IsinRange method

```
Public Static bool IsinRange
  (this T value, T min,  T max)
 where  T :  IComparable<T>
{
    return ( min.CompareTo(value)<=0 )
           &&
          (value.CompareTo(max) <=0 );
}
```

C# Test

- The System.SystemException class is the base class for all predefined system exceptions in C# ?

- C# Does not support multiple inheritance.

- Value types directly contain data.
- Int, Char, & Float, which stores number, alphabet, & floating point numbers, are value types
- When you declare an int type, the system allocates memory to store the value.

- The assignment cannot be overloaded.

- Which of the following converts a type to a unsigned big type in C# ?

  ToUInt64 - Converts a type to an unsigned big type.

- We can use reserved keywords as identifiers in C# by prefixing them with @ character ?

- C# Structures
  - Structures can have methods, fields, indexers, properties, Operator methods, & events.

- Structures can have defined constructors, but not destructors.

- You cannot define a default constructor for a structure.

- & operator returns the address of an variable.

- default access specifier of a class member function?
  - private

- Converts a type to a signed byte type in C#?
  - ToSbyte

- #elif - It allows creating a compound conditional directive

- Converts a type to 32-bit integer in C#? - ToInt32.

- access specifier in C# allows a class to expose its member variables & member functions to other functions & objects?
  - public

- converts a type to a Boolean value, in C#
  - TO Boolean.

- C# is a modern, general-purpose, object-oriented programming language developed by Microsoft.

- C# was developed by Anders Hejlsberg & his team during the development of .Net framework

- C# is designed for common language infrastructure. (CLI)

Operator returns the type of a class in c# typeof.

access specifier in C# allows a class to expose its member variables & member functions to other Functions & objects in current assembly? - internal.

A name must be begin with a letter that could be followed by a sequence of letters, digits (0-9), or underscore.

The first character in an identifier cannot be digit.

Encapsulation is defined as the process of enclosing one or more items within physical or logical package.

Encapsulation, in object oriented programming, methodology, prevents access to implementation details.

Abstraction allows making relevant information visible & Encapsulation enables a programmer to implement the desired level of abstraction.

C# provides a special data types, the nullable types, to which you can assign normal range of values as well as null values.

- You can Assign true, false, or null in a Nullable<bool> Variable

- you can store any value from -2,147,483,648 to 2,147,483,647 or null in a Nullable<int32> Variable.

- Which of the following converts a floating point or integer type to a decimal type in C#?
  - To Decimal

- Property of Array class in C# gets 32-bit integer, the total number of elements in all the dimensions of the array?
  - Length

- access specifier in C# allows a child class to access the member variables & member functions of its base class?
  - Protected

- about static member functions of a class?
  - You can also declare a member functions as static.
  - such functions can access only static variables.
  - The static functions exist even before object is created.

- C# - It is component oriented.
  - It can be compiled on a variety of computer platforms.
  - part of .NET framework

- destructor is a special member function of a class that is executed whenever an object of its class goes out of scope.

- A destructor has exactly the same name as that of the class with a prefixed tilde (~) & it can neither return a value nor can it take any parameters

C# Structures vs classes

- Classes are reference types & Struct are value types

- Structures do not support inheritance.

- Structures cannot have default constructor

Func<T, TResult>

⇒ What is Func<T, TResult> in C#?

In simple terms, Func<T, TResult> is just a generic delegate. Depending on the requirement, the type parameters T & TResult can be replaced with the corresponding type arguments.

Func<T, TResult>

- Input parameter
- What it's going to return.

For Example, Func<Employee, String> is a delegate that represents a function expecting Employee object as Input parameter & returns a String.

Func<TResult>

Requires no Input Parameters

what it's going to Return.

Func<T1, T2, T3, T4 ... T16, TResult>

=> Think of it as a placeholder. It can be quite useful when you have code that follows a certain pattern but need not be tied to any particular Functionality.

* For Example, Consider `Enumerable.Select` extension method.

- the `pattern` is: For every item in a sequence, select some value from that item (e.g. property) and create a new sequence consisting of these values.
- The `placeholder` is: Some selector function that actually gets the value for the sequence described above.
- This method takes an `Func<T, TResult>` instead of any concrete Function. This allows it to be used in any context where above pattern applies.
- So, for example, say I have a `List<person>` and I want just the name of every person in the list.

```
var names = people.Select(p => p.Name);
```

```
var Ages = people.Select(p => p.Age);
```

Alternative would be this.

```
var names = GetPersonNames(people);
var Ages = GetAges(people);
```

`Func<T>` is a predefined delegate type for a method that returns some value of the type `T`.

In other words, you can use this type to reference a method that returns some value of `T`.

```
public static string GetMessage()
{
    return "Hello World";
}
```

May be referenced like this

```
Func<string> f = GetMessage;
```

```
/*
 Func<TResult> represents a method taking 0 arguments and returning an object of TResult, whereas Action<T> represents a method returning void.
    You need two different delegates as you can't specify void as a type argument.

Func is like Action except it returns an object of the last generic type passed in.
    So for instance, Func<string> takes no parameters and returns a string.
Func<int, string> takes an int as a parameter and returns a string.

 */
public static class FuncExample_1
{

    public static void Run()
    {
        Person p1 = new Person() { Name = "John", Age = 41 };
        Person p2 = new Person() { Name = "Jane", Age = 69 };
        Person p3 = new Person() { Name = "Jake", Age = 12 };
        Person p4 = new Person() { Name = "Jessie", Age = 25 };

        List<Person> people = new List<Person>() { p1, p2, p3, p4 };

        Func<Person, string> selector = (person) => "Name = " + person.Name;

        IEnumerable<string> names = people.Select(selector);

        foreach(string name in names)
        {
            Console.WriteLine(name);
        }

    }


}
```

Action in C#

Action is a delegate.

```
public delegate void Action();
```

```
class Program
{
    public static void FooMethod()
    {
        Console.WriteLine("called Foo");
    }

    static void Main()
    {
        Action Foo = FooMethod;

        Foo();
    }
```

An Action is a delegate which has already been defined (void return & no args)

```
Public class Employee
{
    Public int ID {get; set;}
    Public string Name {get; set;}
}
```

```
List<Employees> listemployees = new
                    List<Employees>()
{
    new Employee { ID = 101, Name =
                        "mark" },
    new Employee { ID = 102, Name =
                        "John" };
);
```

```
Func<Employee, string> selector = employee => "Name =" + employee.Name;
IEnumerable<string> names = listEmployees.Select(selector);

    foreach (string name in names)
    {
        Console.WriteLine(name);
    }
```

What is the difference between Func delegate & lambda expression?

- They're the same, just two different ways to write the same thing. the lambda syntax is newer, more concise & easy to write.

What if I have to pass two or more input parameters?
As of this recording, there are 17 overloaded versions of Func which enables us to pass variable number & type of input parameters.

```
class program
{
    public static void main ()
    {
        Func<int, int, string> FuncDelegate = (FNumber, SNumber)
                                              =>
        "SUM =" + (FNumber + SNumber).ToString();

        String result = FuncDelegate (10, 20);
        Console.WriteLine (result);
    }
}
```

Anonymous method

Anonymous method is a method without a name. Introduced in C# 2.0, they provide us a way of creating delegate instances without having to write a separate method.

Step 1: create a method whose signatures matches with the signatures of predicate <Employee> delegate.

```
private static bool FindEmployee (Employee Emp)
{

}
    return emp.ID == 102;
```

Example of Subclass & Superclass

```csharp
using System;

// Define a superclass called Animal
class Animal
{
    // Property
    public string Species { get; set; }

    // Constructor
    public Animal(string species)
    {
        Species = species;
    }

    // Method
    public virtual void Speak()
    {
        Console.WriteLine("I am an animal.");
    }
}

// Define a subclass called Dog, inheriting from Animal
class Dog : Animal
{
    // Constructor
    public Dog(string name) : base("Dog")
    {
        Name = name;
    }

    // Property
    public string Name { get; set; }

    // Method
    public override void Speak()
    {
        Console.WriteLine("Woof!");
    }

    // Method specific to Dog
    public void Fetch()
    {
        Console.WriteLine($"{Name} fetches the ball.");
    }
}

// Define another subclass called Cat, inheriting from Animal
class Cat : Animal
{
    // Constructor
    public Cat(string name) : base("Cat")
    {
        Name = name;
    }

    // Property
    public string Name { get; set; }

    // Method
    public override void Speak()
    {
        Console.WriteLine("Meow!");
    }
```

Class Dog : Animal

Subclass

Superclass

* In C# types are divided into 2 broad Categories.

- Values types - int, float, double, struct, enum, etc.
- Reference types - interface, class, delegates, array, etc. String.
- Reference type default type is null.
- Values type is 0.

* Built-in types in C#.

- Boolean type - only true or false.
- Integral type - sbyte, byte, short, unshort, int, uint, long, ulong, char
- Floating types - float & double.
- decimal types
- string types

* Sbyte = -128 to 127 - Signed 8-bit integer

* byte = 0 to 255 - unsigned 8-bit integer.

```
Console.WriteLine ("Min = {0}", int.MinValue);

Console.WriteLine ("Max = {0}", int.MaxValue);
```

- Floating point Types C#

- float - $\pm 1.5e^{-45}$ to $\pm 3.4e^{38}$ - 7 digits. - 32 bits

- double - $\pm 5.0e^{-324}$ to $\pm 1.7e^{308}$ - 15-16 digits - 64 bits

decimal - much bigger precision. 128-bit

→ String Name = "\" Program " ;

→ Data Type Conversion.

* int i = 100; smaller datatype.

