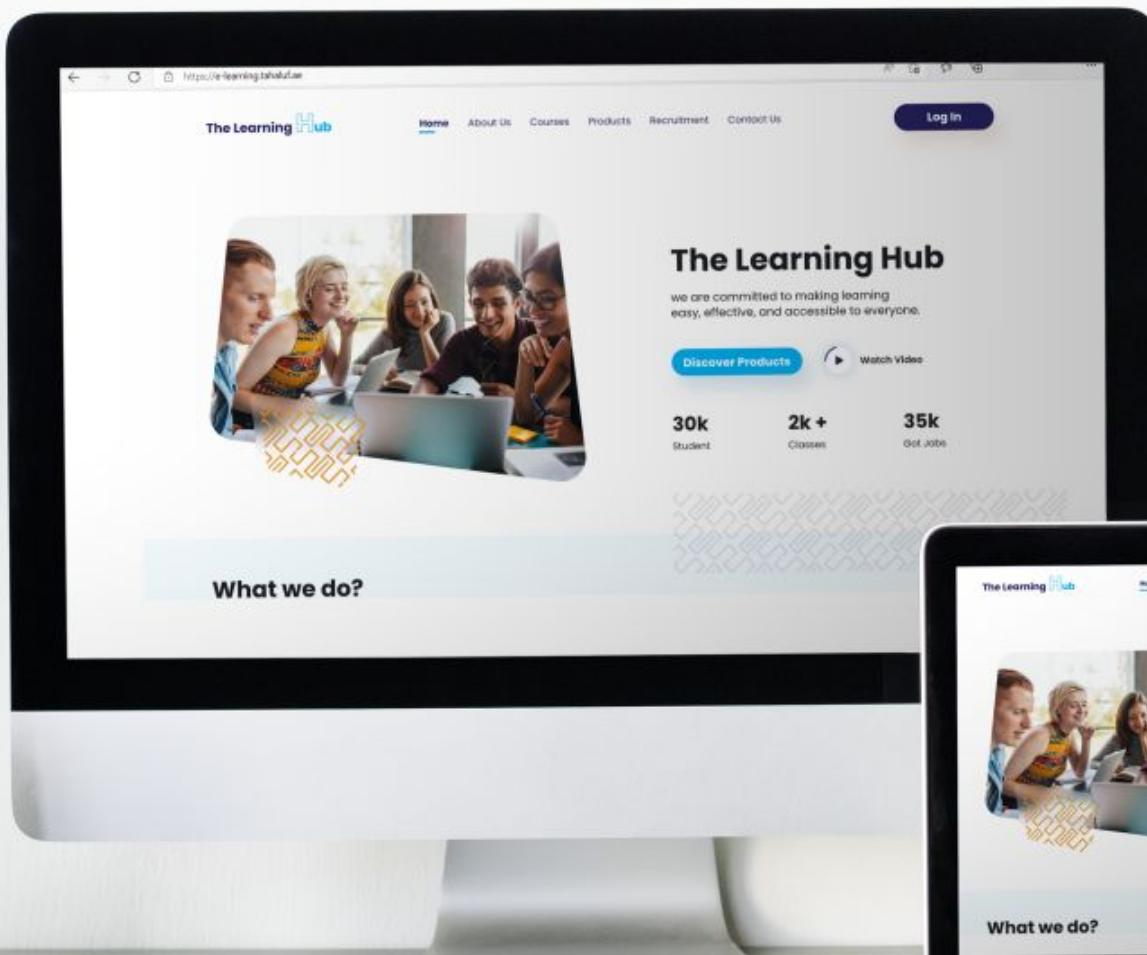


Important note 😊

The main concept of OOP:

- Encapsulation.
- Inheritance.
- Polymorphism.
- Abstraction.





C# Object Oriented programming (OOP)

Education and Training Solutions 2022



- 1 Static keyword Overview
- 2 Static Field
- 3 Static Constructor
- 4 Static Method and Property
- 5 Static Class
- 6 C# Nullable
- 7 Enums







Static Modifier

When programmers declare a class or any member in project as a static one, this mean that the member is belonged to the type itself rather than to specific object, which mean there is one and only one copy from this member.



Points to distinguish

- All members in project such as classes, methods, properties or fields can be declared as static ones.
- The static members are callable by the class name itself not by instances from it, because there is only one copy from these members to be shared among all objects.

Why static modifier is existed?

In cases programmers need to declare members to be shared among all project with the same functionalities and enforce all changes are made on to affect whole places in project that use these members, they usually declare them as a static members.





Uses of Static Field

Based on the concept of static modifier, from the most common uses of static field are:

- Declare a value to be shared among all objects.
- Count the number of instances that are created from owner class.

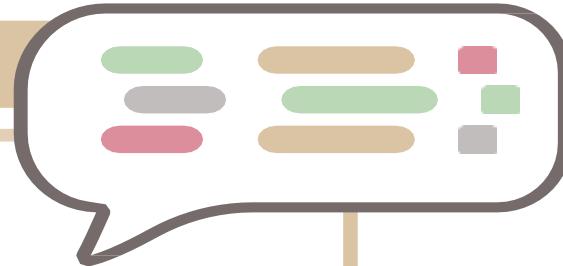


Let's Take an Example



Product class has static field

```
public class Product
{
    public int ID { get; set; }
    public string Name { get; set; }
    public double cost { get; set; }
    public static int totalNumberOfProducts;
    public Product()
    {
        Console.WriteLine("I am an instance
constructor");
        totalNumberOfProducts++;
    }
    public int GetTotalProducts(){
        return totalNumberOfProducts;
    }
}
```



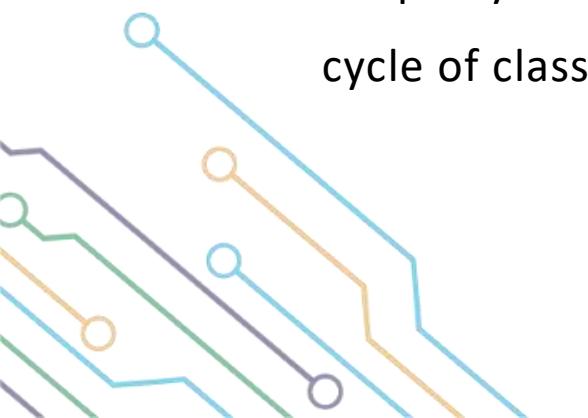
Static Constructor



Static Constructor

Firstly, there are many constraints on static constructor such as: no overloading from, has no parameters, it is called implicitly in class and there is no access modifier for.

Keep in your mind Static constructor is executed firstly and for only once in life cycle of class.

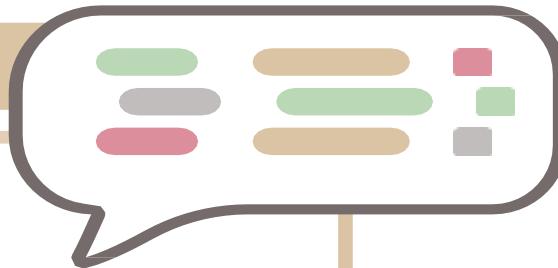


Let's Take an Example



Product class has static constructor

```
static Product()
{
    Console.WriteLine(" I am a static
constructor");
}
```







Static Method and Property

Static methods and properties can not deal with non static fields or attributes.

But by backing to the main idea of using static modifier, a question is raised!!

Do you think programmers can overload and override the static methods?!

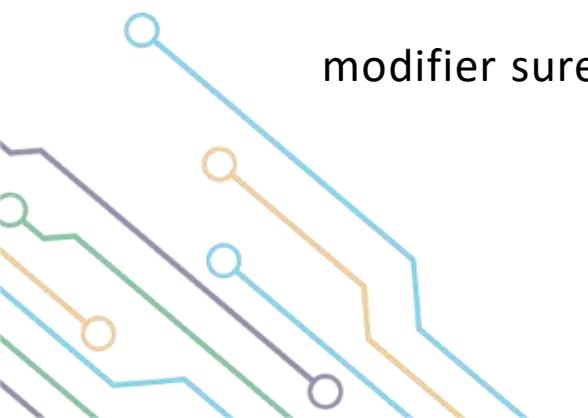




Static Method and Property

The answer is programmers still can overload the static methods but not override them and that because they belong to the owner class not to the instances themselves nor to the derived classes from.

Note that derived classes still can deal with these static methods based on access modifier surely.



Let's Take an Example



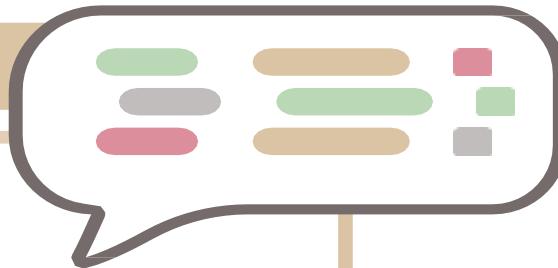
Product class has static method and property

```
static string type;
public static string Type { get=>type; set=>
type=value;}
public static void PrintTotalNumberOfProducts()
{
    //Console.WriteLine($"{Name}"); // error
    Console.WriteLine($"The total number of
products is {totalNumberOfProducts}");
}
public static void
PrintTotalNumberOfProducts (string hiText)
{
    Console.WriteLine($"{hiText}, The total
number of products is {totalNumberOfProducts}");
}
```



Product1 Class Inherits Product class

```
public class Product1 :Product
{
    public void Show()
    {
        Console.WriteLine(totalNumberOfProducts);
        PrintTotalNumberOfProducts();
    }
}
//in main
Product1 pro = new Product1();
pro.Show();
//pro. PrintTotalNumberOfProducts(); //error
Product1.PrintTotalNumberOfProducts(); //child
accesses static method by name
```



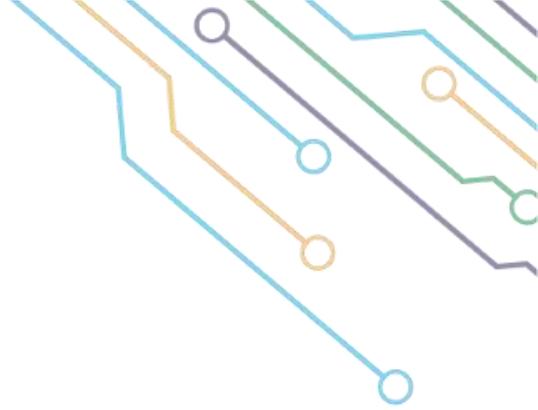




Static Class

The most common use of static modifier is by declaring non static class containing static members not entire static class, but what is the differences between the static class and non one?





Points to distinguish

- Static class contains just static members.
- There is no instances can be created from.
- Static class is considered as a sealed class.



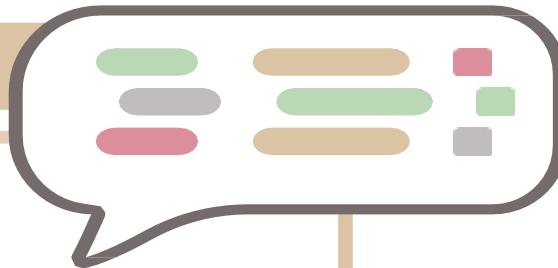
Let's Take an Example



Static Class

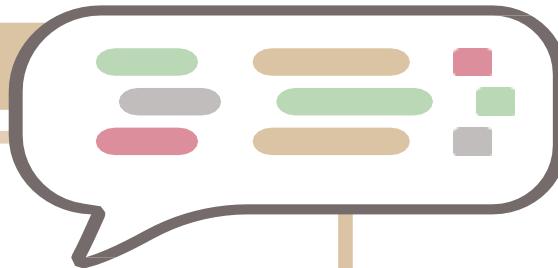
```
public static class Country
{
    public static string CountryName { get; set; }

    public static int Id { get; set; }
    public static void Print()
    {
        Console.
        Console.WriteLine($"{CountryName}
with Id {Id}");WriteLine("Country");
    }
}
```

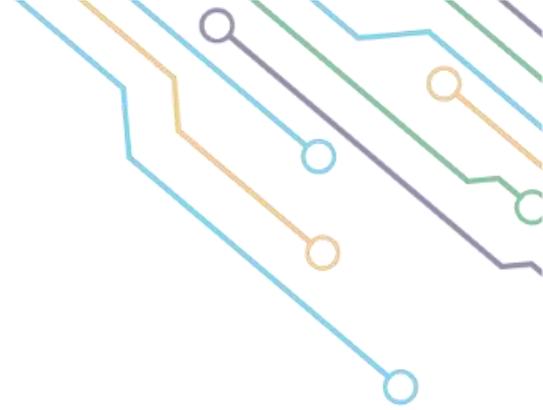


Static Class

```
static Country()
{
    Console.WriteLine("Static Country ");
}
//in main
//Country c = new Country(); // error
Country.Id = 1;
Country.CountryName = "Jordan";
Country.Print();
```

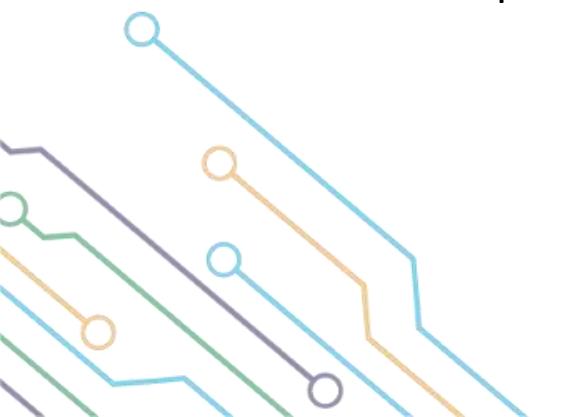






Nullable Value Types

Firstly, value types don't accept null value to be assigned to, which will throw syntax error that they are a non-nullable value type. By declaring value types as a nullable value types, it is meant to allow them to accept **Null** value in addition to the accepted ones.

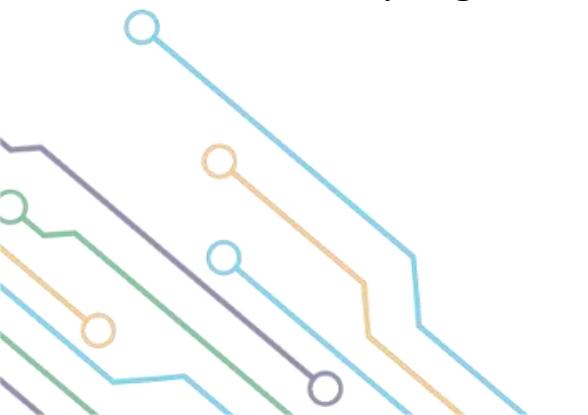


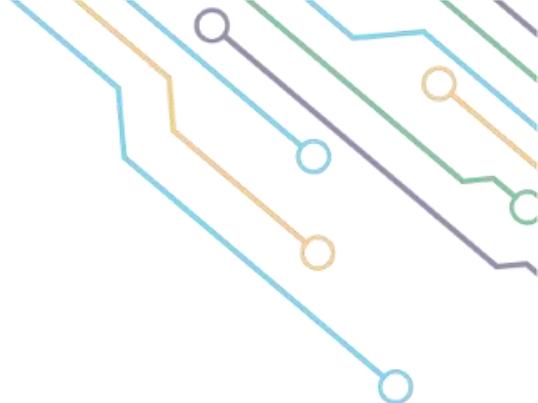


Advantages of C# Nullable

By assign undefined value of value types, many benefits can be accomplished.

- ❑ In some applications, some variables maybe undefined.
- ❑ In database, nullable values are accepted. So, by using C# nullable, programmers make the backend source compatible with database.





Nullable Syntax

Nullable type is an instance of System.Nullable<T> struct.

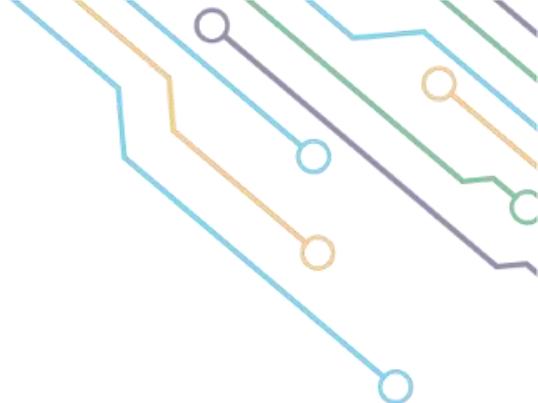
Programmers can represent variables as nullable ones as

`Nullable<T> variableName = null;` or

`T? variableName = null;`

Where T is a non-nullable value type, such as int, double and etc..



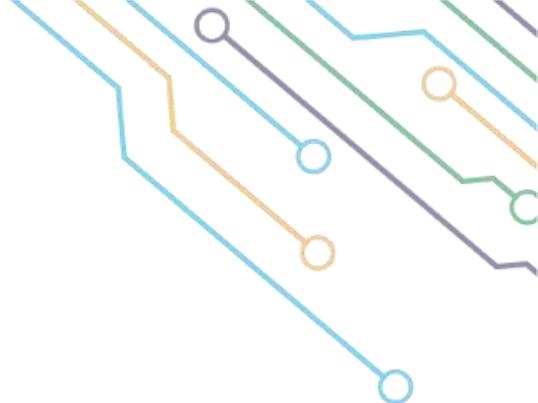


Properties of Nullable Type

HasValue it is a Boolean property returns true if the nullable value type has a non-null value.

