⊙ Dependency Injection / Inversion of control (IoC)

-> Application is divided into 3 parts [illegible] dependant on each other.

-> If one change other also change.

→ So we do loose coupling to remove dependency. Plug and Play environment

→ DI / IoC is structure used for design loosely coupled system.

E.g:

```
: class   FileWriter {

        public void save(String data)
        {
            ≡
        }
}
```

```
→ class  Program
    {     FileWriter fw = new FileWriter();
          fw. save(-);
    }
```

: Now we wanted to save in Database.

```
→ class   SqlWriter
  {
      public void save(-)
      {  ≡
      }
```

: There is fw dependency
: So we make interface

(*) Class represents real-world objects (Properties and functions of it).

⊙ <u>Interfacing</u>:

-> What the class will do, it is defined in interface. Not How part define.

-> Class (contract) for interface and objects/class

-> No Implementation of object is defined here.

-> Abstract Class (No attributes)

-> Interface Class always start with capital "I".

```
interface Iwriter{
        public void Save(string data);
}
```

Not clear where to save

```
-> class SqlWriter : Iwriter {
      public void Save ( .. )
      {
      }
   }
```

```
class FileWriter:Iwriter{
}
```

```
class   program                          ~~still object~~ Made.
 {            fw
      I writer = new (FileWriter) (...);
            fw. Save (....);
 }
```

: ~~We use~~ IOC
: We will make class object outside ~~and inject it to~~ Program code. (Dependency Injection)

```
         class
→ public  Program {
  private  I writer = _Iw
  public  Program ( Iwriter Iw)
   {
        _Iw  = Iw
   }
  _Iw. save(...);
}
```

⊙ In MVC:

→ Controller Layer get data from Repositories.

Controllers	Repositories
Repository rp=new ...();	↳ ~~Product Register~~
~~rp. save( );~~	~~{ public void Save()~~
	{
	≡
	} }
	}

```
-> public   class   Program
   {   private   Iregister = _Ir;
     }

     public Program( IProductRegister Ir)
     {
     }
```

⊙ Model -> class -> customor.

```
     public class customer
     {
            int Id
            string Name
       }
```

-> Model -> Interface -> New Item
(Folder)
ICustomer ⇐ Interface ↵

```
   public interface ICustomer
   {
         public void save (customer c)
                     Updalate
                     Delete
   }
```

-> New Folder -> Repository
CustomederRepository ↵

: Implement
Interface (Option

```
• public class Customer Repository : ICustomer
  {
          Save
          Update
          Delete
          Get All
  }
: CustomerController
  {   private readonly ICustomer _customer;
      public customer(I customer c)
      { _customer = c;
      }
      public IActionResult Index()
      {    return (_customer. GetAll());
      }
```

→ GetAll in Repository.

```
public  List<customers GetAll()
      {  List<customers L = new List.... ;
         List.Add(new customer { Id=1, Name="c1"})
              ''                         2      c2
              ''                         3      c3
              ''                         4      c4
   return (L);
```

→ View → Customer Folder → Index

```
@model IEnumerable<Customer>
       List<Customer>

<table>
  @foreach(var c in Model)
  {
        <tr> <td>@c.Id</td>
             <td>@c.Name</td>
        </tr>
  }
</table>
```

→ Program.cs

→ builder.Services.AddTransient<ICustomer,
                CustomerRepository>();
(In Main(string[] args))

*: Practice Dependency Injection

*: Quiz on Wednesday (BootStrap)