* float f = i; bigger datatype.

- Implicit conversion is done by compiler.
  1. When there is no loss of information if the conversion is done.
  2. If there is no possibility of throwing exceptions during the conversion.

→ float F = 123.45F;

→ int i = f;

→ int i = (int) f;

→ int i = Convert.ToInt32(F);

→ Using PATA = ProjectA.TEAMA.;

→ Using PATB = ProjectA.TeamB;

* Using inheritance

```
Public class Employee {
    Public void Print Full name {
    }
}
```

Move all the common code into base class.

```
public class Full Time Employee {
    float Yearly salary;
}
```

derived class.

```
public class Part Time Employee
{
    float Hourly Rate;
}
```

derived class

- Class can have only one base class. + supports multiple interface inheritance.
- C# supports only single class inheritance.
- C# supports multiple interface inheritance.
- child class is a specialization of base class.
- Base classes are automatically instantiated before derived classes.
- parent class constructor executes before child class constructor.

If you want to hide base class member, then you can use new keyword in child class.

```
public new void PrintFullName()
{
    Console.WriteLine("FirstName");
}

public new void PrintFullName();
{
    base.PrintFullName();
}
```

- `PartTimeEmployee PTE = new Employee();` ✗

  `PartTimeEmployee` ↑ Child class — `Employee()` ↑ parent class.

- `Employee employee = new PartTimeEmployee();` ✓

  `Employee` ↑ parent class — `PartTimeEmployee()` ↑ Child class.

- polymorphism enables us to invoke derived class methods using base class reference variable.

- in the base class the method is declared virtual, & in the derived class we override the same method.

- the virtual keyword indicates, the method can be overriden in any derived class.

Structs

- Just like classes Structs can have

- private fields
- public properties
- constructors
- methods

```
public struct Customer {

    private int _id; // properties.

    public int Id { get { return _id; } set { _id = value; } }
                    ↑
        // public Property.

    public Customer (int id, string Name) { // constructer.
        this._id = Id;
    }

    public PrintDetails() // method
    {
        Console.WriteLine();
    }
}
```

* Values types Vs Reference Types. *

Customer c1 = new customer (); // Struct Initialization.

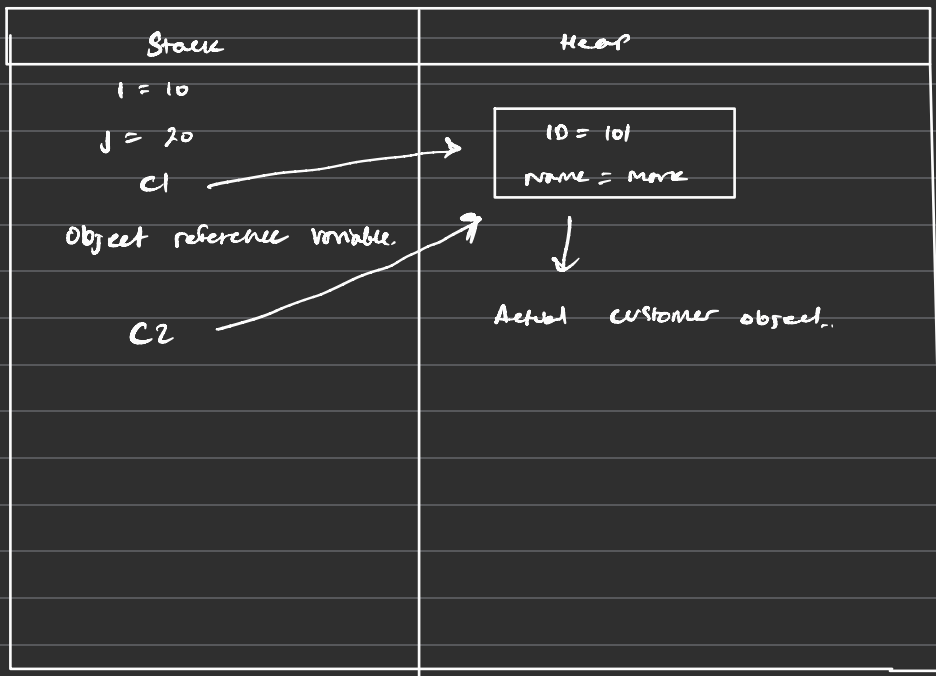
- A **Struct** is a **Value type** where a **class** is **Reference type.**

- Structs are stored on Stack, where as classes are stored on heap.

- Values types are destroyed immediately after the scope is lost, whereas for the reference types only the reference variable is destroyed after the scope is lost. the object is later destroyed by garbage collection.

- When you copy a struct into another struct, a new copy of that struct gets created & modifications on one struct will not affect the values contained by other struct.

- When you copy a class into another class, We get a copy of the reference variable, Both the reference variables point to the same object on the heap. so, operations on one variable will affect the values contained by the other reference variable.



- Struct cannot ~~◎~~ have parameterless constructor.
  ~~#~~ Interface.
- Interfaces cannot contain Fields.

- A class or struct can inherit from more than one interface at the same time, but where as, a class cannot inherit from more than one class at same time.

- Interfaces can inherit From Other Interfaces. A class that inherits this interface must provide implementation for all interface members in the entire Interface inheritance chain.

- Interfaces cannot have Fields.

- We cannot create an instance of an interface, but an interface reference variable can point to a derived class object.

- Explicit interface implementation

- Question: A class inherits from 2 interfaces & both the interfaces have the same method name. How should the class implement the method for the both the interfaces?

- Explicit interface.

Abstract classes

- Abstract keyword is used to create abstract class.
- An Abstract class is incomplete & hence cannot be instantiated.

- abstract class can only be used as a base class.

- abstract class cannot be sealed.

- Abstract class may contain abstract members (methods, properties, indexers, & events) but not mandatory.

- A non-abstract class derived from an abstract class must provide implementations for all inherited abstract members.

- Abstract classes can have implementations for some its members (method's), but the interface can't have implementations for any of its members.

- Interfaces can't have field's where as an abstract class can have field's.

- An interface can inherit from another interface only & cannot inherit from an abstract class, whereas as an abstract class can inherit from another abstract class or another interface.

- Abstract class members can have access modifiers where as interface members cannot have access modifiers.

Delegates in C#

* A delegate is a type safe function pointer. That is, it holds a refference (pointer) to a function.

- Syntax of a delegate.

- `public delegate void HelloFunctionDelegate(string message);`
  (`void` ↙ Return type; `HelloFunctionDelegate` ↙ Name of delegate)

(↙ Points to this function)

```
public class program {

    public static void Main()
    {
        HelloFunctionDelegate del = new HelloFunctionDelegate(Hello);
    }

    public static void Hello(string strMessage)
    {
        Console.WriteLine(strMessage);
    }

}
```

- The signature of the delegate must match the signature of the function, the delegate points to

- A delegate is similar to a class. You can create an instance of it, & when you do so, you pass in the function name as a parameter to the delegate constructor, & it is to this function the delegate will point to.

→ A multicast delegate is a delegate that has reference to more than one function. When you invoke a multicast delegate, all the functions the delegate is pointing to, are invoked.

There are 2 approaches to create a multicast delegate, depending on the approach you use

+ or += to register a method with the delegate
- or -= to unregister a method with a delegate

Note: A multicast delegate, invokes the methods in the invocation list, in the same order in which they are added.

If the delegate has return type other than void & if the delegate is a multicast delegate, only the value of the last invoked method will be returned. Along the same lines, if the delegate has an out parameter, the value of the output parameter, will be the value assigned by the last method.

```
public delegate void SampleDelegate();

class Program {

    public static void Main() {
        SampleDelegate del1, del2, del3, del4;

        del1 = new SampleDelegate(SampleMethodOne);
        del2 = new SampleDelegate(SampleMethodTwo);
        del3 = new SampleDelegate(SampleMethodThree);

        del4 = del1 + del2 + del3;
        del4();
    }

    public static void SampleMethodOne() { Console.WriteLine("1"); }

    public static void SampleMethodTwo() { Console.WriteLine("2"); }

    public static void SampleMethodThree() { Console.WriteLine("3"); }

→   del4 = del1 + del2 + del3 - del2;   ← Remove pointer to two.
```

→ Another way to multicast delegate

```
SampleDelegate del = new SampleDelegate(SampleMethodOne);
del += SampleMethodTwo;
del += SampleMethodThree;

del();
```

→ `del -= SampleMethodTwo;` // Remove Method Two.

- Exceptions.

→ StreamReader → goes out to this file, reads the contents.

→

Exception - unforeseen errors.

- An Exception is an unforeseen error that occurs when a program is running.

- Examples:
  - trying to read from file that does not exists, throws FileNotFoundException
  - trying to read from a database table that doesnot exists. throws SqlException.

Delegate code Example

```
/*
  The point is that some piece of code can  take reference to a method without knowing what method it actually receive.
  It can later call that delegate at will.
  That Enables more abstractions than otherwise possible.
  But, Every once in a while you might feel the need to send a method as parameter to another method, and that's when you'll
  need delegates
 */
public static class DelegateExample_1
{
    public delegate bool FilterDelegate(Person p);

    public static void Run()
    {
        Person p1 = new Person() { Name = "John", Age = 41 };
        Person p2 = new Person() { Name = "Jane", Age = 69 };
        Person p3 = new Person() { Name = "Jake", Age = 12 };
        Person p4 = new Person() { Name = "Jessie", Age = 25 };

        List<Person> people = new List<Person>() { p1, p2, p3, p4 };

        DisplayPeople("Children:", people, IsChild);
        DisplayPeople("Adults:", people, IsAdult);
        DisplayPeople("Seniors:", people, IsSenior);
    }

    public static void DisplayPeople(string title, List<Person> people, FilterDelegate filter)
    {
        Console.WriteLine(title);

        foreach (Person p in people)
        {
            if (filter(p))
            {
                Console.WriteLine("{0}, {1} years old", p.Name, p.Age);
            }
        }
        Console.Write("\n\n");
    }
    public static bool IsChild(Person p)
    {
        return p.Age < 18;
    }
    public static bool IsAdult(Person p)
    {
        return p.Age >= 18;
    }
    public static bool IsSenior(Person p)
    {
        return p.Age >= 65;
    }
}
```

How to save Images / large Data to Database ?

```
Create Table tblImages (
Id int primary key identity,
Name nvarchar (255),
Size int,
ImageData varbinary(max)
)
```

storing Images.