Value it is a value type property which get the value of nullable variable only if the **HasValue** property returns true, otherwise it will throw an Invalid Operation Exception.





Assigning Nullable Value Type to a Non-Nullable Value Type

- ❑ Using `GetValueOrDefault()` method: it is a value type method which assign the value of non-nullable value type or the default of (if it has no value) to the declared variable.
- ❑ Using coalescing operator `??` : by this shorthand way, programmers can assign the non-nullable variables to the value of nullable ones or specified values.

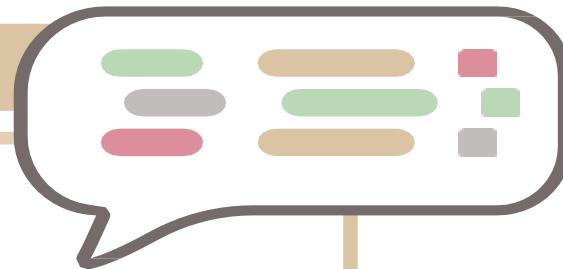


Let's Take an Example



In Main, Value property

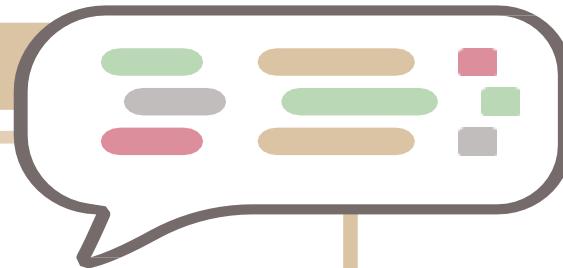
```
//double doubleValueType = null;// error
string name = null;
int? integerAsNullable = null;
Nullable<bool> booleanAsNullable = null;
Console.WriteLine("Integer as nullable Value Type =
" + integerAsNullable);// null
Console.WriteLine("Boolean as nullable Value Type =
" + booleanAsNullable);// null
Console.WriteLine("string accepts null anyway so it
= " + name);//null
```



In Main, Value property

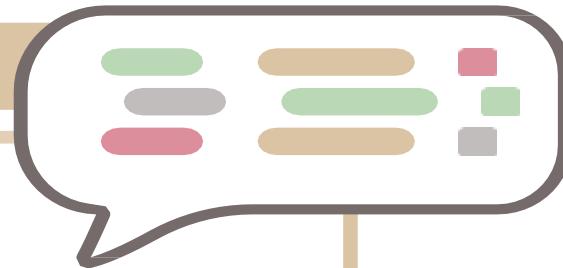
```
try
{
    Console.WriteLine("Integer as nullable Value
Type (VALUE) = " +
integerAsNullable.Value); //exception; in C# null is
not a value

}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
Console.WriteLine("-----");
```



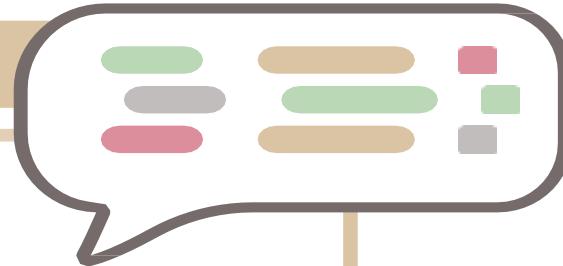
In Main, HasValue property

```
// to avoid this exception, check the nullable value
type variable by using HasValue property
if (integerAsNullable.HasValue)
{
    Console.WriteLine("Integer as nullable Value Type
(VALUE) = " + integerAsNullable.Value);
}
```



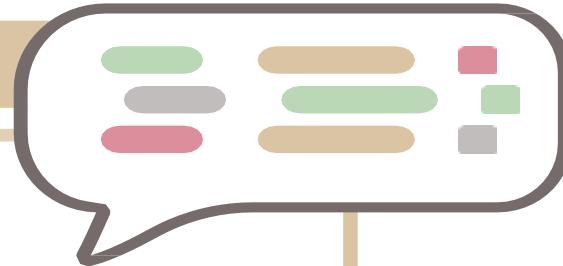
In Main, HasValue property

```
else
{
    Console.WriteLine("Integer as nullable Value Type
has null value");
    Console.Write("After give it any value, ");
    integerAsANullable = 3;
    Console.WriteLine("The Integer as nullable Value
Type (VALUE) = " + integerAsANullable.Value); //3
}
Console.WriteLine("-----");
```



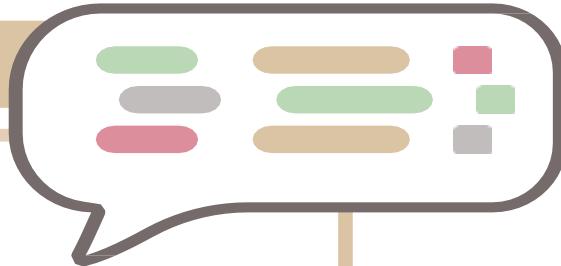
In Main, GetValueOrDefault method

```
//GetValueOrDefault method
int? nullableValue = null;
Console.WriteLine("Get the default value of Integer
when variable has null value " +
nullableValue.GetValueOrDefault());//0
Console.WriteLine($"Set the Integer variable as
{nullableValue.GetValueOrDefault(10)} when variable
has null value ");//10 note that the result will not
be saved
```



In Main, GetValueOrDefault method

```
nullableValue += 2; // null + 2 = null
nullableValue = 3;
Console.WriteLine("Note that you can make any
operation on nullable value types " +
(nullableValue.GetValueOrDefault(5)+6)); //3+6=9 note
that the result will not be stored
Console.WriteLine("-----");
```



In Main, Assign nullable value type to non-nullable value type

```
//Assign nullable value type to non-nullable value type
int nonNullableValue = nullableValue??10 ;
Console.WriteLine("The first non-nullable value type
variable = "+nonNullableValue);//3
int? nullableValue2 = null;
int nonNullableValue2 = nullableValue2 ?? 5;
```



In Main, Assign nullable value type to non-nullable value type

```
Console.WriteLine("The second non-nullable value  
type variable by using coalescing operator "+  
nonNullableValue2); //5  
nonNullableValue2  
=nullableValue2.GetValueOrDefault(5);  
Console.WriteLine("The second non-nullable value  
type variable by using GetValueOrDefault() method  
=" +nonNullableValue2); //5  
Console.WriteLine("-----");
```



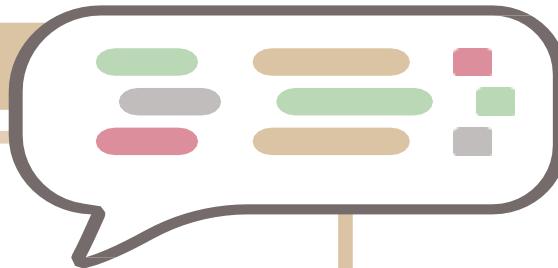
Nullable Value Types

Create a class contains a function that calculate the summation of two numbers and return null if they are undefined or missed?



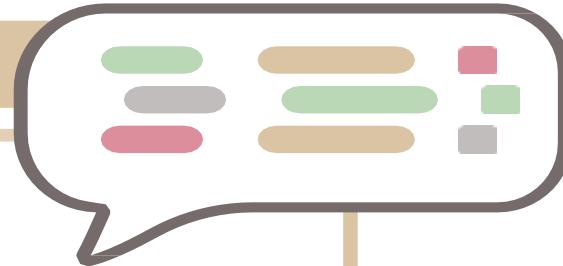
Exercise Solution

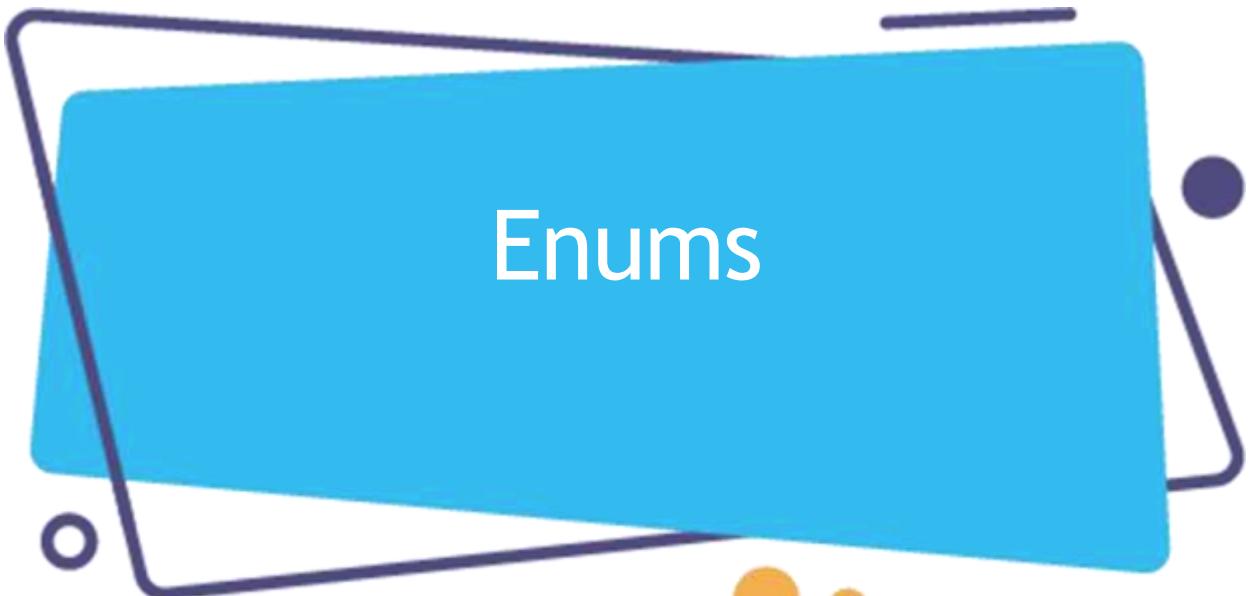
```
class Summation
{
    double? number1;
    double? number2;
    public double? Number1 { get; set; }
    public double? Number2 { get; set; }
    public double? Addition()
    {
        double? result = Number1 + Number2;
        if (result.HasValue)
        {
            return result.Value;
        }
        else
            return null;}}
```



Exercise Solution

```
//in main
Summation sum = new Summation();
Console.WriteLine($"nullable value ({sum.Number1})
+ nullable value ({sum.Number2}) =
{sum.Addition()}");//null
sum.Number1 = 3;
Console.WriteLine($"nullable value ({sum.Number1}) +
nullable value ({sum.Number2}) =
{sum.Addition()}");//null
sum.Number2 = 10;
Console.WriteLine($"nullable value ({sum.Number1}) +
nullable value ({sum.Number2}) =
{sum.Addition()}");//13
```





Enums

Enumeration or Enum is a value data type which contains a set of constant or read only members. It is declared by Enum keyword in namespace or in class.

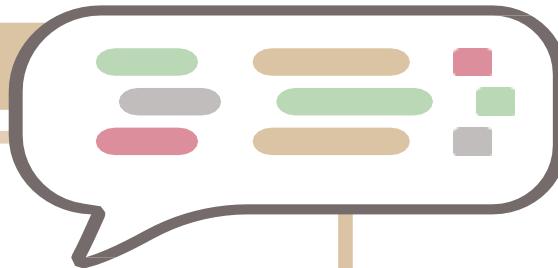
- Enum members have indexes start by 0 and incremented by 1, and it is changeable.
- An explicit cast must be done to convert from enum to Integer.

Let's Take an Example



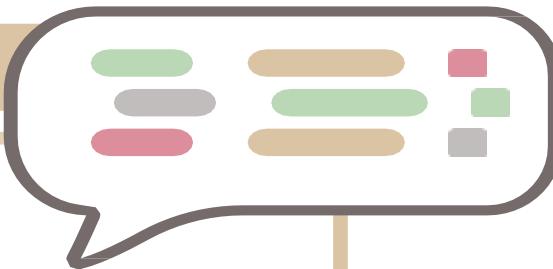
Password Enum

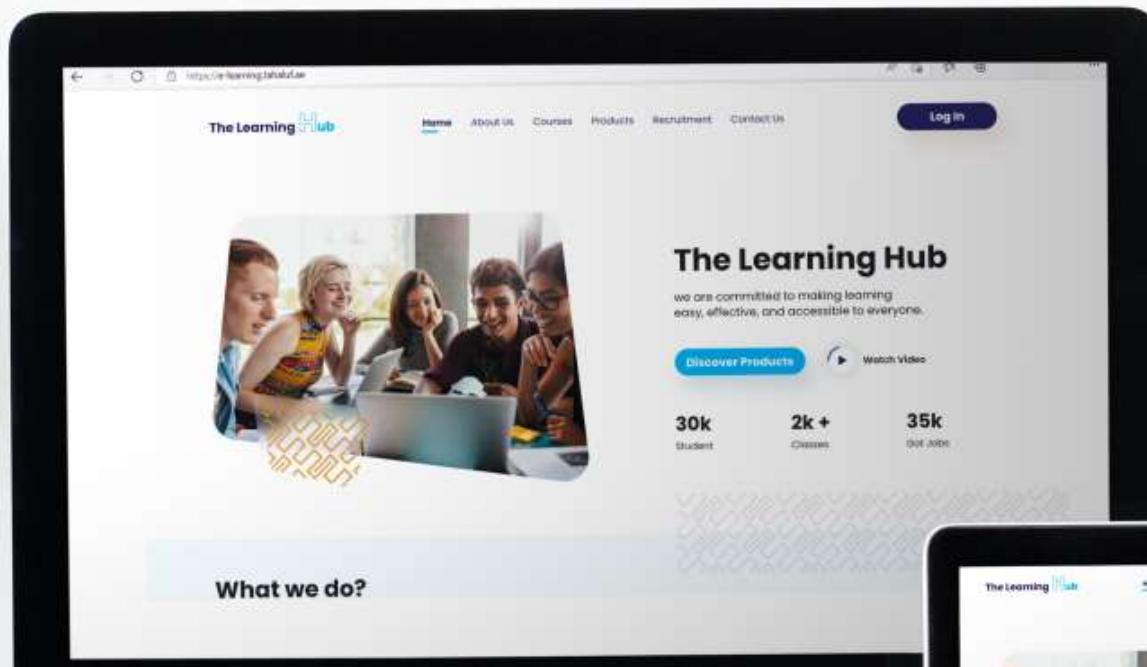
```
//in namespace
enum Password { weak, good, strong }
//in main
Password password = 0;
Password password1=(Password)1;
Password password2=Password.strong;
Console.WriteLine(${(int)password}.{password} -
{ (int)password1}.{password1} -
{ (int)password2}.{password2}");
Console.WriteLine("Enter password :");
string pass = Console.ReadLine();
```



Password Enum

```
if (pass.Length<6)
{
    Console.WriteLine($"Password is {password}");
}
else if (pass.Length>6 && pass.Length<=8)
{
    Console.WriteLine($"Password is {password1}");
}
else
{
    Console.WriteLine($"Password is {password2}");
}
```





C# Object Oriented programming (OOP)

Education and Training Solutions 2022



- 1 Generic
- 2 Exception handling







What is generic?

- It is the concept that allow type (String, Integer ... etc. and user-defined types) to be a **parameter** to methods, classes (as type for its property), and interfaces.
- Means that you can provide the actual type during invocation.

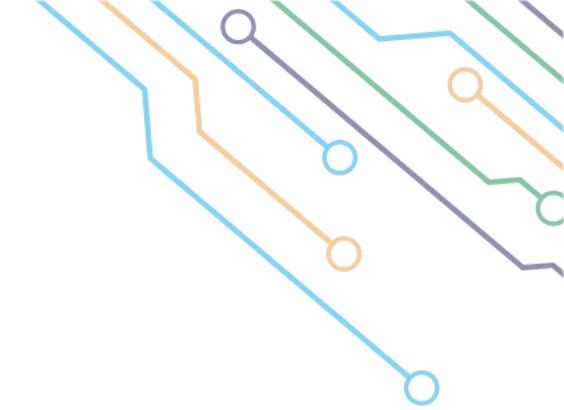




Generic method

- Is method that can be overloaded on several type parameters.
- You could specify its datatype when calling it.





Let's Take an Example



Generic method example

```
public class Print
{
    public void PrintInfo<T>(T message)
    {
        Console.WriteLine(message);
    }
}
// in main
Print print = new Print();
print.PrintInfo<string>("Raghad");
print.PrintInfo<int>(1);
print.PrintInfo<double>(5.5);
print.PrintInfo<bool>(true);
```





Generic class

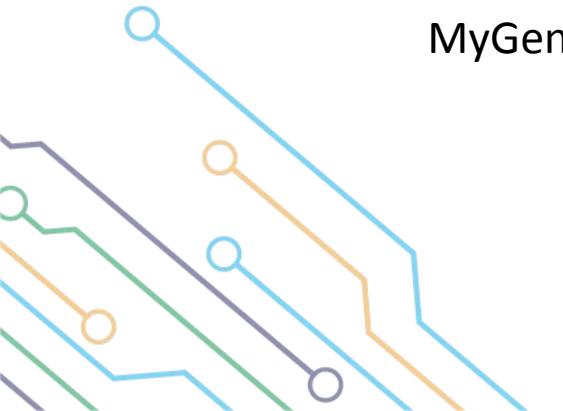
- ❑ It allows you to define type-safe data structures.