InputStream ← returns a stream objects that points to uploaded file.

```
BinaryReader binaryReader = new BinaryReader(Stream);
```

- Sends in form of bytes[] → varbinary(max) to SP.

ExecuteScalar() ← returns single value.

```
byte[] bytes = (byte[]) cmd.ExecuteScalar();
string stringBase64 = Convert.ToBase64String(bytes);
Image1.ImageUrl = "data:Image/png;base64," + stringBase64;
```

→ How & where are indexers used.

to store or retrieve data from session state or application state variables, we use indexers.

* If you view the Metadata of HttpSessionState class, you can see that there is an Integral & String indexers defined.

We use "this" keyword to create indexers in C#.

[ ] ← int indexers
[ ] ← String indexers

rdr[0] ← position or rdr["column"] ← columnName;

- What are indexers in C#?
  - From the above examples, it should be clear that, indexers allow instances of a class to be indexed just like arrays.

C# Class

```
public class Company
{
    private List<Employee> listEmployees;

    public Company()
    {
        listEmployees = new List<Employee>();
        listEmployees.Add(new Employee()
                          {
                              EmployeeId = 1,
                              Name = "Mike",
                              Gender = "Male"
                          });
    }

    public string this[int employeeId]
    {
        get {
            return listEmployees.FirstOrDefault(
                                emp => emp.EmployeeId == employeeId).
                                Name;
        }
        set {
            listEmployees.FirstOrDefault(
                                emp => emp.EmployeeId == employeeId).
                                Name = value;
        }
    }
}
```

indexer

→ Using Indexers.

```
Company company = new Company();
```

- `company[1];` // Mike } getters

- `company[1] = "John";` } Setters.
- `company[1];` // John.

→ Indexers are overloaded based on the number & type of parameters.

- `public string this[int employeeId]` Indexer 1
- `public string this[int employeeId, int Age]` Indexer 2.

→ You can do this as long as parameter signature is different.

Int 32 ↙ is Structure.

Enums

- Enums are strongly typed Constants.

- If a program uses set of integral numbers, consider replacing them with enums.

- enum is value type.

```
public enum Gender
{
    Unknown
    Male
    Female
}
```

1. Enums are enumerations.
2. Enums are strongly typed constants. Hence, an explicit cast is needed to convert from enum type to an integral type & vice versa. Also, an enum of one type cannot be implicitly assigned to an enum of another type even though the underlying value of their members are the same.
3. the default underlying type of an enum is int.
4. the default value for first element is zero & gets incremented by 1.
5. It is possible to customize the underlying type & values.

6. enums are value types.

7. Enum keyword (all small letters) is used to create enumerations, where as enum class, contains static GetValues() & GetNames() methods which can be used to list Enum Underlying type values & Names.

Type → Class, enum, structs, etc.

↓ typeof → Enum.GetValues(type .enumTypes)

↓ typeof (Gender) Enum.GetValues(TypeOf (Gender));

by default enums are int, but we can change the data type by

```
public enum : short
{
    Unknown,
    Male,
    Female
}
```

```
Gender gender = (Gender) 3; ✓
Gender gender = 3; X
int num = Gender.Unknown; X
int num = (int) Gender.Unknown; ✓
```

→ Types & Types Members.

- Customer is the type & fields, properties & methods are type members.

- classes, structs, enums, interfaces, delegates are called as types & fields, properties, constructors, methods etc., that normally reside in a type are called as type members.

- In C#, there are 5 different access modifiers.

1. private
2. Protected
3. internal
4. protected internal
5. public

* Type members can have all the access modifiers, where as types can have only 2 (internal, public) of the 5 access modifiers.

private members are available only with in the containing types where as public members are available any where. There is no restriction.

- protected members are available, with in the containing type & to the types that derive from the containing type.

- Member with internal access modifier is available anywhere with in the containing assembly. It's a compile time to access, an internal member from outside the containing assembly.

- protected internal members can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly.

- It is a combination of protected & internal.

+ assembly → .exe or dll

→ When to use a dictionary over list.

- Find() method of the list class loop thru each object in the list until match is found.

* So, if you want to lookup a value using a key dictionary is better for performance over list. So, use dictionary when you know the collection will be primarily used for lookups.

1. A dictionary is a collection of (key, value) pairs.
2. Dictionary class is represent in System.Collections.Generic namespace.
3. When creating a dictionary, we need to specify the type for key & value.
4. Dictionary provides fast lookups for values using keys.
5. Keys in the dictionary must be unique.

→ Dictionary<TKey, TValue>

to get Record by key ⇒ Dictionary[KeyValue]

* A way to Loop through a Dictionary

⇒ foreach (KeyValuePair <int, Customer> customer in Dictionary)
{ ... }
or var

throws Exception when Looking up a key that Doesn't exist.

try Get Value ( ) to avoid Exception.

→ dictionary . try Get Value (101 , out customer)
Key
variable.

→ Ways to count elements in Dictionary.

→ dictionary . Count . or dictionary . Count ( )
comes with dictionary.
Linq.

→ dictionary . Count (kvp ⇒ kvp . Value . Salary > 4000)
Predicate.

→ dictionary . Remove (key) If key doesn't exists, no exception happens.

List Collection in C#

[] to Dictionary.

```
customers.ToDictionary(cust => cust.ID, cust => cust);
```
(array; key, value)

⇒ **List** is one of the generic collection classes present in

System.Collections.Generic namespace.

there are several generic collection classes in System.Collections.Generic namespaces as listed below.

1. Dictionary
2. List
3. Stack
4. Queue, etc.

- A List class can be used to create a collection of any type.

for example, we can create a list of integers, strings & even complex types.

→ the objects stored in the list can be accessed by index.

→ Unlike arrays, lists can grow in size automatically

→ class also provides methods to search, sort, & manipulate lists

* Ctrl + Shift + B → Build.

→ List are strongely typed.

```
public class Customer {

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

public class SavingCustomer : Customer {

}

List<Customer> customers = new List<Customer>();
customers.Add(customer);
customer.Add(SavingCustomer); // This will work b/c
                              // SavingCustomer is derived from
                              // customer.
```

- List.Insert (int index, item);
  Adds the records @ index position.

- List.IndexOf (Item);

  List.Indexof (item, position to start)

  Item was not found. Returns -1.

* Contains() function - Checks if an item exists in the list. The method returns true if the item exists, else false.

* Exists() function - checks if an item exists in the list based on a condition. This method returns true if the items exists, else false.

* Find() function - Searches for an element that matches the conditions defined by the specified lambda expression & returns the first matching item from the list.

* FindLast() Function - Searches for an element that matches the conditions defined by the specified lambda expression & returns the Last matching item from the list.

* FindAll() function - Returns all the items from the list that match the conditions specified by the lambda expression.

- List to Array — `[]` `List.ToArray();`

- List to Dictionary. — `List.ToDictionary(key, item);`

- Array to List. — `array.ToList();`

Sorting a list of simple types like int, string, char etc, is easy with List.

```
List<int> numbers = new List<int>{1, 8, 7, 5, 2, 3, 4, 9, 6};
numbers.Sort();
```

If you want the data to be retrieved in descending order, Use Reverse() method on list instance.

```
numbers.Reverse();
```

However, when you do the same thing on a complex type like customer, we get a runtime invalid operation exception - Failed to compare 2 elements in the array.

- .NET runtime does not know, how to sort complex types. We have to tell the way we want data to be sorted in the list by implementing IComparable interface.

* .NET Doesn't know how to sort complex type objects.

↑ class.

- List<Customers> customer = new List<Customers>();
  Customers.Sort(); // Runtime error.

to sort a list of complex types without using LINQ, the complex types has to implement IComparable interface & provide implementation for CompareTo() method.

CompareTo() method returns an integer, & the meaning of the return values is shown below.

Greater than Zero - Current instances is greater than the object being compared with.

Less than zero - the current instance is less than the object being compared with.

Is Zero. - the current instance is equal to the object being compared with.

Sorting Complex types (class)

```
public class Customer : IComparable<Customer>
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Salary { get; set; }

    public int CompareTo(Customer other)
    {
        if (this.Salary > other.Salary)
            return 1;
        else if (this.Salary < other.Salary)
            return -1;
        else
            return 0;
    }
}
```

- listCustomers.Sort();

If you prefer not to use the sort functionality provided by the class, then you can provide your own by implementing IComparer interface.

For example, if I want the customers to sorted by name instead of salary.

Step 1 : Implement IComparer Interface

```
public class SortByName : IComparer<Customer>
{
    public int Compare (Customer x, Customer y)
    {
        return x.Name.CompareTo(y.Name);
    }
}
```

Step 2:

```
SortByName sortByName = new SortByName();

listCustomers.Sort(sortByName);
```

1. IComparable<Type> // interface requires CompareTo method.
2. IComparer<Type> // interface requires Compare method.
3. Comparison<T> // delegate

Sorting a list of complex types using Comparison delegate.

- One of the overloads of the Sort() method in List class expects Comparison delegate to be passed as an argument.

```
public void Sort ( Comparison < T > comparison ) ;
```

Step 1: Create a function whose signature matches the signature of system.Comparison delegate. This is method where we need to write the logic to compare 2 customer objects.

```
private static int CompareCustomers ( Customer c1, Customer c2)
{
    return c1.ID.CompareTo (c2.ID);
}
```

Step 2: Create an instance of System.Comparison delegate, & then pass the name of the function created in step 1 as the argument. So, at this point Comparison delegate is pointing to our function that contains the logic to compare 2 customer objects.

```
Comparison <Customer> customerComparer = new Comparison <Customer>
                                          passing Function -> (CompareCustomers);
```

(delegate)

Step 3: pass the delegate instance as argument, to Sort() function.

```
listCustomers.Sort ( customerComparer ) ;
```

Useful methods of List Collection class

TrueForAll() - Returns true or false depending on whether if every element in the list matches the conditions defined by the specified predicate.

AsReadOnly() - Returns a read-only wrapper for current collection. Use this method, if you don't want the client code to modify the collection. i.e Add or remove any elements from the collection. The ReadOnly Collection will not have methods to add or remove items from the collection. You can only read items from this collection.

TrimExcess() - Sets the capacity to the actual number of elements in the list, if that number is less than a threshold value.

ref vs out

The **ref modifier** means.

1. the value is already set &
2. the method can read & modify it.
3. When ref keyword is used the data may pass in bi-directional. (two ways)

The **out modifier** means

1. the value isn't set & can't be by the method until it is set.
2. the method must set it before returning.
3. When out keyword is used the data only passed in Unidirectional (out only)

Queue

Queue is a generic FIFO (First in First out) collection class that is present in System.Collections.Generic namespace

The Queue collection class is analogous to a queue at the ATM machine to withdraw money. The order in which people queue up, will be the order in which they will be able to get out of the queue & withdraw money from ATM. The queue collection class operates in similar fashion. The first item to be added (enqueued) to the queue, will be the first item to be removed dequeued from the queue.

- TO Add items to the end of the queue, use Enqueue() method.

- To remove an item that is present at the beginning of the queue, use Dequeue() method.

- foreach loop iterates thru the items in the queue, but will not remove them from the queue.

- To check if an item, exists in the queue, use Contains() method.

Threads c# Parallelism

→ Thread class, Thread pool, the Async Pattern, & Background Worker.

* parallel programming covers a wider spectrum & refers to the ability to have multiple tasks going on concurrently.

* concurrent does not mean multithreaded.

.NET Parallelism

Most of the parallel extensions in the .NET framework were released with .NET 4.

- The Task parallel library
- The Async & Await keywords
- parallel ~~pp~~ programming in .NET is the Task parallel library. → System.Threading.Tasks.
- two important classes are **Task** & **parallel**
  - the task class is the centerpiece of the Task parallel library
  - A task represents an operation that is running or going to run.

* Using a task class, you benefit from a state-of-the-art fluent API that is easy to use & offers extreme flexibility.

* Another benefit of the Task parallel library is that when it incorporates multithreading, it uses the thread pool.

* the thread pool manages thread usage for maximum throughput & scalability.

```
Task task = new Task ( () =>
{
    Console.WriteLine (" Task on thread {0} started ",
        Thread. corrent thread. ManagedThreadId );

    Thread.Sleep (3000);

    Console.WriteLine (" task on thread {0} Finished ",
        thread. currentthread. ManagedThreadId );
});

task . Starte );
Console.WriteLine (" This is the main thread " );
```

* Since the code defined in the task executes on another thread, any code that follows the execution of the task's Start method will continue to execute immediately.

* Since the asynchronous code is simply going to sleep its thread for 3 seconds, the text "This is main thread!!" will display before the text, "task on thread 'x' finished."

```
Task<int> task = new Task<int>(() =>
{
        Thread.Sleep(3000);
        return 2+3;
});
```

```
task.Start();
Console.WriteLine("This is the main thread");
int sum = task.Result;
Console.WriteLine("the result from the task is {0}",
                  sum);
```

* The `Result` property carries the return value as returned by the code defined in the task.

* the type of the `Result` variable is the type defined by the generic argument used when declaring the task instance.

`Important` - reference to the `Result` property will block the calling thread until the task completes

- on the main thread you would immediately see the message that follows the start method, but the Result message will not show until the task is complete.

- This means that you can execute whatever code you want between the time you start the task & the time you reference the `Result` property, but when you get to the latter you'll be waiting for completion of the task & the thread will be blocked while you wait.

- This is important to remember when you're calling thread happens to be UI thread.

- the **ContinueWith** method lets me define a callback code construct that will get automatically executed when my task completes.

```
task.ContinueWith(taskInfo =>
{
    Console.WriteLine("The result from the task is {0}",
                      taskInfo.Result);
});
```

* the argument for this method is an Action type with its generic argument being of the type of the task.

* So, in this case, the signature of the **continue with** method is `Action<Task<int>>`.

- **Fluent API:** It's an object-oriented API with emphasis on readability. What makes a fluent API visually unique is its use of method chaining.

* `ContinueWith` method actually returns a Task object itself. In fact, if you add a generic argument to the ContinueWith<T>, it will return a Task<T> instance.

* This task can then also call upon its own `ContinueWith` method, & so on.

* This is the reason that the code defined within a `ContinueWith` method executes on another background thread.

→ It is in itself another task.

Firing off the `Start` method on the initial task will start the process going.

the first task will execute on a background thread & when it is complete, its callback as defined in its `ContinueWith` method will execute in another background thread & when it completes, yet another thread will execute any code that may be defined in another `ContinueWith` method.

you can chain the `ContinueWith` method one after the other, each returning its own instance of a Task or Task<T> class. But because the last one is the task whose type must be the initial task type defined at the begining.

Task Factory

* I can easily modify my code to use the task factory. I start by creating an instance of the

* TaskFactory<T> class, where the generic is the type I need to return.

* This is similar to creating a Task<T> instance.

* Using the factory instance, I can define a new task & have it executed by calling upon the `StartNew` method.

* the `StartNew` method's argument is the same Action type as the constructor of the task class & through its lambda expression, can then be used to contain the code construct that will get executed asynchronously. You can do callback chaining using `ContinueWith` exactly the same way as shown above.

* To save even more code, the Task<T> class contains a static property called `Factory` that returns an instance of `TaskFactory<T>`.

When defining task, you may need to use variables that are defined outside of the task. This requires conventional locking techniques in the case that those same variables are modified in parallel while task is running.

Another choice is to send state into the task. provides better encapsulation & code elegance.

```
Task<int> task = new Task<int>((value) =>
{
    int num = (int) value;
    Thread.Sleep(3000);
    return 2 + num;
}, 4);
```

Passing Methods

```
Static int PerformMath (int amt1, int amt2)
{
    return amt1 + amt2;
}

Static void ShowResults (int result)
{
    console.writeline ("Final result is {0} & thread is
        {1}.", result, Thread.Id);
}
```

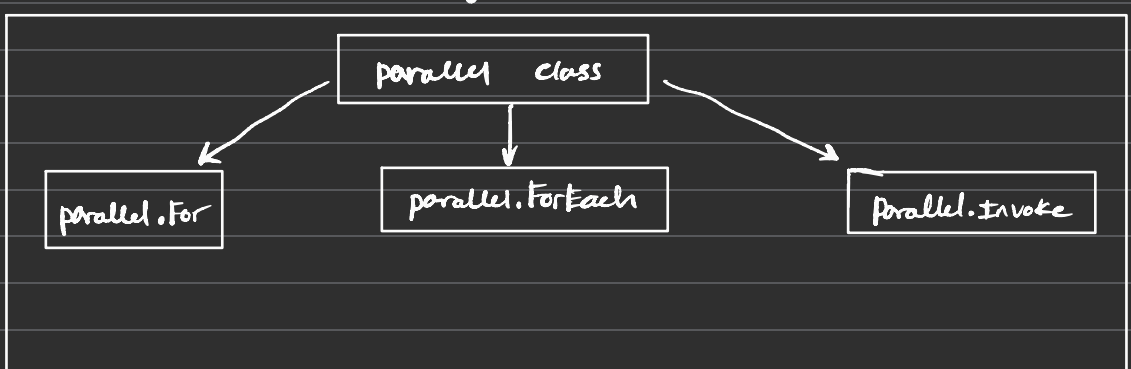
```
Task<int>.Factory.StartNew (() => PerformMath(5,7)).
        ContinueWith (taskInfo => ShowResults (taskInfo.
                Result));
```

⇒ Developers use the `parallel` class for a scenario referred to as Data Parallelism.

* This is a scenario where the same operation is performed in parallel on different items.

The most common example of this is items in a collection or an array which need to be acted upon.

Using `parallel` class, you can perform an operation on members of a collection or any kind of 'Iteration', in Parallel.



```
public Task<string> Get() {
}
```

```
public string Get() {
}
```

Task<string>

⇑

Asynchronous operation.