- ❑ Declaring a Generic Class:

```
public class MyGenericClass<U>
```

- ❑ Instantiating a Generic Class:

```
MyGenericClass<int> MyGeneric = new MyGenericClass<int>();
```





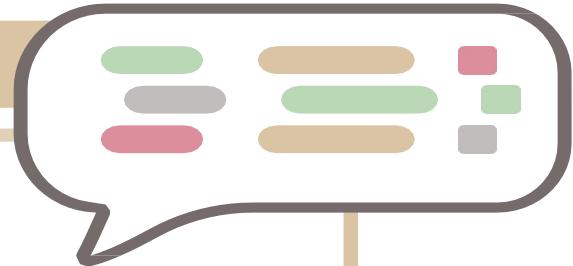
Generic class

We will take an example about how you can use generic class to set and get array items with dynamic array length, lets see how...



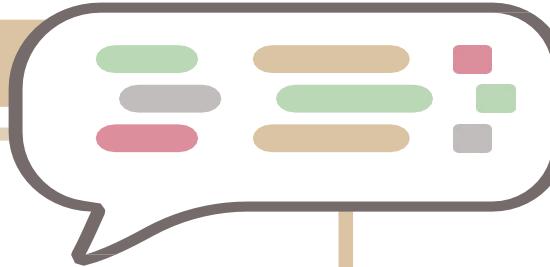
Generic class example

```
public class MyGenericArray<T>
{
    private T[] array;
    public MyGenericArray(int size)
    {
        array = new T[size + 1];
    }
    public T getItem(int index)
    {
        return array[index];
    }
    public void setItem(int index, T value)
    {
        array[index] = value;
    }
}
```



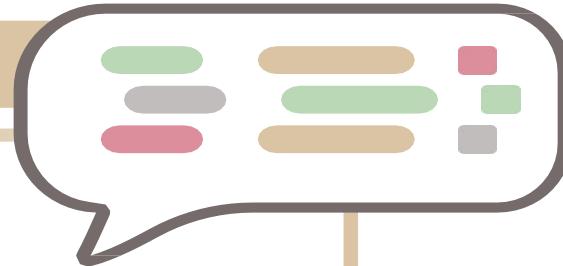
Main: set (int) values:

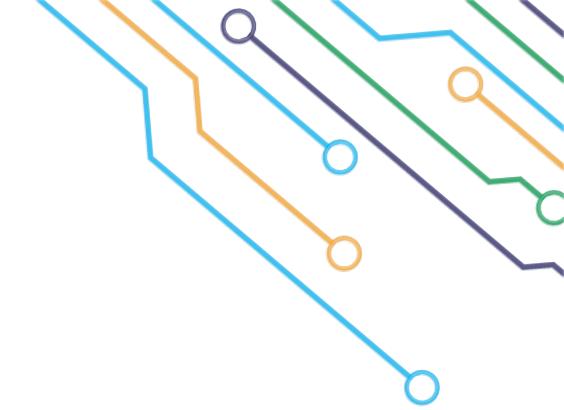
```
static void Main(string[] args)
{
    MyGenericArray<int> intArray = new
MyGenericArray<int>(5);
    //setting values
    for (int c = 0; c < 5; c++)
    {
        intArray.setItem(c, c * 5);
    } //retrieving the values
    for (int c = 0; c < 5; c++)
    {
        Console.WriteLine(intArray.getItem(c) + " ");
    }
    Console.WriteLine();
}
```



Main: set (char) values:

```
static void Main(string[] args)
{
    //declaring a character array
    MyGenericArray<char> charArray = new
    MyGenericArray<char>(5);
    //setting values
    for (int c = 0; c < 5;
c++){charArray.setItem(c, (char)(c + 97 ));
    }
    //retrieving the values
    for (int c = 0; c < 5; c++)
    {
        Console.WriteLine(charArray.getItem(c) + " ");
    }
}
```



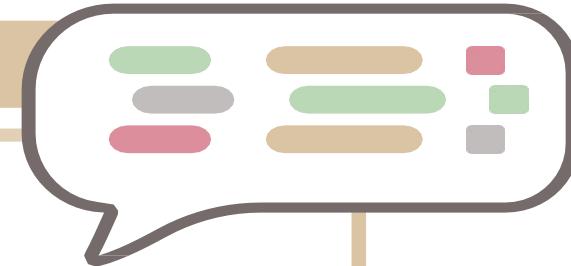


Create generic class called 'Comparison' that receives 2 generic datatypes and has 1 method called Compare; it receives 2 generic inputs and print if they are equals to each other or not.



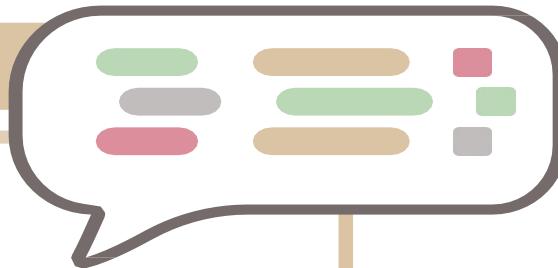
Comparison class

```
public class Comparison<T1, T2>
{
    public void Compare(T1 inp1, T2 inp2)
    {
        if (inp1.ToString() == inp2.ToString())
        {
            Console.WriteLine("The 2 input are
equal to each other");
        } else
            Console.WriteLine("The 2 input are
not equal to each other");
    }
}
```



Main

```
static void Main(string[] args)
{
    Comparison<int, string> comparison = new
Comparison<int, string>();
    comparison.Compare(0, "0");
}
```







Exception handling

- Exception handling helps you to deal with any unexpected situations that might occur when a program is running.
- It use **try**, **catch** and **finally** keywords to handle failure.





Exception handling

- When you run a program and an exception is thrown, if it finds no appropriate catch block, the program will terminate the process and display a message to the user.

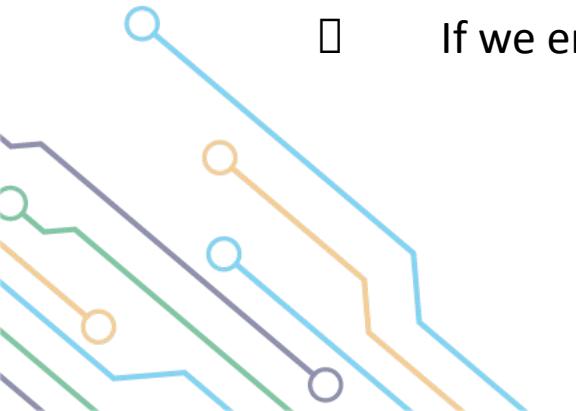




Exception handling (Example)

A program that divides 2 numbers;

- If the divisor was 0 the program will terminate and **DivideByZeroException** exception will be thrown.
- If we enter string instead of int or double **FormatException** will be thrown.
- If we enter too large or too small number, **OverflowException** will be thrown.





Exception handling

We will use try and catch to catch any unexpected exception in our program as shown:

```
try
{
    // put the code that may raise
exceptions here
}
catch
{
    // handle exception here
}
```





Let's Take an Example



Exceptions handling

```
try
{
    Console.WriteLine("Enter First number: ");
    var num1 = int.Parse(Console.ReadLine());
    Console.WriteLine("Enter second number: ");
    var num2 = int.Parse(Console.ReadLine());

    Console.WriteLine("The result = " + num1 / num2);
}
catch
{
    zero.");
}
```

Console.Write("Error division by



Exceptions handling

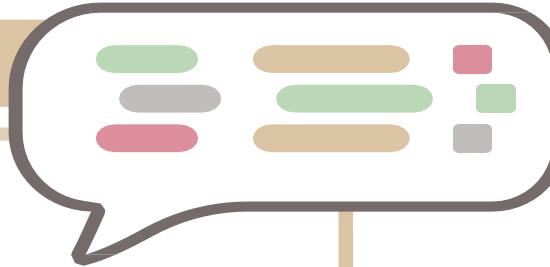
```
Console.Write("Please enter a number to divide 100  
on: ");  
try  
{  
    int num = int.Parse(Console.ReadLine());  
    int result = 100 / num;  
  
    Console.WriteLine("100 / {0} = {1}", num, result);  
}
```

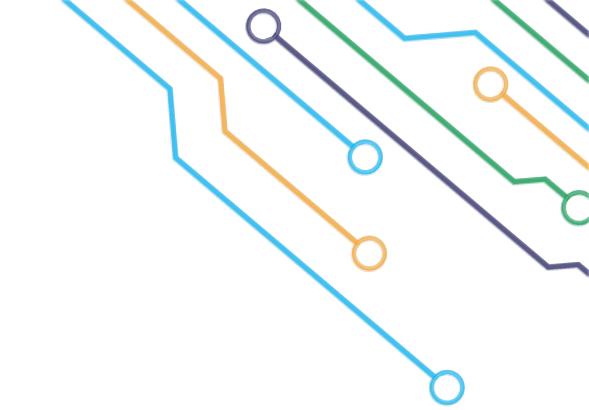


Continue..

```
catch (DivideByZeroException ex)
{
    Console.Write("Cannot divide by zero. Please try
again.");
}

catch (FormatException ex)
{
    Console.Write("Not a valid format. Please try
again.");
}
```



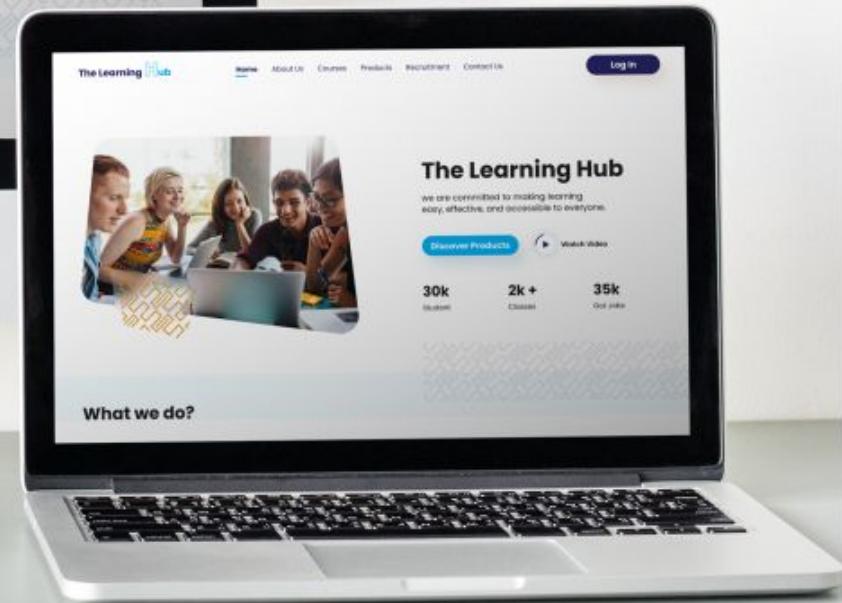
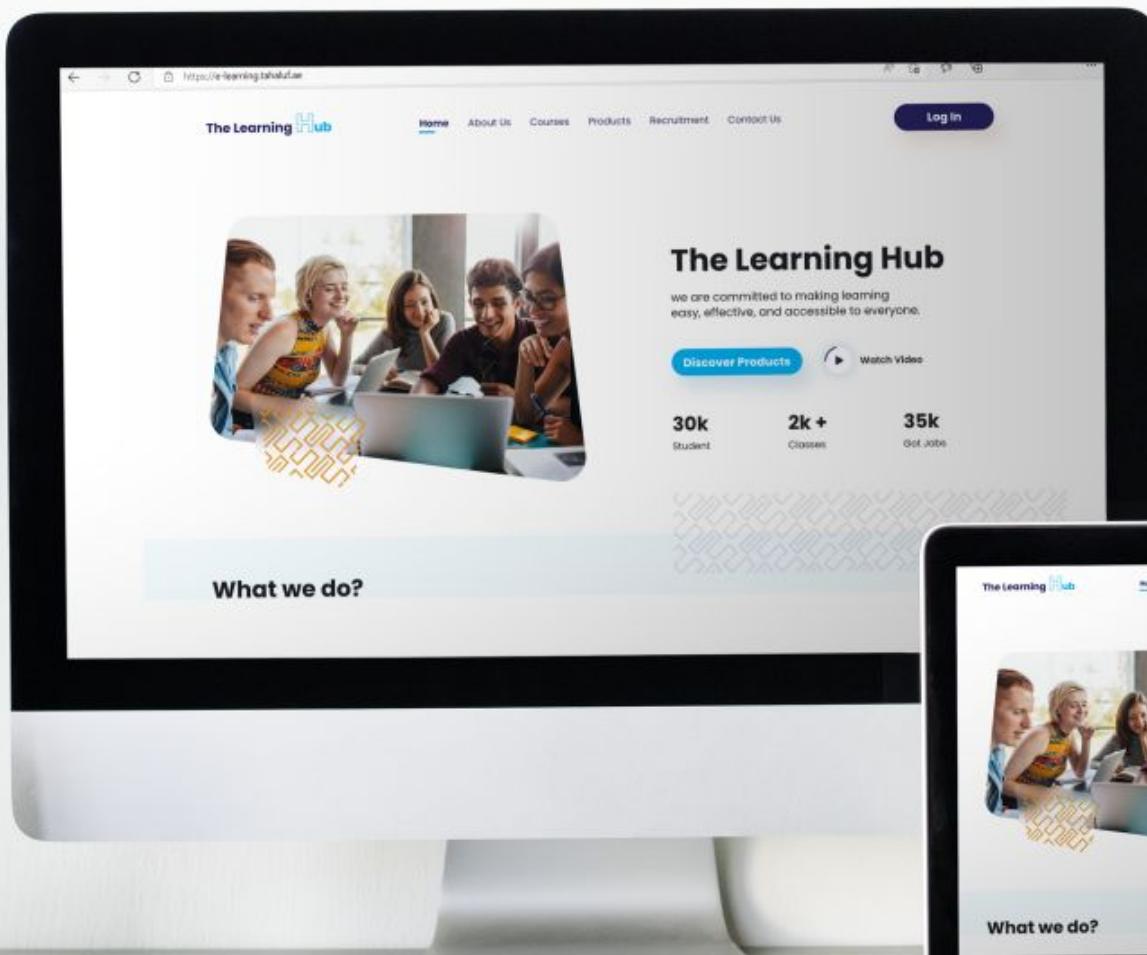


For you:

Search about if you can throw an exception made by you; for example: you want to throw exception if the user enter an odd number..

Try it if possible 😊!





C# Object Oriented programming (OOP)

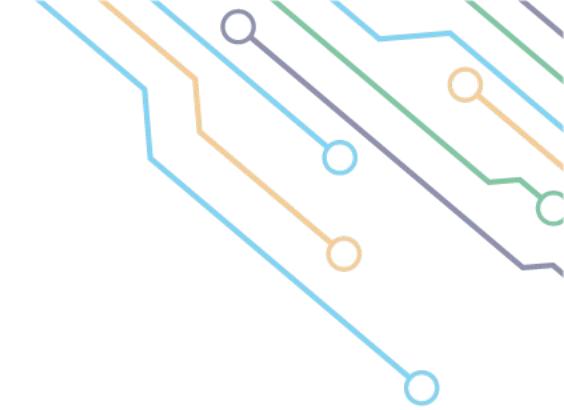
Education and Training Solutions 2022



- 1 C# List
- 2 ArrayList
- 3 LINQ



C# List



List in C#

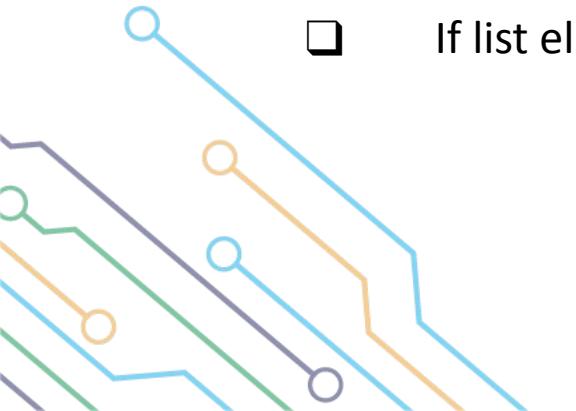
C# list is a class used to store elements from different datatypes or objects. It is defined or declared under System. Collection. Generic namespace.

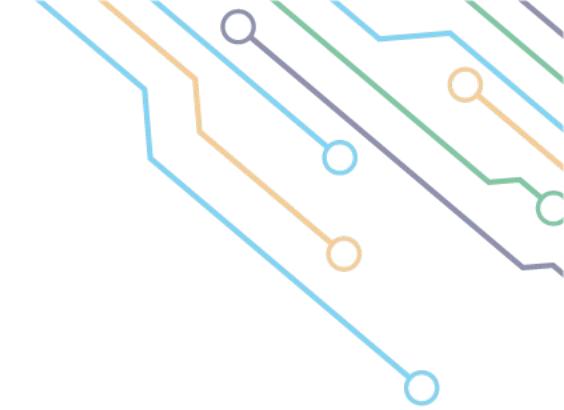




Points to distinguish

- The elements of list can be accessed by their indexes starting with zero.
- C# list has dynamic size which means programmers can change, add or delete, etc.. from list .
- It allows elements duplication with different indexes.
- If list elements are reference ones, then the list can accept null values.