- Delayed result
- The method does not block the calling thread while it is waiting for the operation to complete.

string

⇑

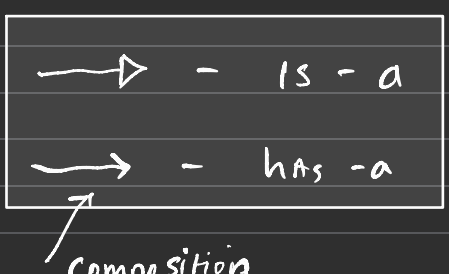
Synchronous operation

- Immediate Result
- The method blocks the calling thread

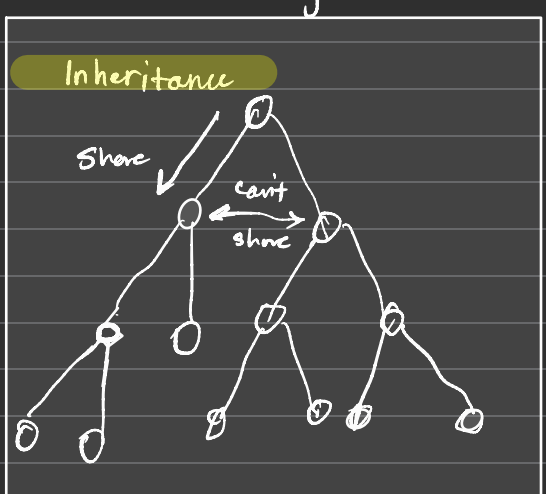
* Design patterns *

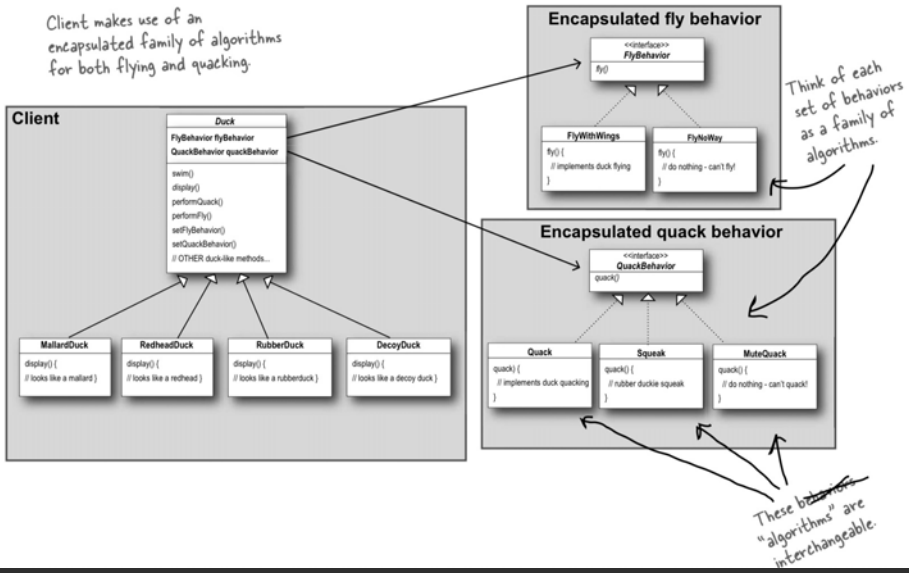
- Dependency injection :- Wishfull thinking.
  - decoupled.

* the **Strategy pattern** defines a family of algorithms, encapsulates each one, & makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



- UML - Diagram.



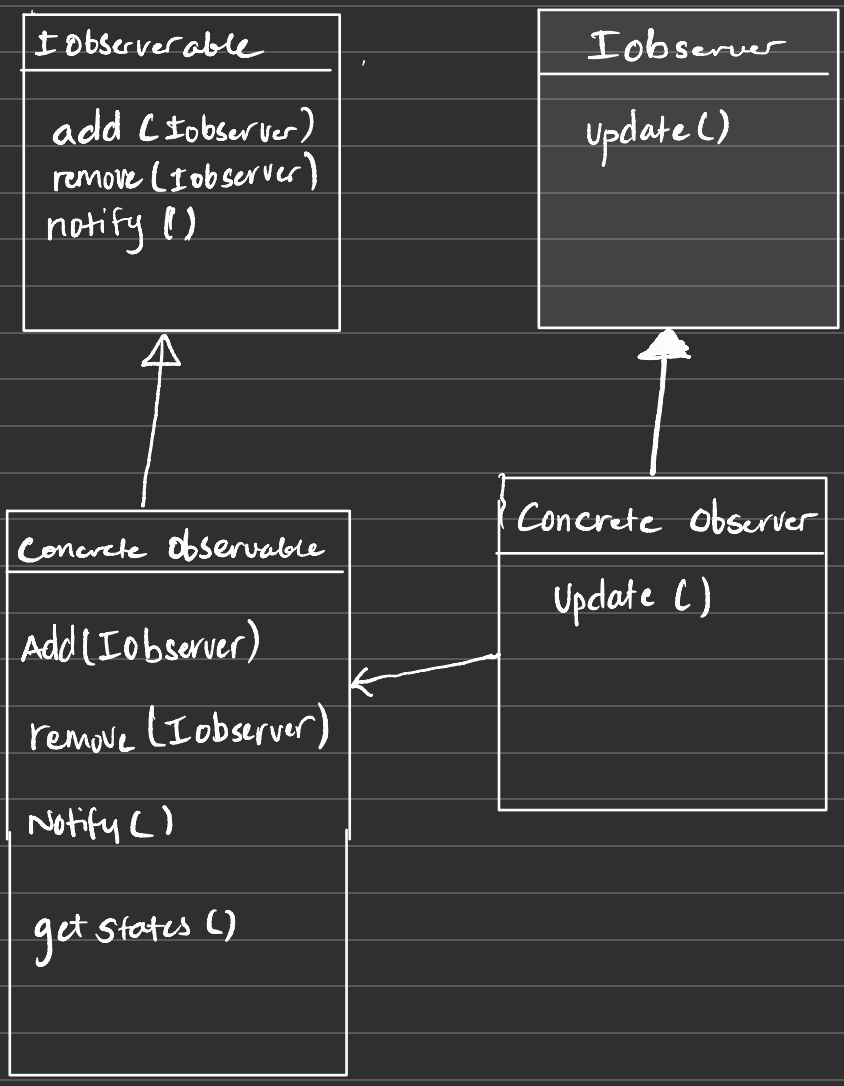



```
public class Duck
{
    IFlyBehavior Fb;

    public Duck (FlyWithWings Fb)
    {
        this.Fb = Fb;

    }
}
```

Observer Pattern

the observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependent are notified & updated automatically

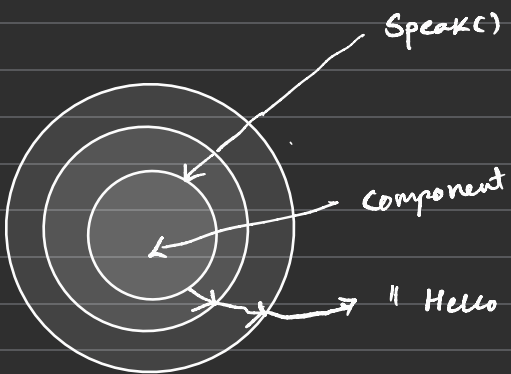
Observer pattern UML

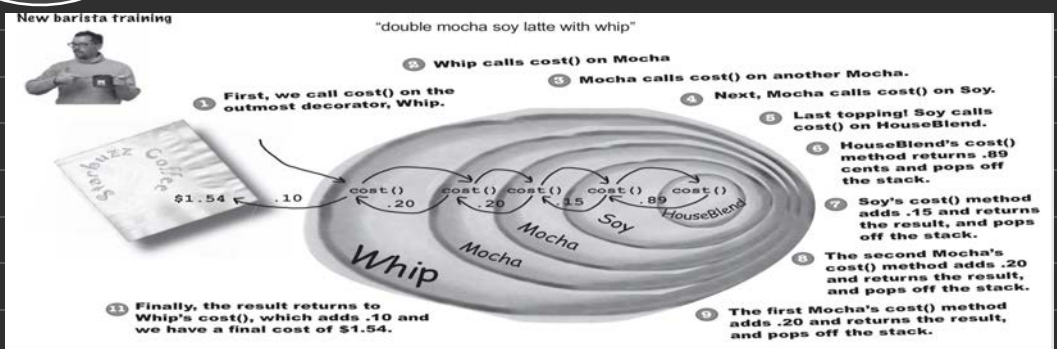


```
Interface IObservable
{
    add(Iobserver o);
    Remove(Iobserver o);
    Notify();
}

Class WeatherStation: Iobservable
{
    public void Add(Iobserver o)
    {
        this.observer.Add(o);
    }
    Public void Notify() {
    foreach(Iobserver o in this.observers)
    {
        O.update();
    }
}
public void getTemperatures()
{
    return this.temperature;
}
}
```

```
Interface IObserver
{
    update();
}

class phoneDisplay : Iobserver
{
    WeatherStation station;
    public phoneDisplay
        (WeatherStation st)
    {
        station = st;
    }
    public void update()
    {
        this.station.
            getTemperature
            ();
    }
}
```

Decorator Pattern



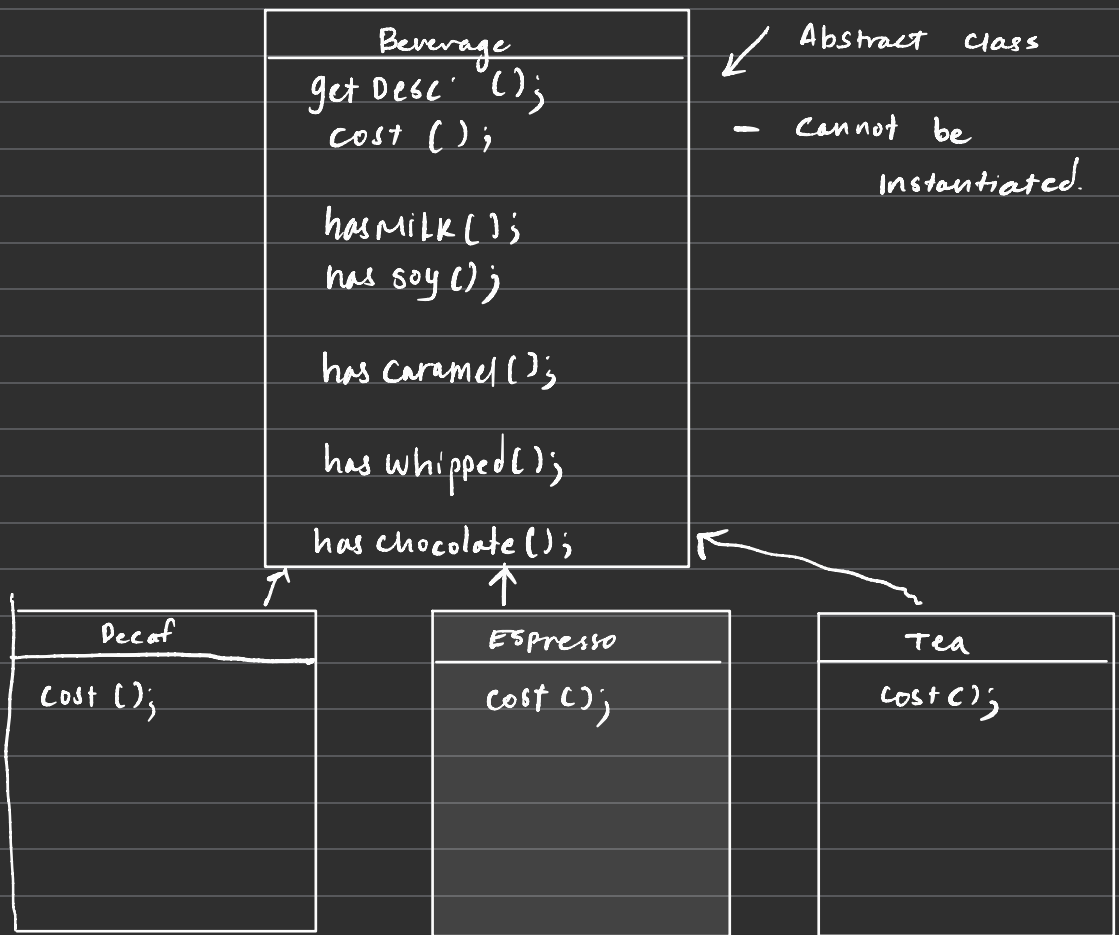


⇒ the Decorator pattern attaches additional responsibilities to an object dynamically. Decorators provides a Flexible alternative to subclassing For extending Functionality.

⇒ Inheritance is NOT For code reuse.

⇒ Abstract classes cannot be instantiated.

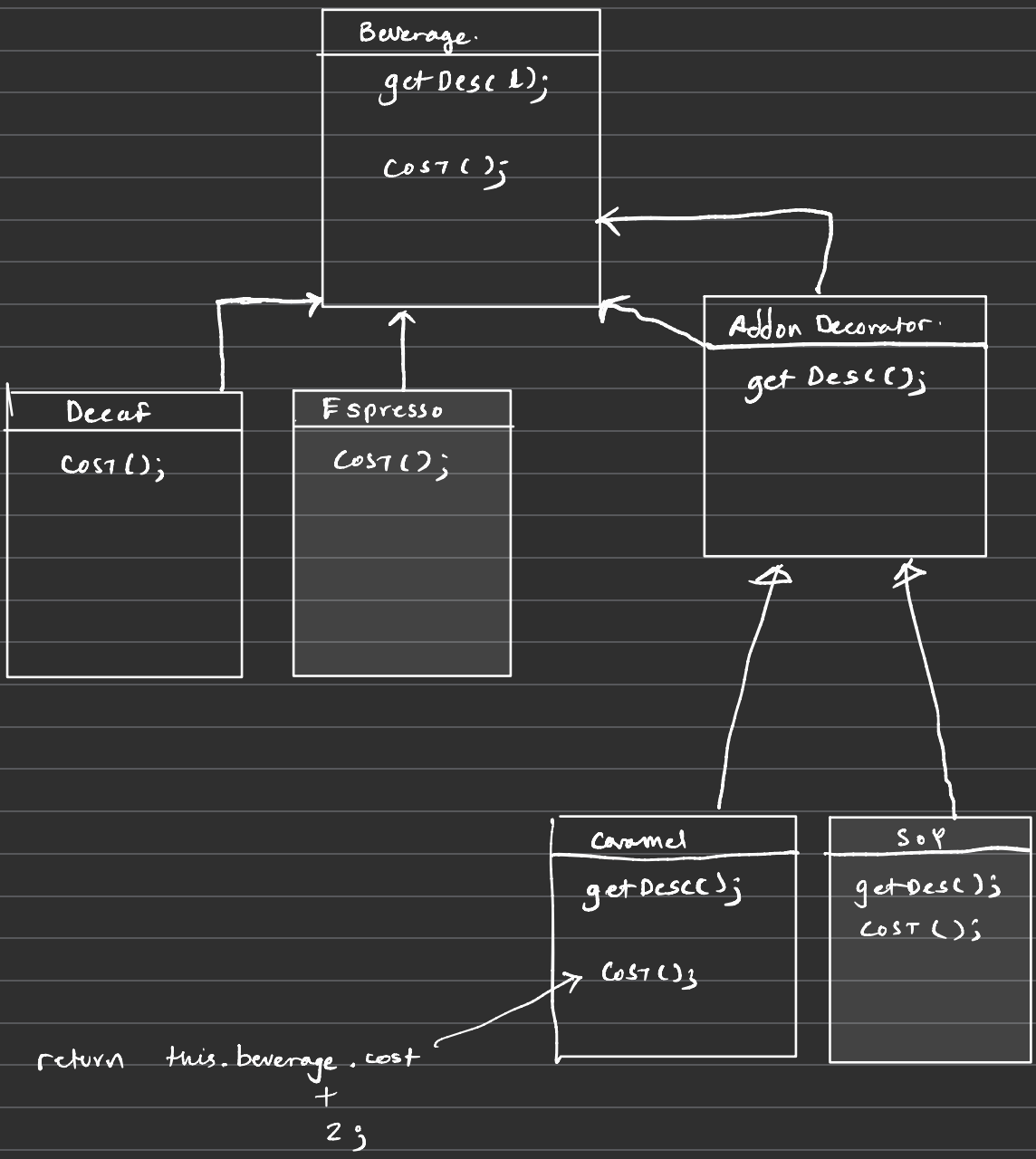
you can Inherit From Abstract class.



- As time goes, number of types of coffee class could go bigger. there could be hundreds of types of coffee.

- Interface segregation : sub class should not be forced to depend on method that it Does Not Use.

- List of substitution : Later.


Beverage
getDesc();
cost();
Addon Decorator
getDesc();
Decaf
cost();
Espresso
cost();
Caramel
getDesc();
cost();
Soy
getDesc();
cost();
return this.beverage.cost
+
2;

```
abstract class Beverage {

    public abstract int cost() {
    }
}
```

```
abstract class AddonDecorator
        : Beverage
{
    public abstract int cost();
}
```

```
Class Espresso() {

    public int cost() {
        return 1;
    }
}
```

```
Class Caramel : AddonDecorator
{
    Beverage beverage;

    public Caramel(Beverage b)
    {
        this.beverage = b;
    }

    public int cost() {

        return this.beverage.cost() + 2;
    }
}
```

Singleton Pattern

⇒ **Singleton pattern** :- ensures a class has only one instance, & provides a global point of access to it.

⇒ Whenever you ask for an instance, you will always get the same instance.

→ Making sure you will only have one instance.

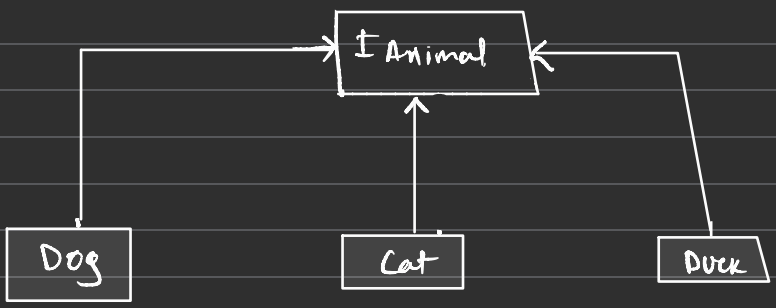
→ & providing a global access.