C# List Syntax

List is defined as the following where T is any datatype, value or reference one.

- `List<T> list_name = new List<T>();`
- `List<T> list_name = new List<T>(initial Capacity);`

Then, many operations can be done on defined list.





Let's Take an Example



Define List of string

```
//You can declare the list using List<string>
var stringList= new List<string>()
{
    "The",//0
    "Learning",//1
    "Hub");//2
}
//the same you can declare the item using string
foreach (var item in stringList)
{
    Console.WriteLine(item);
}
Console.WriteLine("*****");
```





C# List Methods

`Add()` to add elements to the defined List from different datatypes.

`Insert()` to add elements to the defined List at specific location.

`Contains()` a Boolean method to check containing the determined element.





C# List Methods

`Remove()` to remove the determined element by its value.

`RemoveAt()` to remove the determined element by its index.

`ReamoveRange()` to remove range of element.

`Clear()` to remove all elements in the list.





C# List Methods

`Sort()` to arrange the element ascending.

`BinarySearch()` an integral method return the number of elements of specified one.

`Reverse()` to reflect the order of elements in the list.



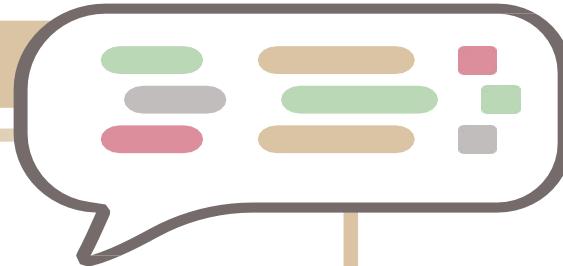
Make some operations on

```
stringList.Add("2023");
foreach (var item in stringList)
{
    Console.WriteLine(item);
}
Console.WriteLine("*****");
stringList.Insert(2, "Shaimaa");
foreach (var item in stringList)
{
    Console.WriteLine(item);
}
```



Make some operations on

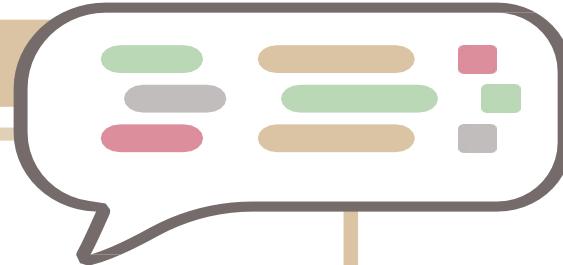
```
Console.WriteLine("*****");
stringList.Remove("IT");
stringList.Remove(stringList[3]);
foreach (var item in stringList)
{
    Console.WriteLine(item);
}
Console.WriteLine("*****");
stringList.RemoveAt(2);
foreach (var item in stringList)
{
    Console.WriteLine(item);
}
```



Make some operations on

```
Console.WriteLine("*****");
    stringList[0] = "Shaimaa";
    foreach (var item in stringList)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("*****");
if (stringList.Contains("IT"))
    Console.WriteLine("There is IT element in the
                    list");
else
    Console.WriteLine("There is no IT element in
                    list");
```

the



Make some operations on

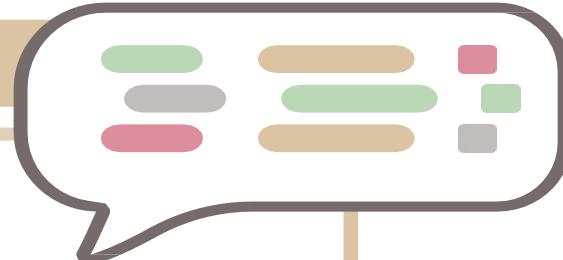
```
//stringList.BinarySearch("Shaimaa");
    int x = stringList.BinarySearch
(stringList[0]);
    Console.WriteLine($"# of elements with
        {stringList[0]} is {x}" );

Console.WriteLine("*****");
    stringList.Sort();
    foreach (var item in stringList)
{
    Console.WriteLine(item);
}
```



Make some operations on

```
Console.WriteLine("*****");
stringList.Clear();
foreach (var item in stringList)
{
    Console.WriteLine(item);
}
```



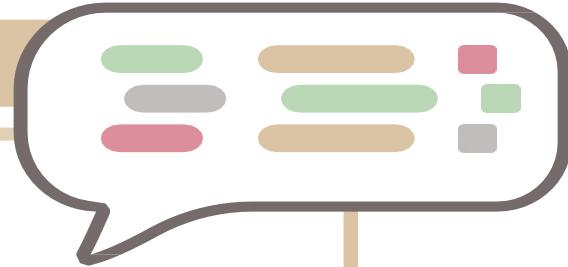
List of user defined objects

Create a class student contains some properties like name, Id and phone number, and then define a list of objects from this class?



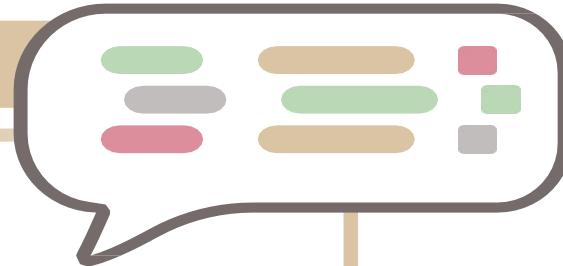
Exercise Solution

```
class Student
{
    public string Name { get; set; }
    public int Id { get; set; }
    public string PhoneNumber { get; set; }
}
// in main
Student s = new Student();
var studentList = new List<Student>()
{
    new Student() {Name="Shaimaa", Id=1,
PhoneNumber="123"},
    new Student(){Name="Amal",Id=2,
PhoneNumber="456"},
};
```



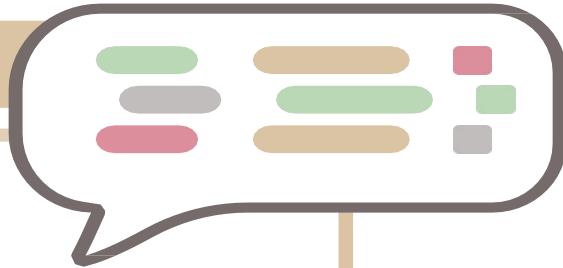
Exercise Solution

```
foreach (var item in studentList)
{
    Console.WriteLine(item.Id + "-" +
        item.Name);
}
studentList.Add(new Student() { Name = "Bayan", Id =
3, PhoneNumber = "789" });
foreach (var item in studentList)
{
    Console.WriteLine(item.Id + "-" + item.Name);
}
Console.WriteLine("*****");
studentList.Insert(1,new student() { Name =
"Raghad", Id = 4, PhoneNumber = "147" });
```



Exercise Solution

```
foreach (var item in studentList)
{
    Console.WriteLine(item.Id + " - " + item.Name);
}
Console.WriteLine("*****");
studentList.Remove(studentList[1]);
foreach (var item in studentList)
{
    Console.WriteLine(item.Id + " - " + item.Name);
}
Console.WriteLine("*****");
```



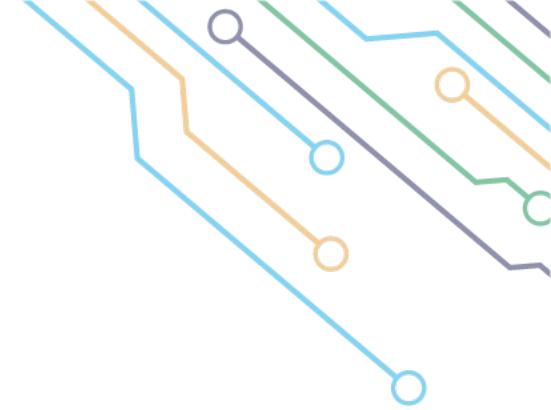




ArrayList in C#

ArrayList is a class used to create a dynamic array with changeable size. It is defined under System. Collection. namespace. Using ArrayList, programmers can store any datatype of elements in and there are many methods can be applied on as list.

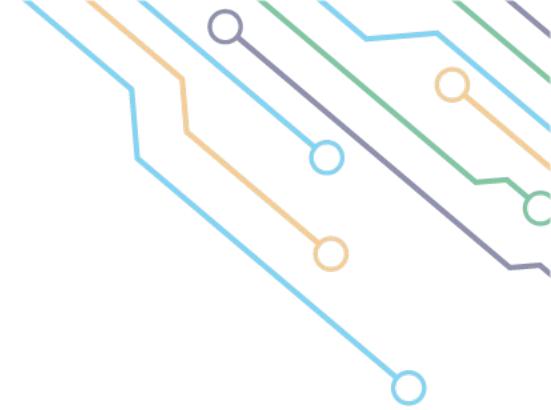




Points to distinguish

- The elements of ArrayList can be accessed by their indexes starting with zero.
- ArrayList has dynamic size which means programmers can change, add or delete, etc.. from list .
- It allows elements duplication with different indexes.
- It has many operation on like search and sort.





ArrayList Syntax

ArrayList is defined as the following:

- `ArrayList list_name = new ArrayList()`
- `ArrayList list_name = new ArrayList(initial Capacity)`



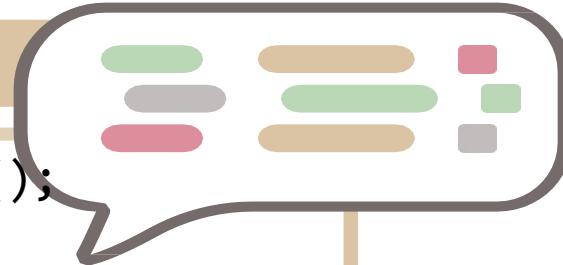


Let's Take an Example



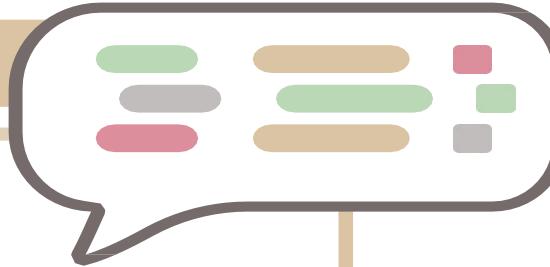
Define ArrayList

```
ArrayList array = new ArrayList();
array.Add(1);
array.Add("Shaimaa");
array.Add(true);
array.Add(new student());
foreach (var item in array)
{
    Console.WriteLine(item);
}
array.RemoveAt(3);
array.Remove(array[0]);
```



Define ArrayList

```
Console.WriteLine("*****");  
foreach (var item in array)  
{  
    Console.WriteLine(item);  
}
```







LINQ in C#

Language Integrated Query (LINQ) is constructed in C# as first class language just like other members such as methods, classes and others which provides a strong set of technologies in retrieving data from different data sources such as database.





LINQ in C#

To make the image clearer, let's back to the SQL usage in saving and retrieving data from database, the same for LINQ which is built in C# to retrieve data in different format from different resources such as databases,

In this chapter we will deal with local data source like collection, array or list.





LINQ Expression Syntax

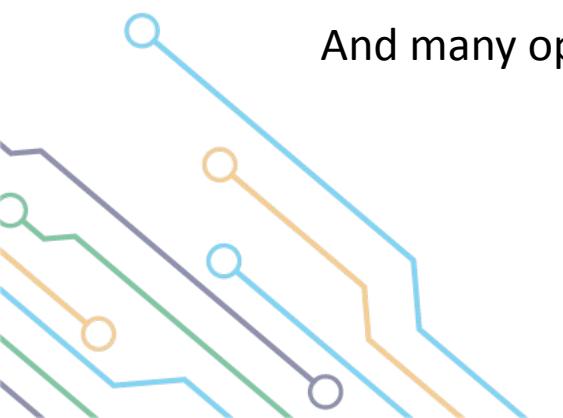
The standard Query will be as the following :

`from variable in Sequence`

`where condition`

`select variable;`

And many operators can be used with.





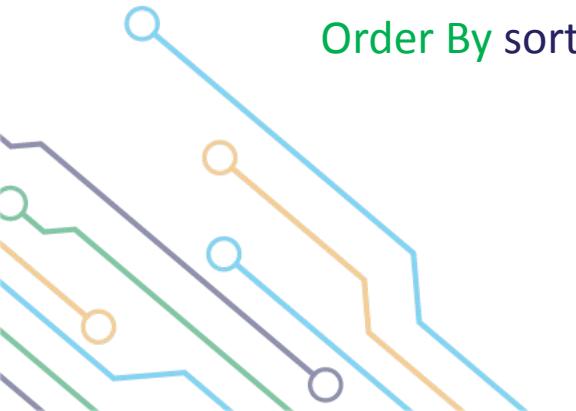
Standard Query Operators

Before continue let's look on the operators that are available to use :

Where filter operator returns elements based on condition

Select projection operator returns **IEnumerable** collection of elements based on body statement.

Order By sort operator returns elements in order Ascending or Descending.





Standard Query Operators

Group By grouping operator returns group of elements based on key value

Join joining operator returns the result of joining two sequences

Contains quantifier operator returns a Boolean if the data source contains the specified element





Standard Query Operators

Any quantifier operator returns Boolean value if any elements satisfy the specified condition.

All quantifier operator returns Boolean value if all elements satisfy the specified condition.

Take partitioning operator returns one of two parts after splitting the sequence.



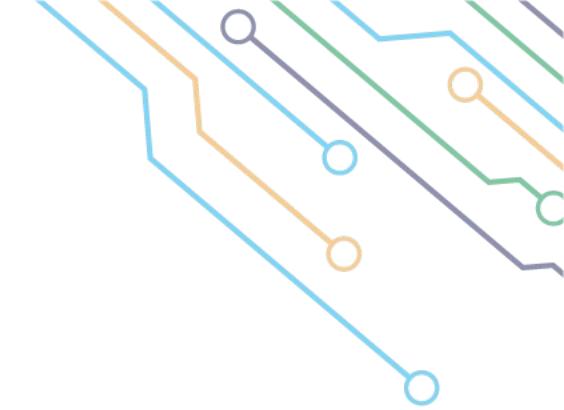


Standard Query Operators

In addition to aggregation operators that are used on numeric value in the collection or data source such as Min, Max, Sum, Average and Count.

Note that All, Any, Take are not supported in C# query syntax, and they will be covered in next chapter using lambda.





Let's Take an Example



Array of string

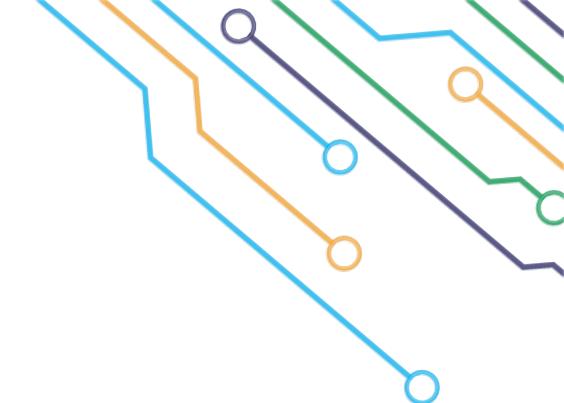
```
string[] names = { "Shaimaa", "Amal", "Bayan" };
var namesInArray=from name in names
                  where name=="Shaimaa"
                  select name;
Console.WriteLine("Students names are
:");
foreach (var item in namesInArray)
{
    Console.Write($" {item}");
}
Console.WriteLine();
```



Get Even and positive numbers in integer array

```
int[] numbers = { 1,2,3,5,88,-8,-20,9,-5 };
var integerInArray = from num in numbers
                    where num %2==0
                    where num>0
                    select num;
Console.WriteLine("Even Numbers are
:");
foreach (var item in integerInArray)
{
    Console.Write($" {item}");
}
Console.WriteLine(" _____");
```





Elements form list containing 'a' letter

From list of student class that created in previous exercise, display the names which contains 'a' letter using LINQ?