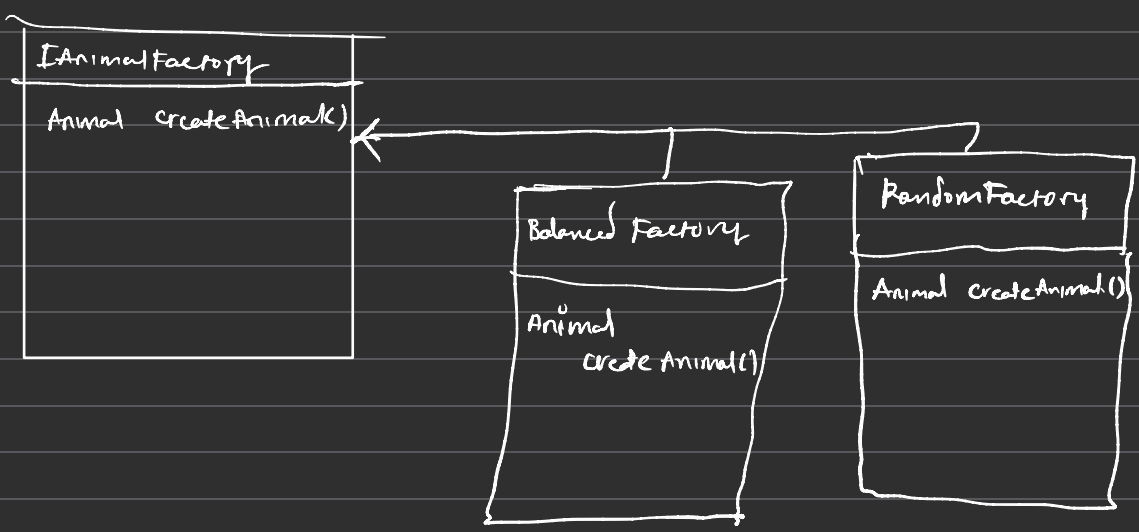
* Programming 101 :- global variables / classes are bad.

```
Class Singleton {

    private static Singleton instance;

    private Singleton () {
    }

    public static Singleton getInstance ()
    {
        if ( instance == null )
        {
            instance = new singleton ();
        }
        return instance;
    }

}
```

Factory method Pattern

- Factory Method pattern : defines an interface for creating an object, but lets subclasses decide which class to instantiate. Method lets a class defer instantiation to subclasses.





Abstract Factory Pattern

=> The abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete class.

The Command Pattern

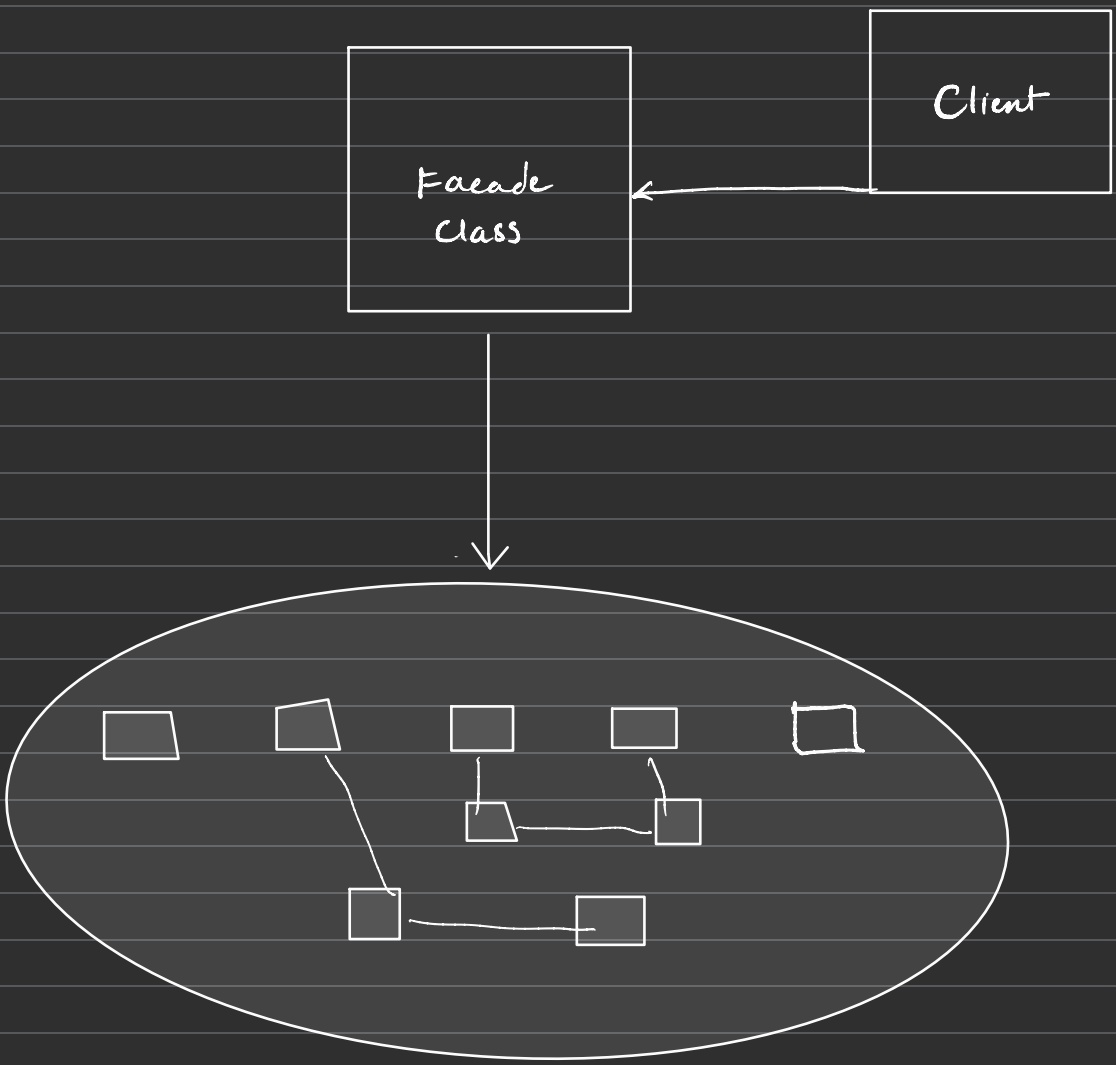
=> encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, & support undoable operations.

The Adapter Pattern

The Adapter pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces

Facade Pattern

Facade pattern you can take a complex subsystem and make it easier to use by implementing a Facade class that provides one, more reasonable interface.



Lambda expressions & Anonymous function

⇒ You can use a lambda expression to create an anonymous function. Use the **Lambda declaration operator** ⇒ to separate the lambda's parameter list from its body.

- **Expression lambda** that has an **expression** as its body.

  (input-parameters) ⇒ expression

- **Statement** lambda that has a **statement block** as its body.

  (input-parameters) ⇒ { <sequence-of-statements> }

parallelism means that we can do two or more things at the same time, you need at least two CPUs for that.

while asynchronism can be achieved with one CPU & means that your threads don't actively wait for the end of some external operation.

→ When we want to return some value, we have two options Task<T> and ValueTask<T>. what is the difference?

- the Task<T> is reference type.
- the ValueTask<T> is value type

parallel.For

parallel.ForEach

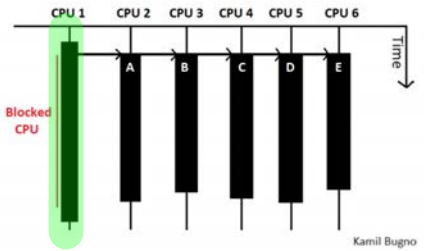
parallel.ForEachAsync

```
static void Main()
{
    var timer = new Stopwatch();
    timer.Start();
    Parallel.Invoke(
        () => HeavyComputation("A"),
        () => HeavyComputation("B"),
        () => HeavyComputation("C"),
        () => HeavyComputation("D"),
        () => HeavyComputation("E")
    );
    timer.Stop();
    Console.WriteLine("All: " + timer.ElapsedMi
}
```

The output is following:

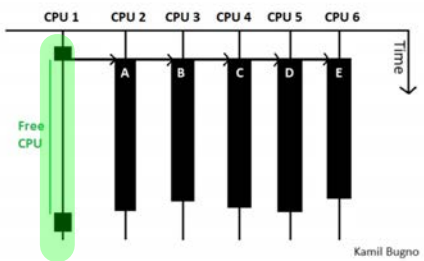
```
Start: B
Start: A
Start: D
Start: C
Start: E
End: B 103
End: C 109
End: E 111
End: A 112
End: D 112
All: 200
```

parallelism with asynchronism.



The first CPU run the `Main` method and execute our computations (A-E) on separate CPUs. It is important that during the computations, the calling thread is actively waiting for the end of the execution of all computations.

- in async way the picture will be as follows:



During the heavy computation, the first CPU is free and can be used as needed.

```
static async Task Main()
{
    var timer = new Stopwatch();
    timer.Start();
    await Task.Run(() =>
    {
        Parallel.Invoke(
            () => HeavyComputation("A"),
            () => HeavyComputation("B"),
            () => HeavyComputation("C"),
            () => HeavyComputation("D"),
            () => HeavyComputation("E"));
    });
    timer.Stop();
    Console.WriteLine("All: " + timer.ElapsedMi
}
```

We just wrapped the `Parallel.Invoke` call inside `Task.Run`. Thanks to it we can use the await/async keywords and enhance the solution with asynchronous programming. Although everything looks fine now, there is still one thing to focus on: our computation returns some value, but so far, we haven't used it. Let's change that!

```
atic async Task Main()

  var timer = new Stopwatch();
  timer.Start();

  var myData = new ConcurrentBag<int>();

  await Task.Run(() =>
  {
      Parallel.Invoke(
          () => { myData.Add(HeavyComputation("
          () => { myData.Add(HeavyComputation("
          () => { myData.Add(HeavyComputation("
          () => { myData.Add(HeavyComputation("
          () => { myData.Add(HeavyComputation("
  });
  timer.Stop();
  Console.WriteLine("All: " + timer.ElapsedMill

  Console.WriteLine(string.Join(",", myData));
```

In C#, We have Special collections designed for concurrent use. One of them is `ConcurrentBag<T>` that is thread-safe & store unordered collection of objects.

Synchronization

⇒ Synchronization is a response to a situation when a lot of threads access the same data.

```
var finalResult = 0;

await Task.Run(() =>
{
    Parallel.For(0, 20, i =>
    {
        finalResult += HeavyComputation(i.ToSt
    });
});
```

⇒ We used the **Parallel.For** method that is a loop whose iterations are executed at the same time. The first Argument starting index, second is the final index. third is an action that contains actual index. Every Parallel operation adds its Result to the finalResult variable.

here we end up w/ race condition.