Exercise Solution

```
// List from Student Class
List<Student> students = new List<Student>()
{
    new
    Student(){Name="Shaimaa", Id=1, PhoneNumber="123" },
    new
    Student(){Name="Amal", Id=2, PhoneNumber="456" },
    new
    Student(){Name="Bayan", Id=3, PhoneNumber="789" }
};
var studentsNames = from name in students
where name.Name.Contains('a')
select name.Name;
```



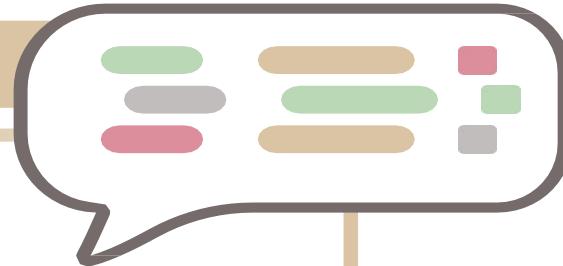
Exercise Solution

```
foreach (var item in studentsNames1)
{
    Console.WriteLine($"Students names
are : {item}");
}
Console.WriteLine("_____");
var orderedByIdDesc = from s in students
orderby s.Id descending
select s.Id + " " +s.Name;
```



Exercise Solution

```
foreach (var item in orderedById)
{
    Console.WriteLine($"Students names
ordered by Id are : {item}");
}
Console.WriteLine("_____");
// Now, Create Course class and make list from
class Course
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string TrainerName { get; set; }
}
```



Exercise Solution

```
// in main
List<Course> courses = new List<Course>()
{
    new
    Course(){Name="OOP",Id=1,TrainerName="Shaimaa" },
    new
    Course(){Name="DB",Id=2,TrainerName="Amal" },
    new
    Course(){Name="JS",Id=3,TrainerName="Bayan" },
    new
    Course(){Name="Web",Id=3,TrainerName="Shaimaa" },
    new
    Course(){Name="QA",Id=3,TrainerName="Bayan" }

};
```



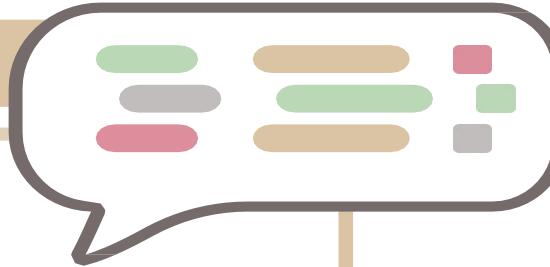
Exercise Solution

```
// Group courses by the name of trainer
var coursesPerTrainer=from course in courses
                      group course by
course.TrainerName into trainerName
                      select
trainerName;
                      foreach (var trainer in
coursesPerTrainer)
{
    Console.WriteLine($"courses of :
{trainer.Key}");
```



Exercise Solution

```
foreach (var course in trainer)
{
    Console.WriteLine($"{course.Name} {course}");
}
Console.WriteLine("_____");
```



Exercise Solution

```
// in student class
public int courseId { get; set; }
// in main update the list by adding course id
List< Student > students = new List<Student>()
{
    new
    Student(){Name="Shaimaa",Id=1,PhoneNumber="123",courseId=1},
    new
    Student(){Name="Amal",Id=2,PhoneNumber="456",
    courseId=2},
    new
    Student(){Name="Bayan",Id=3,PhoneNumber="789",course
    Id=3 }
};
```



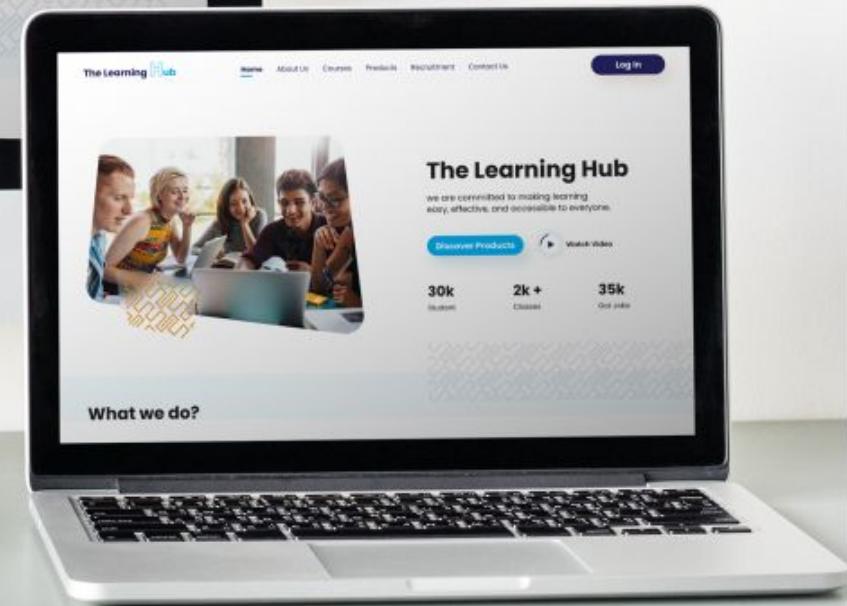
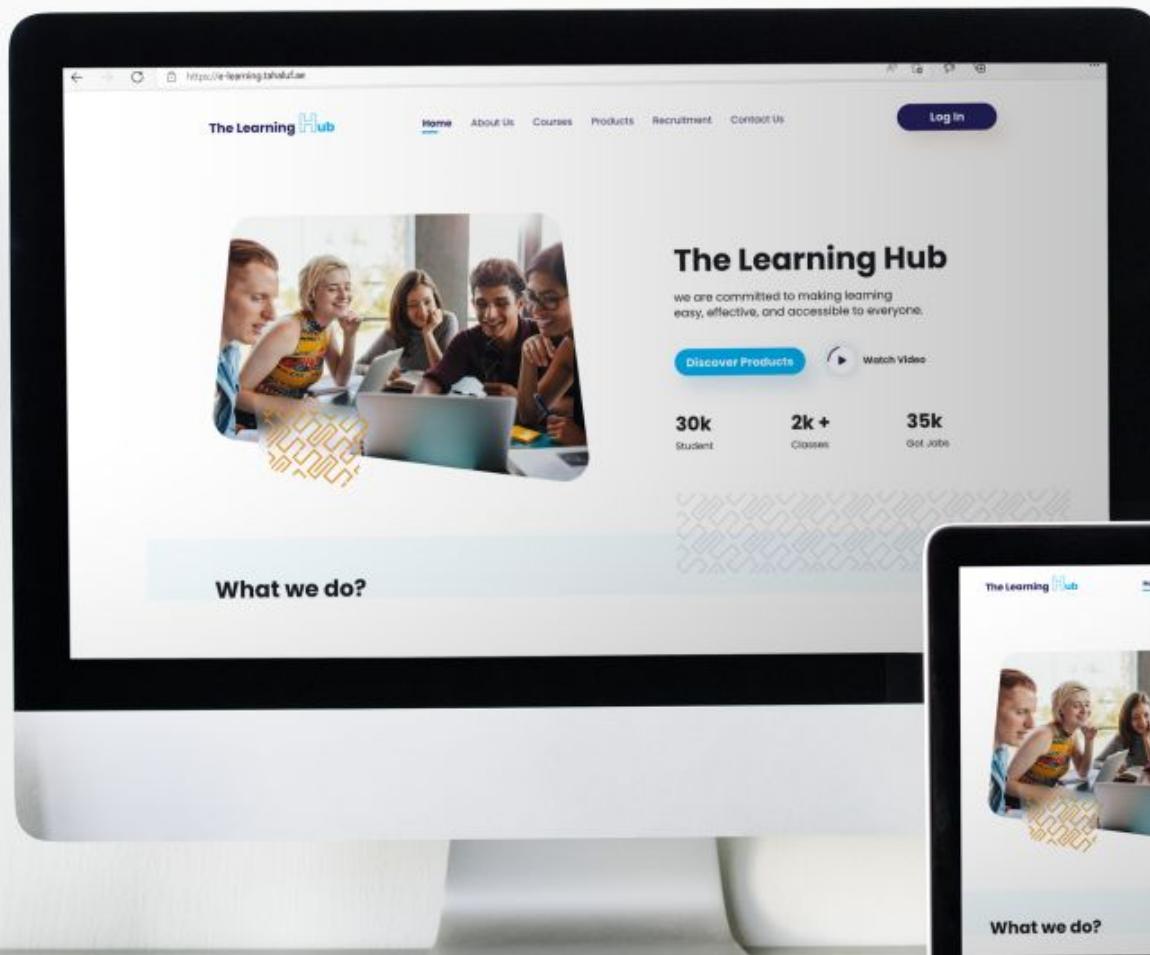
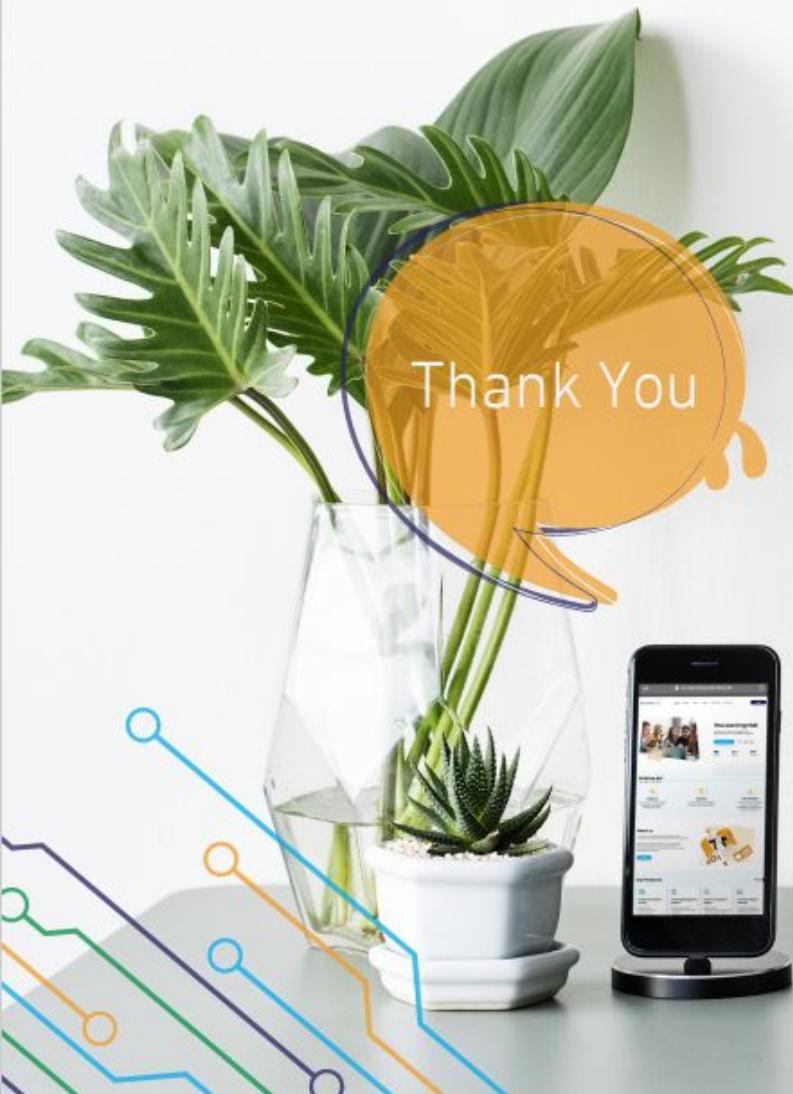
Exercise Solution

```
// Join Student and Course classes on course Id
var studentCourses=from student in students// outer
list
join course in courses// inner list
on student.courseId equals course.Id// key
select new { StudentName = student.Name,// result
CourseName=course.Name};
foreach (var item in studentCourses)
{
    Console.WriteLine($"  

{item.StudentName} attend {item.CourseName}  

course");
}
```



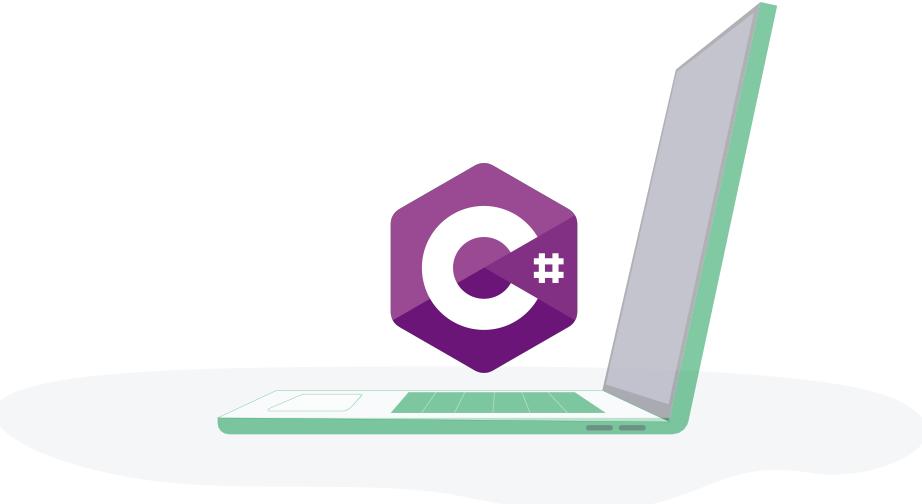


C# Object Oriented programming (OOP)

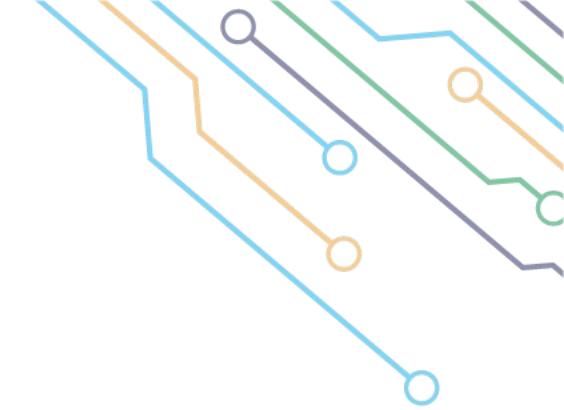
Education and Training Solutions 2022



- 1 Lambda Expression
- 2 Element Operators in LINQ
- 3 First and FirstOrDefault Methods
- 4 Single and SingleOrDefault Methods
- 5 Asynchronous Programming



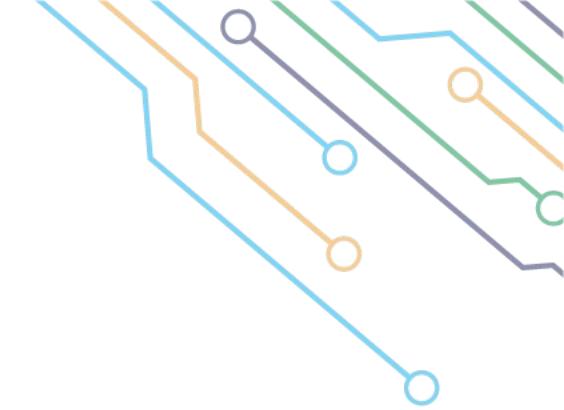




Lambda Expression

C# provides lambda expression to be used along with LINQ, so for the same purposes programmers use it ; to retrieve data in different formats from different resources such as database. It is considered a shorter way to represent the anonymous methods that use delegate keyword.





Points to distinguish

- Using lambda expression, programmers can use one parameter or multiple ones.
- Using lambda expression, programmers can use one statement or more in the body using curly braces .





Lambda with single parameter Syntax

Lambda expression as mentioned in the previous slide has progressed from anonymous method as the following:

- delegate (Student x){ return x.Id > 0 && x.Id < 10;};
- x => x.Id > 0 && x.Id < 10 ;

Whereas **x** is the parameter or input, **=>** is the lambda operator and **x.Id>0 && x.Id < 10** acts as the body of lambda expression.



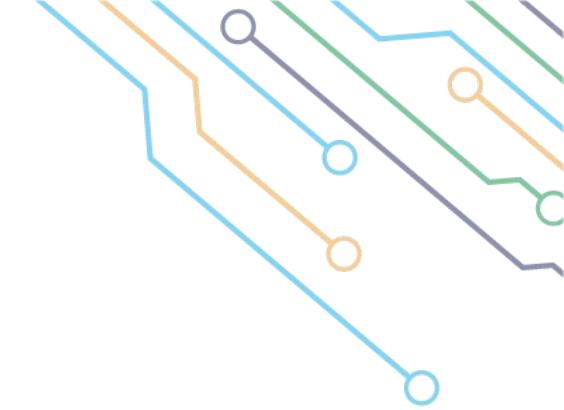


Lambda with multiple parameters Syntax

The same programmers can use lambda expression if they want to pass more than one parameter using parenthesis around them:

- `(x , y) => x.Id > y;`





Lambda with bunch of statements Syntax

Programmers can use lambda expression with bunch of statements in its body as the following:

- `(x) => { Console.WriteLine(x); return x; }`





Take benefits of

Many methods can be applied on lists and their queries.

`ForEach()` to perform the determined operations on all elements.

`FindAll()` returns all elements in data source that satisfy the condition.

Any, All and take operators in LINQ will be used in Lambda syntax as mentioned in previous chapter.



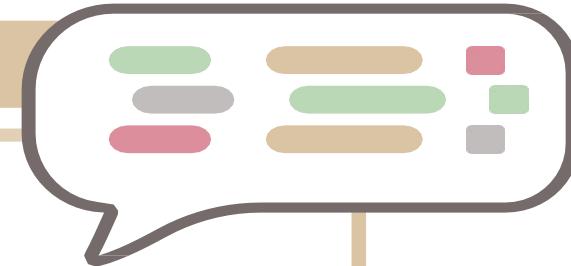


Let's Take an Example



Define List from Student Class

```
class Student
{
    public string Name { get; set; }
    public int Id { get; set; }
    public string PhoneNumber { get; set; }
}
// in main
List<Student> students = new List<Student>()
{
    new Student() {Name="Shaimaa",Id=1,
    PhoneNumber="123" },
    new Student() {Name="Amal",Id=2, PhoneNumber="456"
    },
    new Student(){Name="Bayan",Id=3, PhoneNumber="789" }
};
```



Continue..

```
var name=students.FindAll(x => x.Name == "Shaimaa");
foreach (var item in name)
{
    Console.WriteLine($"Element with name {item.Name}
and Id {item.Id}");
}
students.FindAll(x => x.Id >= 2).ForEach(x =>
Console.WriteLine(x.Id));
    Console.WriteLine("-----");
    var count = students.FindAll(x => x.Name
== "Shaimaa").Count;
    Console.WriteLine($"# of element with
name Shaimaa is {count}");
```



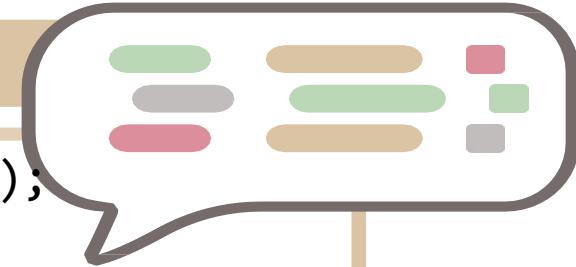
Continue..

```
Console.WriteLine("-----");
    bool id=students.All(x => x.Id >= 1);
    Console.WriteLine($"IS all IDs more than
1? {id}");
    Console.WriteLine("-----");
    bool name3 = students.Any(x =>
x.Name=="Shaimaa");
    Console.WriteLine($"IS any of element
has Shaimaa as a name? {name3}");
    Console.WriteLine("-----");
    bool y = students.Select(x =>
x.Name).Contains("Shaimaa");
    Console.WriteLine($"Does lit contain
Shaimaa{y}");
    Console.WriteLine("-----");
```



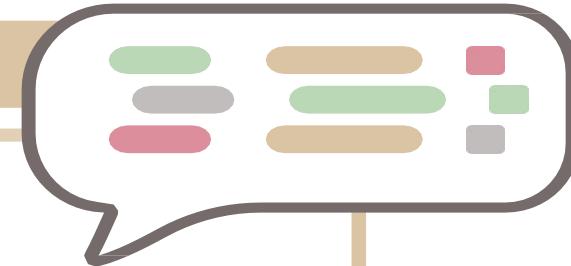
Continue..

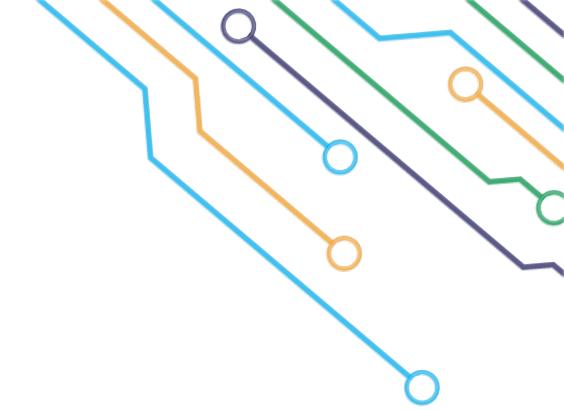
```
var name2 = students.Select(x => x.Name);
    foreach (var item in name2)
    {
        Console.WriteLine($"Names using
Select operator {item}");
    }
    Console.WriteLine("-----");
    var orderNameDesc=
students.OrderByDescending(x=>x.Id).Select(x=>x.Name
).ToList();
    foreach (var item in orderNameDesc)
    {
        Console.WriteLine($"Names Descending
by Id {item}");
    }
```



Continue..

```
Console.WriteLine("-----");
    students.Select(x => x.Id *
2).Select((x) => { Console.WriteLine($"double Id
{x}");return x; }).ToList();
    Console.WriteLine("-----");
var maxID = students.OrderByDescending(x =>
x.Id).Take(1);
    foreach (var item in maxID)
{
    Console.WriteLine($"Names Descending
by Id {item.Id}");
}
Console.WriteLine("-----");
Console.WriteLine(students.Max(x => x.Id));
Console.WriteLine("-----");
```





By using lambda

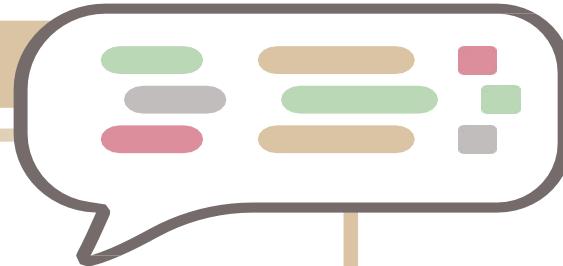
Create list to display 10 numbers, calculate the square of each one
and finally find the ones the divisible by 3?



Exercise Solution

```
List<int> numbers = new List<int>()
{
    2,4,6,8,44,33,3,9,45,10
};
Console.WriteLine("The numbers in list are
:");
foreach (var item in numbers)
{
    Console.Write(" " + item);
}
Console.WriteLine();

Console.WriteLine("_____");
```



Exercise Solution

```
var squares = numbers.Select(x => x * x);
Console.WriteLine("The square of numbers in
list is : ");
foreach (var item in squares)
{
    Console.Write(" " +item);
}
Console.WriteLine();

Console.WriteLine(" _____");
Console.WriteLine("Numbers are divisible by
3 are");
numbers.FindAll(x => x % 3 ==
0).ForEach(x =>{ Console.Write(" ");
Console.Write(x); });
```





Element Operators in LINQ



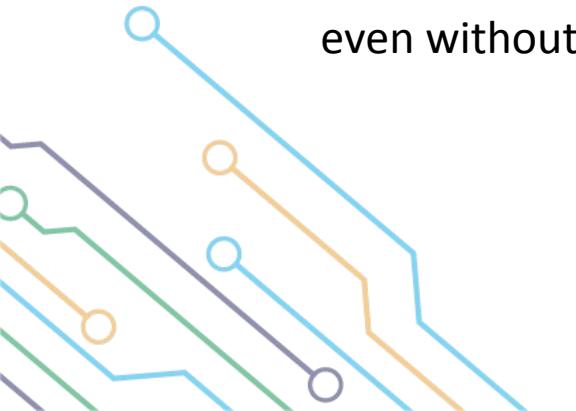


Element Operators

Programmers deal with element operators which are defined under System. LINQ when they want to return a particular elements from a collection.

There are many of them such as first , last , single and many other.

Note that a condition can be specified on these operators using lambda expression or even without.





Let's Take an Example



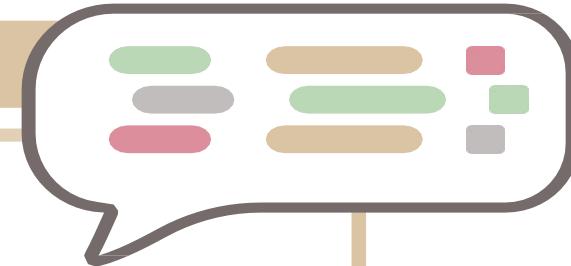
Add these Lists to work on

```
List<int> list0 = new List<int>() { 7 };

List<int> list1 = new List<int>() { 7,1,2,3,4,5,6 };

List<string> list2 = new List<string>() {null,
"Shaimaa","Amal","Bayan"};

List<string> list3 = new List<string>() { };
```





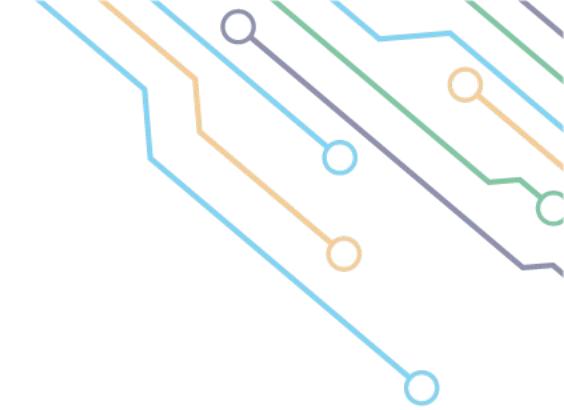


First()

Returns the first element or the first element satisfies the condition if there any in the collection.

Throw `InvalidOperationException` if there is no elements in the collection or no elements satisfy the added condition.

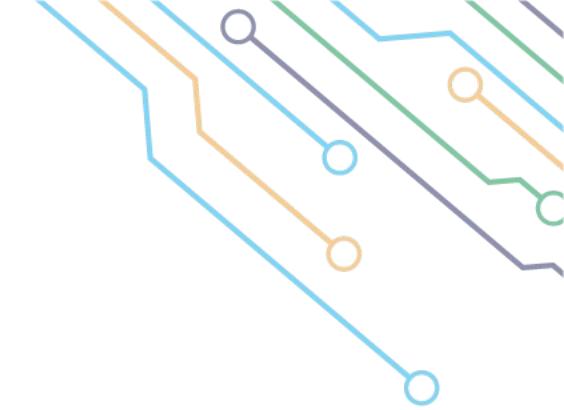




FirstOrDefault()

Returns the first element or the first element satisfies the condition in the collection or the default of the collection datatype in cases there is no elements in the collection, or no element satisfies the condition.



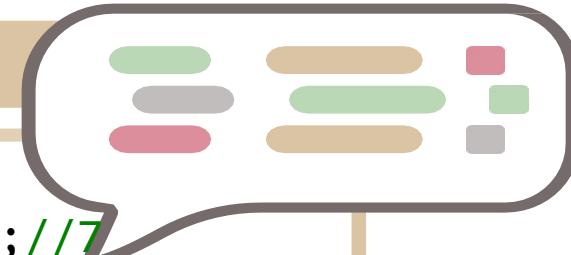


Let's Take an Example



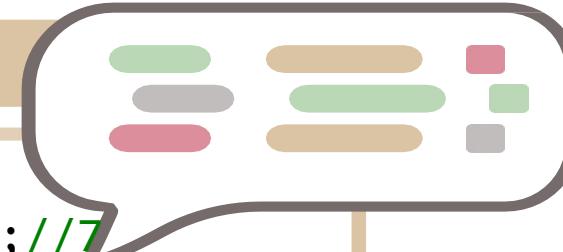
First() & FirstOrDefault()

```
//list0
Console.WriteLine($"0- {list0.First()}");//7
Console.WriteLine($"0- {list0.First(x => x ==
7)}");//7
//Console.WriteLine($"0- {list0.First(x=>x==4)}");// exception: no element
Console.WriteLine($"0-
{list0.FirstOrDefault()}");//7
Console.WriteLine($"0- {list0.FirstOrDefault(x => x
== 7)}");//7
Console.WriteLine($"0- {list0.FirstOrDefault(x => x
== 4)}");// default value=0
```



First() & FirstOrDefault()

```
//list1
Console.WriteLine($"1- {list1.First()}"); //7
Console.WriteLine($"1- {list1.First(x => x ==
5)}"); //5
Console.WriteLine($"1- {list1.First(x=>x>4)}"); // 7
Console.WriteLine($"1-
{list1.FirstOrDefault()}"); //7
Console.WriteLine($"1- {list1.FirstOrDefault(x => x
== 4)}"); //4
Console.WriteLine($"1- {list1.FirstOrDefault(x => x
> 4)}"); // 7
Console.WriteLine($"1- {list1.FirstOrDefault(x => x
> 8)}"); // default value=0: no element
```



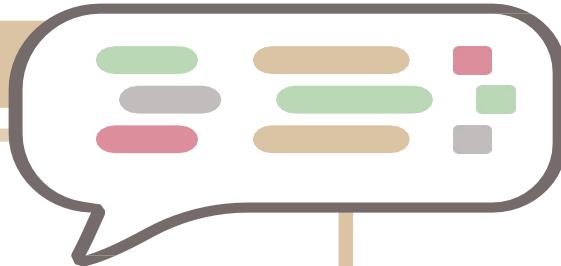
First() & FirstOrDefault()

```
//list2
Console.WriteLine($"2- {list2.First(x => x == null)}");//null
Console.WriteLine($"2- {list2.First()}");//null
Console.WriteLine($"2-
{list2.FirstOrDefault()}");//null
Console.WriteLine(list2.First(x=>x=="Shaimaa"));//sh
aimaa
Console.WriteLine(list2.FirstOrDefault(x => x ==
"Shaimaa"));//shaimaa
//Console.WriteLine($"2- {list2.First(x =>
x.Contains("S"))}");//exception: starts with null
//Console.WriteLine($"2- {list2.First(x =>
x.Contains("A"))}");//exception: starts with null
```



First() & FirstOrDefault()

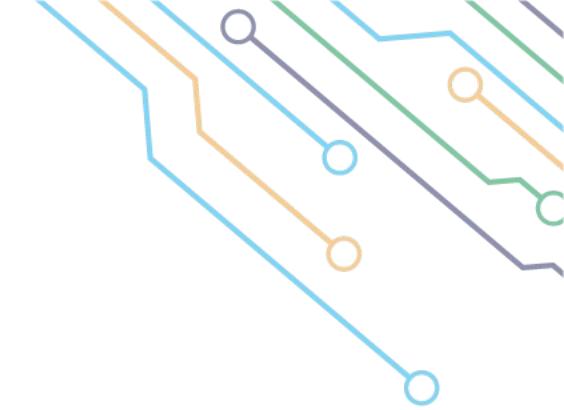
```
//list3
//Console.WriteLine($"3-
{list3.First()}");//exception: no element
//Console.WriteLine($"3- {list3.First(x => x ==
“Raghad”)}");//exception: no element
Console.WriteLine($"3-
{list3.FirstOrDefault()}");//default value= null: no
element
Console.WriteLine($"3- {list3.FirstOrDefault(x => x
== “Raghad”)}");//default value= null: no element
```





Single & SingleOrDefault





Single()

Returns the only element or the only element satisfies the in the collection, it always expects one element.

Throw `InvalidOperationException` if there is more than one element in the collection or more than one element satisfy the added condition.

And the same if there is no elements or no element satisfy the condition.





SingleOrDefault()

Returns the only element or the only element satisfies the condition in the collection **or** the default of the collection datatype in case there is no elements in the collection, or no element satisfies the condition.

Throw `InvalidOperationException` if there is more than one element in collection or more than one element satisfy the added condition.





Special Case

While evaluating a condition, if the collection has null before the element that satisfy the condition, the two element operators will **Throw NullReferenceException**.



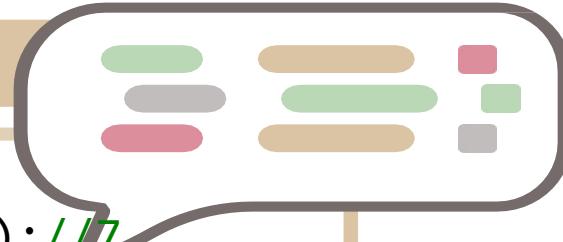


Let's Take an Example



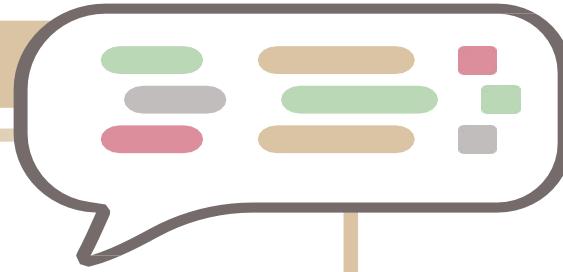
Single() & SingleOrDefault()

```
//list0
Console.WriteLine($"0- {list0.Single()}"); //7
Console.WriteLine($"0- {list0.Single(x => x ==
7)}"); //7
//Console.WriteLine($"0-
{list0.Single(x=>x==4)}"); // exception : no element
Console.WriteLine($"0-
{list0.SingleOrDefault()}"); //7
Console.WriteLine($"0- {list0.SingleOrDefault(x => x
== 7)}"); //7
Console.WriteLine($"0- {list0.SingleOrDefault(x => x
== 4)}"); // default value=0 : no element
```



Single() & SingleOrDefault()

```
//list1
//Console.WriteLine($"1- {list1.Single()}");//exception
Console.WriteLine($"1- {list1.Single(x => x == 7)}");//7
//Console.WriteLine($"1- {list1.Single(x=>x>4)}");// exception : more than one element
//Console.WriteLine($"1- {list1.SingleOrDefault()}");//exception:more than one element
Console.WriteLine($"1- {list1.SingleOrDefault(x => x == 7)}");//7
Console.WriteLine($"1- {list1.SingleOrDefault(x => x > 4)}");// exception more than one elament
Console.WriteLine($"1- {list1.SingleOrDefault(x => x > 8)}");// default value=0 :no element
```



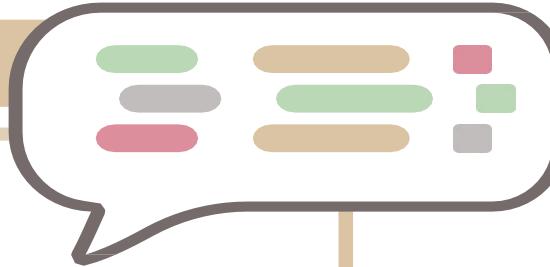
Single() & SingleOrDefault()

```
//list2
Console.WriteLine($"2- {list2.Single(x => x ==
null)}"); //null
//Console.WriteLine($"2- {list2.Single(x => x == " "
")}); //exception : no element
//Console.WriteLine(list2.Single(s =>
s.Contains("A"))); //exception: has null
//Console.WriteLine(list2.SingleOrDefault(s=>s.Conta
ins("A"))); //exception: has null
```



Single() & SingleOrDefault()

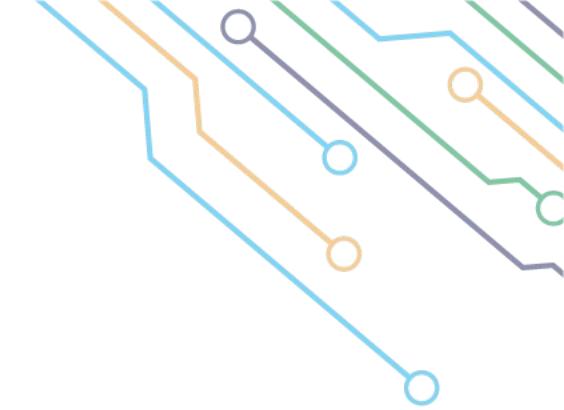
```
//list3
//Console.WriteLine($"3-
{list3.Single()}"); //exception: no element
Console.WriteLine($"3-
{list3.SingleOrDefault()}"); //default value null :no
element
Console.WriteLine($"3- {list3.SingleOrDefault(x => x
== "Raghad")}" ); //default value null: no element
```





Asynchronous Programming





Asynchronous Programming

In large projects, imagine the time is needed to fetch or submit data from or to external resources such as database or even files. Can the delay and lag that will occur be accepted while calling these functions? So, to avoid loading and lagging while pages are requested, asynchronous programming comes to solve this issue.