1. **Thread X** enters the line FinalResult += HeavyComputation(i); It checks the finalResult value & currently, it's zero.
2. Thread Y enters the same line, checks the Final Result Value that is zero, conducts the computations & modify the value to 10.
3. Thread X executes computation & add ten to the value of FinalResult. Because it checked the FinalResult before it was modified by thread Y. the final value is ten instead of 20.

Locks

Race condition usually occurs when a lot of threads share the same data & want to modify them.

Locks are part of exclusive locking constructs. It means that only one thread can be in a section of the code that is protected by the **lock**.

```
var finalResult = 0;
var syncRoot = new object();

await Task.Run(() =>
{
    Parallel.For(0, 20, i =>
    {
        var localResult = HeavyComputation(i.To
        lock (syncRoot)
        {
            //one thread at the same time
            finalResult += localResult;
        }
    });
});
```

We used **lock keyword** that introduces the section where only one thread can be at the same time.

One of the risks of using lock is dead lock. It is situation when all threads are waiting (usually for each other) & application cannot continue working.

To avoid dead lock, it is good to have a separate syncRoot for every lock & minimizing the nesting of the locks.

public int Value1 { get; private set; }

This means we need constructor.

IsyncEnumerable vs IEnumerable

In general, you should use IsyncEnumerable when you need to iterate over a collection of elements asynchronously, or when you need to stream data.

You should use IEnumerable when you need to iterate over a collection of elements synchronously, or when you are working with in-memory collection.

IsyncEnumerable is in System.Linq.Async.

IsyncEnumerable uses the Yield keyword to return elements.

The Yield keyword allows an IsyncEnumerable to return elements one at a time, without having to wait for all of the elements to be available.

```
private static void Main(string[] args)
{
    Console.WriteLine("Hello, World!");

    DoSomething();
}

static  async Task DoSomething()
{
    var numbers = Enumerable.Range(0, 10000000).ToAsyncEnumerable();

    await foreach (int number in numbers)
    {
        Console.WriteLine($"Something {number}");
    }
}
```

Yield class

```
Var people = DataAccess.GetPeople();

foreach (Var p in people)
{
    Console.WriteLine(p);
}
```

```
public class DataAccess
{
    public static IEnumerable<personModel> GetPeople()
    {
        Yield return new personModel("Tim", "Corey");
        Yield return new personModel("Sue", "Storm");
        Yield return new personModel("Jane", "Smith");
    }
}
```

Dependency Injection

⇒ It allows us to instead of saying you know create a new instance of a class whenever we want it, we can put it into dependency injection & just ask for an instance of that class.

→ `builder.Services.AddScoped`

→ `builder.Services.AddSingleton`

→ `builder.Services.AddTransient`

→ `builder.Services.AddTransient<IDemoLogic, DemoLogic>();`

```
public class Result<T>
{
    public bool IsSuccess {get; set;}

    public T Value {get; set;}

    public string Error {get; Set;}

    private Result (T Value, bool IsSuccess, String error)
    {
        Value = Value;
        Error = error;
        IsSuccess = isSuccess;
    }

public static Result<T> Success (T value)
            => new Result<T> (Value, true, null);

public static Result<T> Failure (string error)
            => new Result<T> (default, false, error);
```

Try-catch

In C++, try-catch is used to handle exceptions - unexpected errors that occur during the execution of a program.

When an exception occurs, the normal flow of a program is disrupted, and try-catch allows the developer to handle this disruption gracefully.

Structure

1. **try Block :** The code that might throw an exception is placed inside the try Block.

2. **Catch Block :-** If an exception occurs in the try Block, the catch Block is executed. You can catch specific types of exceptions or catch all exceptions using a general Exception.

3. **Finally Block :** The finally block is optional. It runs regardless of whether an exception was thrown or caught, and is often used for cleanup code (e.g. closing files, releasing resources).

1. If the called method throws an exception and the calling method has a try-catch Block, the calling method can catch the exception.

```
public class Example
{
    public void CallerMethod()
    {
        try {
            calledMethod();
        }
        catch(Exception ex)
        {
            Console.WriteLine($"callerMethod caught the
                exception : {ex.Message}");
        }
    }

    public void CalledMethod()
    {
        throw new InvalidOperationException("something +
            Went Wrong");
    }
}
```

2. If the called method has its own try-catch Block, it can catch the exception itself, and the calling method won't see it unless it rethrows the exception.

```
public Void CallerMethod()
{
  CalledMethod(); // calls the method, but no exception is seen
  console.WriteLine("CallerMethod Finished execution");
}

public void CalledMethod()
{
    try
    {
        throw new InvalidoperationException
              ("Something Went wrong in CalledMethod.");
    }
    Catch (exception ex)
    {
        Console.WriteLine($"CalledMethod caught the
              exception : {ex.Message}"
    }
}
```

In this case, the exception is thrown by CalledMethod(), but CallerMethod() catches it because it has a try-catch Block.

3. If neither method has a try-catch block, the exception propagates up the call stack, potentially crashing the application if it's not caught by any higher-level handlers.

```
public void CallerMethod()
{
    try
    {
        CalledMethod();
    // Calls the method that throws and rethrows.
    }
    catch (Exception ex)
    {
        Console.WriteLine($"CallerMethod caught the
        exception: {ex.Message}");
    }
}
public void CalledMethod() {
    try {
    throw new InvalidOperationException
        ("Something went wrong in CalledMethod.");
    }
    catch(exception ex)
    {
        Console.WriteLine($"CalledMethod caught and
        rethrow the exception.");
        throw;
    }
}
```

Generics

=> generics in c# allow you to write flexible code that can work with different types of data, without specifying the exact data type in advance.

=> This makes your code reusable and safer, as it will work for different types without needing to rewrite the same logic multiple times.

SOLID PRINCIPLES

⇒ The SOLID principles are a set of five object-oriented design principles that aim to make software designs more understandable, flexible, and maintainable.

⇒ Each letter in "SOLID" represents one of these principles.

⇒ these principles help developers create modular, maintainable, and extensible software designs that are less prone to bugs & easier to understand & modify over time.

1. Single Responsibility principle (SRP) : A class should have only one Reason to change, meaning it should have only one responsibility or job within the system. This principle helps keeps classes focused & makes them easier to understand, maintain, & test.

Example :- consider class called Employee that is responsible for both storing employee information & calculating their salaries. This violates the SRP because the class has more than one reason to change. Instead, we could Separate these responsibilities into two classes. Employee for storing information & SalaryCalculator for calculating salaries.

2. Open / closed principle [OCP]: Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that you should be able to extend the behavior of a system without modifying its existing code, which promotes code reuse & minimizes the risk of introducing bugs

Example :- Imagine we have class called Shape with a method calculateArea() to add support for new shapes without modifying the Shape class, we can create a new subclass like Circle and Rectangle that override the calculateArea() method to provide the specific implementation for each shape.

Example : Suppose our messaging application supports sending text messages. Later, we want to extend it to support sending multimedia messages (e.g. images, videos). We can achieve this by creating a new class or module for handling multimedia messages without modifying existing codebase.

3. Liskov substitution principle (LSP) :- objects of a superclass should be replaceable with objects of a subclass without affecting the correctedness of the program. In other words, derived class should be substitutable for their base classes without altering the desirable properties of the program, ensuring that polymorphism works correctly.

Example: If we have a superclass Bird with a method Fly(), then LSP states that any subclass of bird, such as Duck or penguin should be able to substitute for bird without changing the behavior of the program. Therefore, both Duck & penguin should implement the Fly() method penguin might implement it as no-operation.

4. Interface Segregation principle (ISP) :

clients should not be forced to depend on Interfaces they do not use. Instead of having one large interface, It's better to have smaller, more specific Interfaces tailored to the needs of the clients. This helps prevent the clients from being burdened with unneccessary dependencies.

Example :- suppose we have an interface called worker with method work() & eat(). If there are classes that only need to implement work() but not eat(), those classes would be forced to provide an empty implementations for eat(), violating ISP. Instead, we could split the worker interface into two smaller interfaces : Workable & Eatable, allowing classes to implement only the methods they need.

5. Dependency inversion principle (DIP) :- High-Level modules should not depend on Low-Level modules. Both should depend on abstractions. Abstractions should not depend on details, details should depend on abstractions. This principle encourages decoupling between modules by introducing abstractions (interfaces or abstract classes) that both high level & Low-Level modules depend on. Rather than having them depend on directly on each other.

Example :- Let's say we have a class called MessageService responsible for sending messages. Instead of directly, depending on concrete implementations of network communication (e.g. HTTPclient, MessageService should depend on an abstractions like a MessageSender interface. This way, we can easily switch between different network communications implementations without modifying MessageService.

Extension Methods

Extension classes in C# are used to define extension methods, which allow you to add new functionality to existing types (both custom & built-in) without modifying their source code or using inheritance.

Key points about Extension classes & methods

1. Static classes :- The class containing extension methods must be declared as static.

2. Static methods :- Extension methods themselves must also be static.

3. this keyword :- The first parameter of an extension method specifies the type it extends & must be prefixed with the this keyword.

4. Namespace : To use extension methods, the namespace of the extension class must be included with a using directive.

Syntax of Extension Method.

```
public static class MyExtensions
{
    public static string ToFirstUpper(this string str)
    {
        if (string.IsNullOrEmpty(str)) return str;
        return char.ToUpper(str[0]) + str.Substring(1);
    }
}
```