Asynchronous Programming

Let's make the idea clearer and more understood, asynchronous programming allows users to keep going with their business in applications while other processes are work on background to get some requested responses which overall enhance the user experience and raise the application quality and performance.

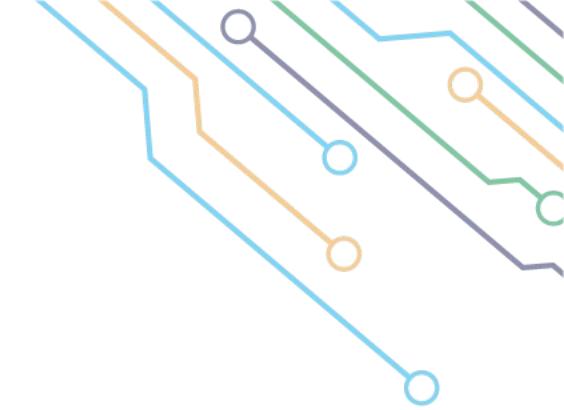




Asynchronous Programming

Programmers can achieve this by dividing process into multiple threads or multiple operations to be executed asynchronously. But before starting, let's know more about the keywords that are used so the methods can be considered as async ones.

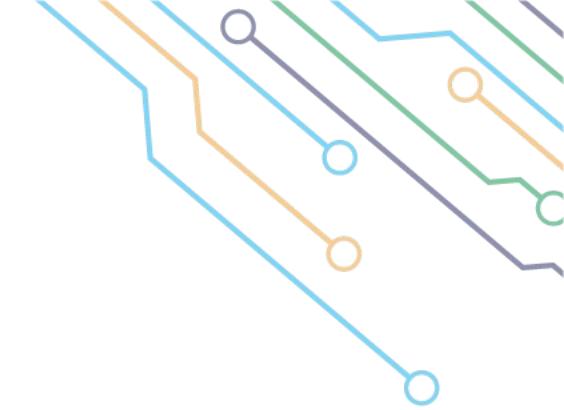




Task Data Type

Programmers use **Task** data type to represent that the operation or method will be executed asynchronously and there is no need to complete its functionality immediately. To be clearly known, Tasks is executed by task scheduler in .Net framework which will request a thread from thread pool which by its turn will create new thread or reuse an existing one to execute this operation.





Task Data Type

Using `Task` means that the method has no return type just like void methods in synchronous programming.

Using `Task<T>` means that the method has return type of T which can be string, int, double, etc....





Task Data Type

Programmers usually use `async` and `await` keywords with asynchronous programming.

Using `async` means that the method is asynchronous.

Using `await` to prevent blocking the current thread by running operation to complete its execution, instead of that it will be free to be used in another tasks.



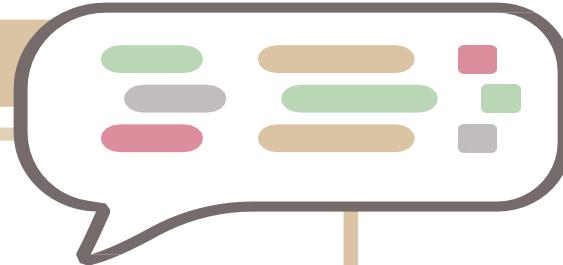


Let's Take an Example



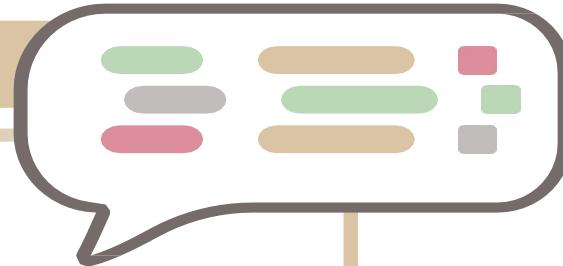
Create sync and async methods and notice the delay in both

```
public static void SyncMethod()
{
    Console.WriteLine("I am a synchronous
method");
    Thread.Sleep(5000); // synchronous method
    to make a delay by 5 sec
}
public static void SyncMethod2()
{
    Console.WriteLine("I am a synchronous
method 2");
}
```



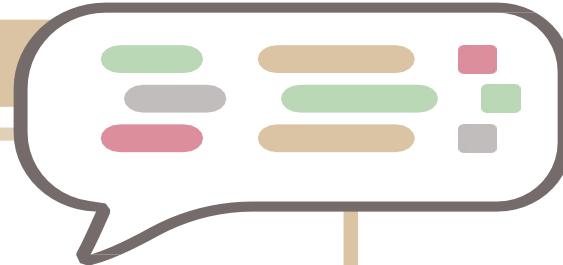
Continue..

```
public static async Task AsyncMethod()
{
    Console.WriteLine("I am an Asynchronous
method");
    await Task.Delay(5000); // asynchronous
method to make a delay by 5 sec
}
//call them in main method
AsyncMethod();
SyncMethod();
SyncMethod2();
```



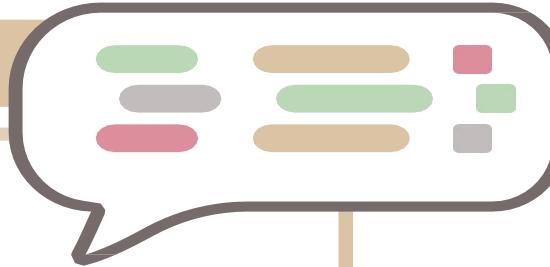
We can use await Task.Run() to run the operation in thread pool

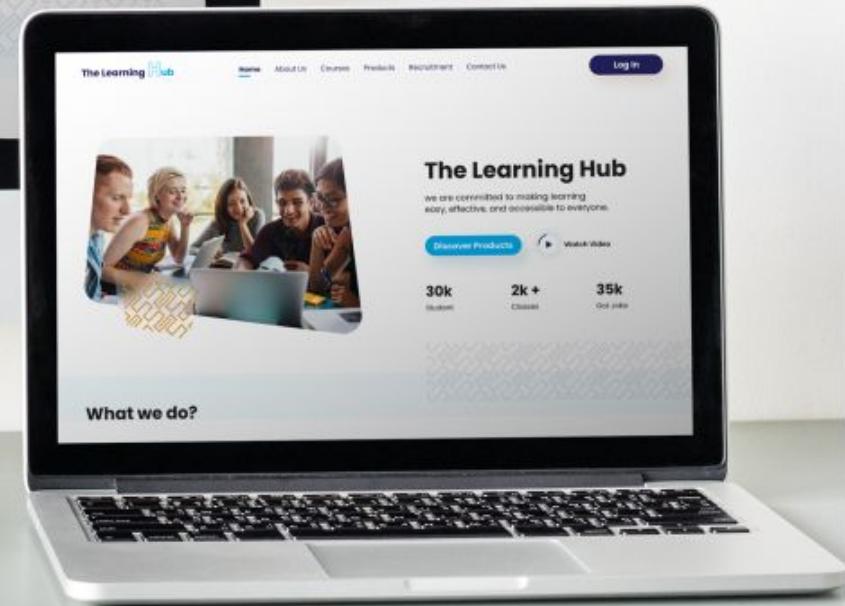
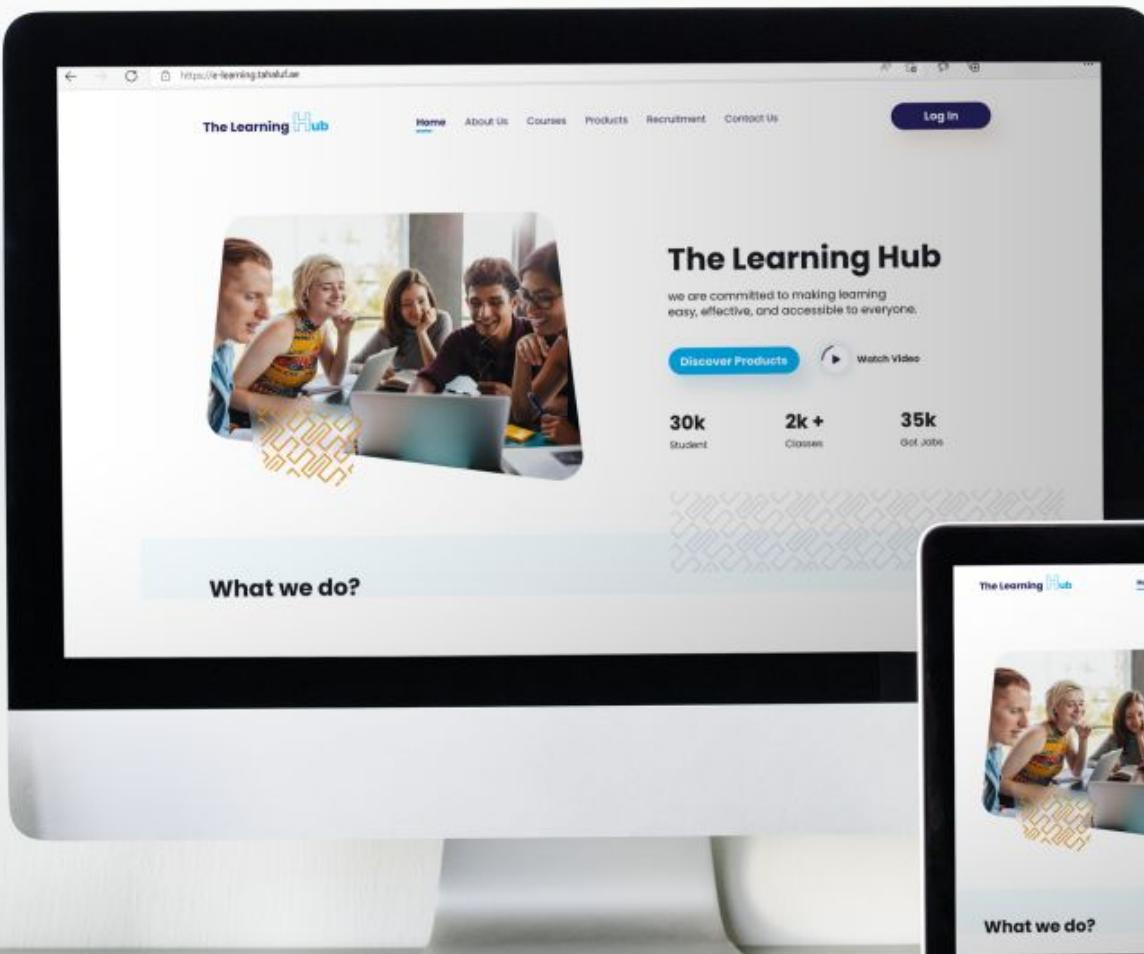
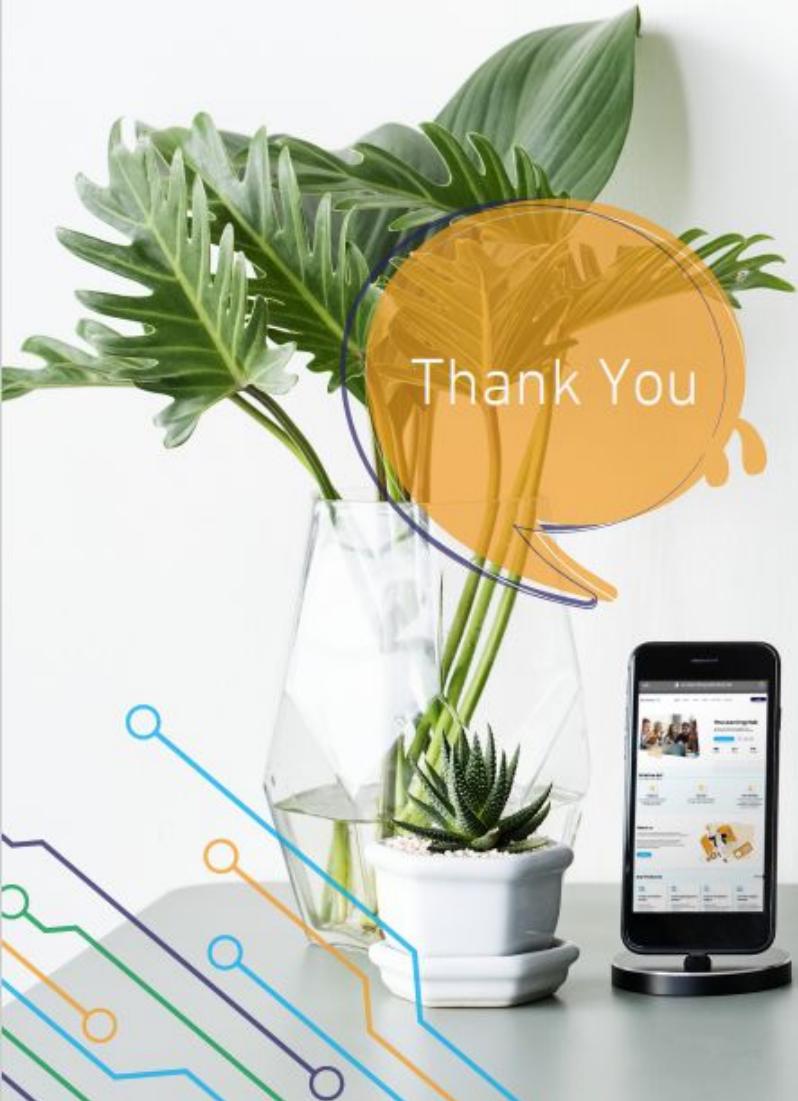
```
public async static Task Method1()
{
    await Task.Run(() =>
    {
        for (int i = 0; i < 25; i++)
        {
            Console.WriteLine("Don't wait
me");
            Task.Delay(1000).Wait();
        }
    });
}
```



Continue..

```
public static void Method2()
{
    for (int i = 0; i < 25; i++)
    {
        Console.WriteLine("I am not
waiting");
        Thread.Sleep(1000);
    }
}
//in main
Method1();
Method2();
```





- 1 C# FileInfo
- 2 StreamWriter in C#
- 3 StreamReader in C#



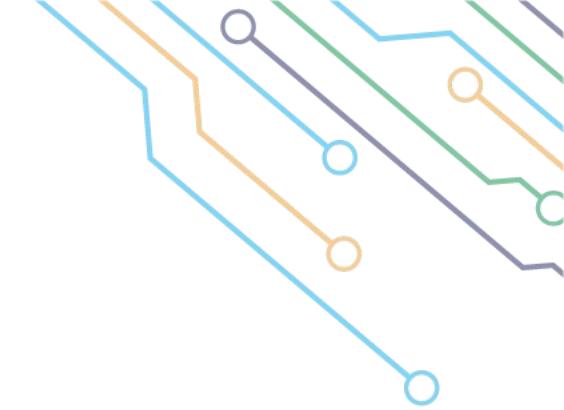




FileInfo

- The FileInfo is a sealed class provides control on read/write operations on files by writing code manually for reading or writing bytes to a file.
- To work with files in .NET framework, the important namespace used is system.IO namespace and similarly.



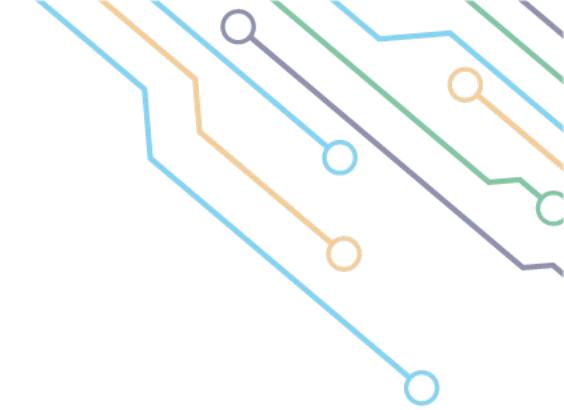


FileInfo

FileInfo class uses the following properties:

- **FileInfo(string):** A new object of the FileInfo class is initialized and it acts as a wrapper for the path of the file.
- **Attributes:** We can get or set the attributes for the current directory or current file using Attributes property.





FileInfo

- **CreationTime:** We can get or set the creation time for the current file or the current directory using Creation Time property.
- **Directory:** We can get an instance of the parent directory using Directory property.





FileInfo

- DirectoryName:** returns a string that represents the full path of the directory.
- Exists:** this property let you know if a file exists by returning true or non exist by returning false.

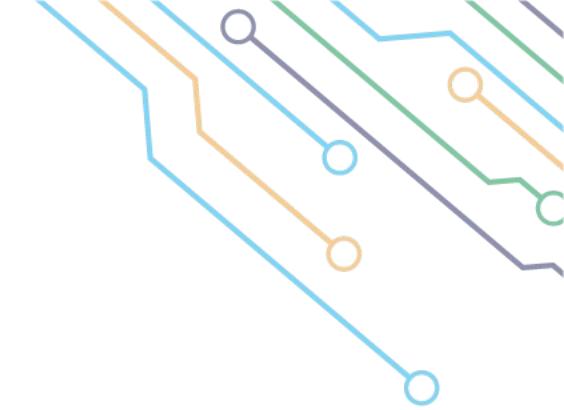




FileInfo

- FullName:** through this property you can get the full path of the directory or the full path of the file.
- IsReadOnly:** a property that returns true if the file is read only, otherwise it return false.





FileInfo

- LastAccessTime:** property returns the latest access time for the current time.
- Length:** property to get the size of the current file in bytes.
- Name:** property to get the name of the file.

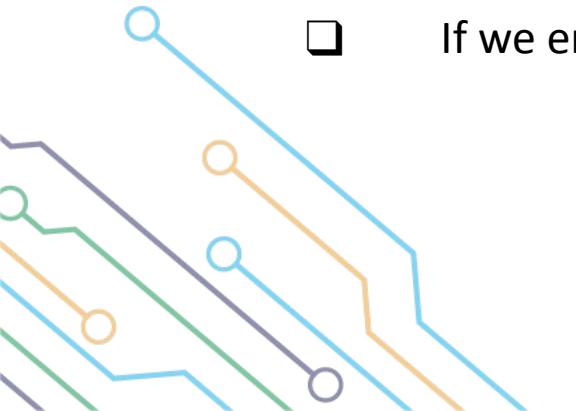




Exception handling (Example)

A program that divides 2 numbers;

- If the divisor was 0 the program will terminate and **DivideByZeroException** exception will thrown.
- If we enter string instead of int or double **FormatException** will thrown.
- If we enter too large or too small number, **OverflowException** will thrown.



Use FileInfo properties:

```
FileInfo file = new  
FileInfo("C:\\Work\\OOP\\Trainees.txt");  
Console.WriteLine(file.Name);  
Console.WriteLine("-----");  
Console.WriteLine(file.FullName);  
Console.WriteLine("-----");  
  
Console.WriteLine(file.Length);  
Console.WriteLine("-----");  
Console.WriteLine(file.Directory);  
Console.WriteLine("-----");  
Console.WriteLine(file.DirectoryName);  
Console.WriteLine("-----");
```

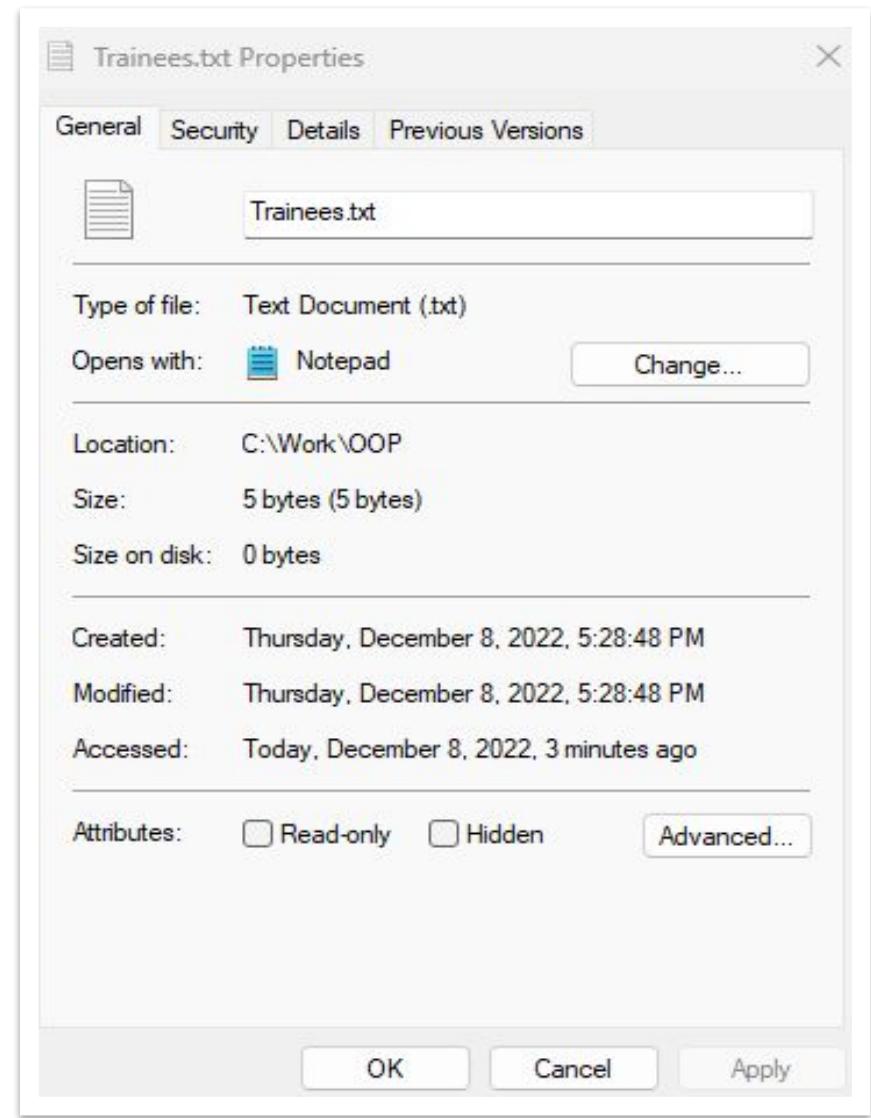


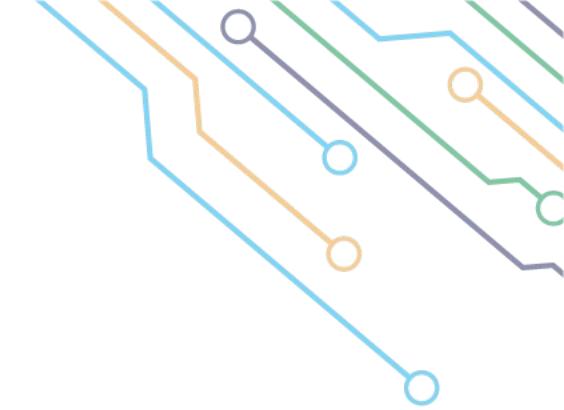
Use FileInfo properties:

```
Console.WriteLine(file.IsReadOnly);
    Console.WriteLine("-----");
    Console.WriteLine(file.Attributes);
    Console.WriteLine("-----");

Console.WriteLine(file.Exists);
    Console.WriteLine("-----");
    Console.WriteLine(file.CreationTime);
```







FileInfo methods

FileInfo class uses the following methods:

- Create():** this method creates a file in specific location.
- Delete():** this method deletes a specific file.

Let's write their codes...



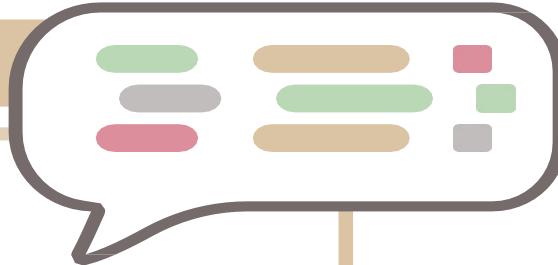
Create a file using FileInfo class:

```
try
{
    // the file location is specified
    where the file is to be created
    string location =
        @"C:\Users\User\Desktop\Tahaluf\OOP\Day 7\new.txt";
        // instance of the fileInfo class is
        created
        FileInfo file = new
        FileInfo(location);
```



Create a file using FileInfo class (continued..)

```
// an empty file will be created
    file.Create();
    Console.WriteLine("Creation of file
is successfull");
}
catch (IOException e)
{
    Console.WriteLine("Failed attempt to
create file " + e);
}
```

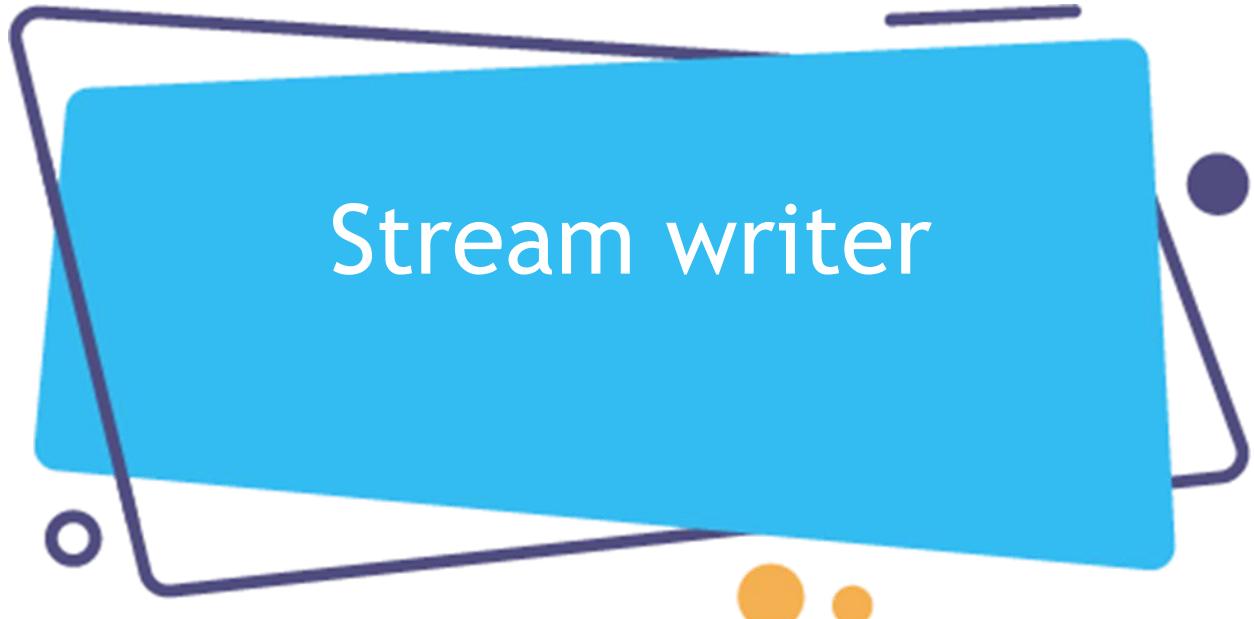


Delete a file using FileInfo class:

```
// the file location is specified where the file is
// to be located
string location =
@"C:\Users\User\Desktop\Tahaluf\OOP\Day 7\new.txt";
// object from the fileInfo class is
created
FileInfo file = new FileInfo(location);

// The specified file will be deleted
file.Delete();
Console.WriteLine("successfully
deleted");
```







FileInfo methods

FileInfo class uses the following methods:

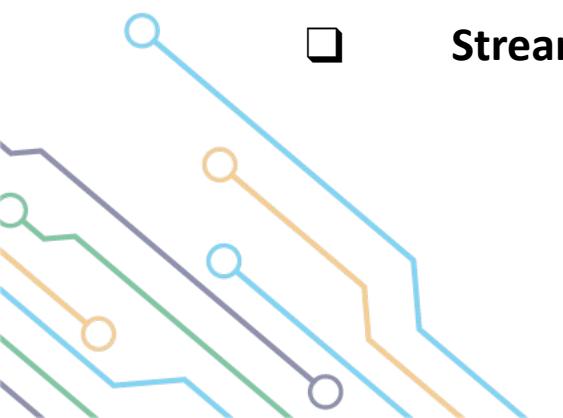
- Create():** this method creates a file in specific location.
- Delete():** this method deletes a specific file.





Stream writer

- **StreamWriter** is class in C# that uses specified encoding to write characters to a stream.
- **TextWriter** class is parent class for StreamWriter class it provides methods to write strings to a file.
- **StreamWriter.Write()** is method is responsible for writing text to a stream.





Stream writer

StreamWriter is defined in the System.IO namespace ,provides the following Write methods:

- Write** Writes data to the stream.
- WriteAsync** Writes data to the stream asynchronously.





Stream writer

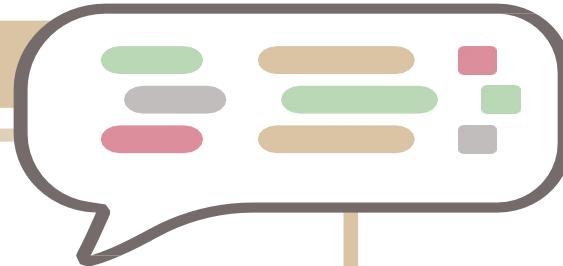
- **WriteLine** □ Writes text string or stream with a line terminator.
- **WriteLineAsync** □ Writes text string or stream with a line terminator asynchronously.

Let's write their codes □



Use StreamWriter class to write on file

```
String filePath = @"C:\Work\OOP\Trainee.txt";  
  
using (StreamWriter stream =  
File.AppendText(filePath))  
{  
    stream.WriteLine("Welcome to The learning hub");  
    stream.Close();  
    Console.WriteLine(File.ReadAllText(filePath));  
}
```



Create a program in C# to request lines from the user and write them in a text file.



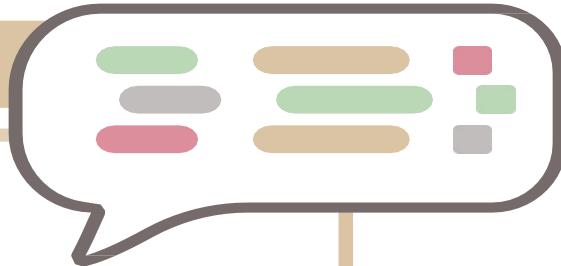
Exercise Solution

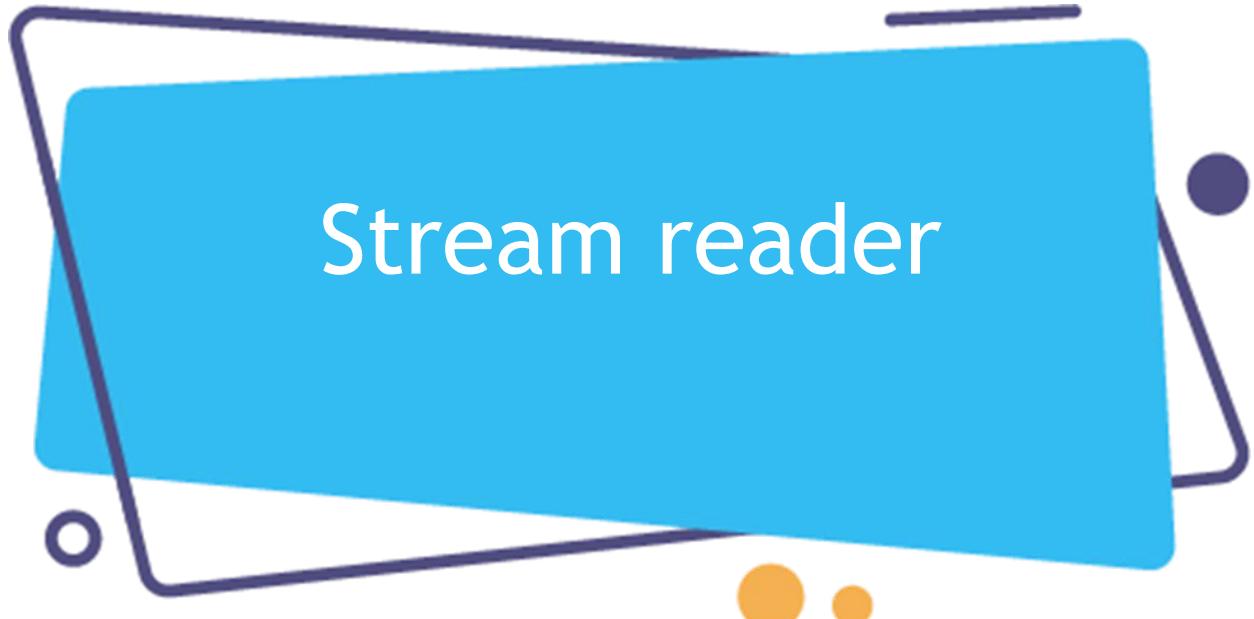
```
string fileName = "Ex.txt";

using (StreamWriter file= File.CreateText(fileName))
{
    string line;
    do
    {
        line = Console.ReadLine();

        if (line.Length != 0)
        {
            file.WriteLine(line);
        }
    }
    while (line.Length != 0);}

```







Stream reader

- **StreamReader** use specified encoding to read characters to a stream.
- **StreamReader.Read()** method reads the next set of characters from the input stream.
- **TextReader** is the parent class for **StreamReader**, it provides methods to read a character, block, line, or all content of the file.



Use StreamReader class to read a file:

```
try
{
    using (StreamReader read = new
StreamReader(@"C:\Work\OOP\Trainee.txt"))
    {
        string line1;
        while ((line1 = read.ReadLine()) != null)
        {
            Console.WriteLine(line1);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

