**Dynamic types**

C# 4.0 (.NET 4.5) introduced a new type called dynamic that avoids compile-time type checking. A dynamic type escapes type checking at compile-time; instead, it resolves type at run time.

dynamic a = 3;

a = "hello";

The last one is gonna be taken as a type which is a string in our case.

Int a =3; is a static type

The difference between generic types and dynamic types is that generic

types are resolved at compile time however dynamic types are decided at runtime.

**TimeSpan**

This .NET type represents a length of time. There are 2 ways to create a timespan. One is to use the ctor, the other is to use the static methods of the Timespan object.

TimeSpan time1 = new TimeSpan(1,1,1);

Console.WriteLine(time1);

TimeSpan time2 = TimeSpan.FromMinutes(1);

Console.WriteLine(time2);



We can also use the properties of the TimeSpan object🡪

TimeSpan time = TimeSpan.FromMinutes(1);

Console.WriteLine(time.Minutes); 🡪 the output is 1

TimeSpan time = new TimeSpan(1,1,1);

Console.WriteLine(time.Minutes);

Console.WriteLine(time.TotalMinutes);



Minutes property just takes the number of minutes but total takes the entire timespan and calculates the minutes.

We can also add another time span to our existing timespan🡪

TimeSpan time = new TimeSpan(1,1,1);

Console.WriteLine(time.Add(TimeSpan.FromMinutes(1)).Minutes);



We can also subtacrt Timespan.

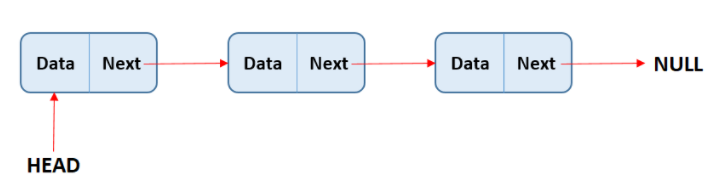
TimeSpan time = new TimeSpan(1,1,1);

Console.WriteLine(time.Subtract(TimeSpan.FromMinutes(1)).Minutes);



**LinkedList**

LinkedList consists of nodes. Each node has a piece of data and reference to the next node.The last node in the LinkedList always points to null.



public class Node

{

int \_data;

public Node next;

public Node(int data)

{

\_data = data;

next = null;

}

public void Print()

{

if (next!=null)

{

Console.Write($"{\_data}-->");

next.Print();

}

else

{

Console.Write($"{\_data}-->");

}

}

public void AddToEnd(Node newNode)

{

if (next != null)

{

next.AddToEnd(newNode);

}

else

{

next = newNode;

}

}

}

Node node = new Node(1);

node.AddToEnd(new Node(3));

node.Print();

**Unary operations**

int x = 0;

Console.WriteLine(x++); //0 it first prints it then increments

Console.WriteLine(++x); //2 increments it then prints

**Constructor**

List<int> x = new List<int>();

Here we initialize parameterless constructor.

List<int> y = new List<int> {1,3};

Here we don’t need a parameterless constructor.

**Method Signature**

It includes The return type, the name, and parameters.

Public int GetName(bool isGraduated)

**S.O.L.I.D. Principles**

SOLID is one of the most popular sets of design principles in object-oriented software development.

Single Responsibility Principle - Robert C. Martin describes it as: A class should have one, and only one, reason to change. Meaning that one class should have just one responsibility and not more. Responsibility here is a reason.



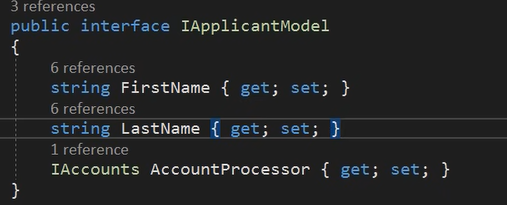
For example, here Employee class has 3 reasons to change. It can change if the CalcPay method fails, ReportHours or WriteEmployee. So It violated the SRP.

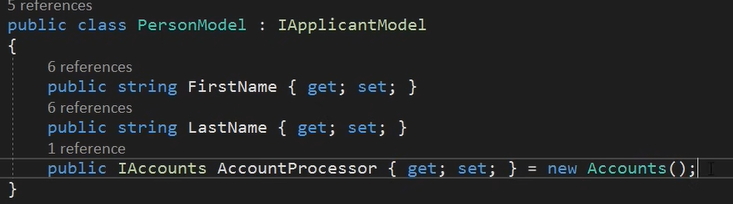


For instance, here this Program class has just one responsibility which is to control the flow of the application. And all the other responsibilities such as Capturing Person’s first name and last name or Messages(StandardMessage) are handled by other classes so it is their responsibility. So we should separate everything. It is not a problem to have many classes. Each of them has their own responsibility. Of any of the classes that we create if one of them has a scroll then we have done something wrong because they gotta be short. Only Main( ) method ,of course, can be a little bit bigger.

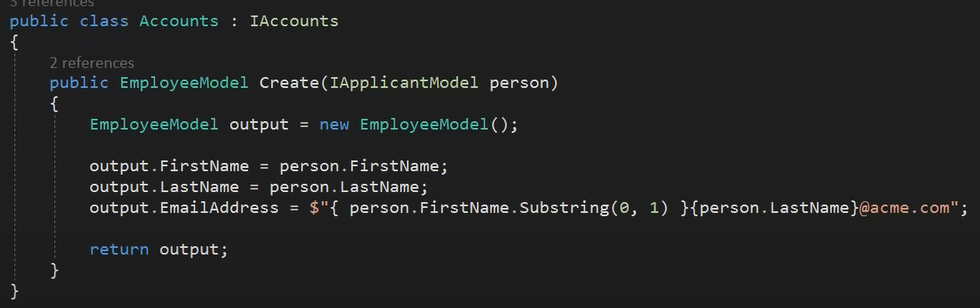
This principle aims to separate behaviours so that if bugs arise as a result of your change, it won’t affect other unrelated behaviours.

Open-Closed Principle - The Open-Closed Principle (OCP) states that software entities (classes, modules, methods, etc.) should be open for extension, but closed for modification. For example, let’s say that we have different models which are just classes: Person, ManagerModel, ExecutiveModel, and also we have different accounts for this models: Accounts, ManagerAccounts, ExecutiveAccounts. So to implement OCP we need to create a IApplicantModel which will be inherited by Person, ManagerModel, ExecutiveModel. And for accounts we will also create an interface (Accounts, ManagerAccounts, ExecutiveAccounts)🡪

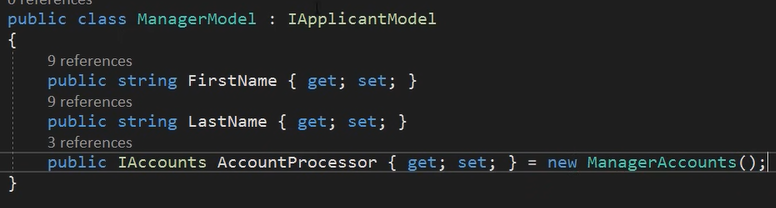




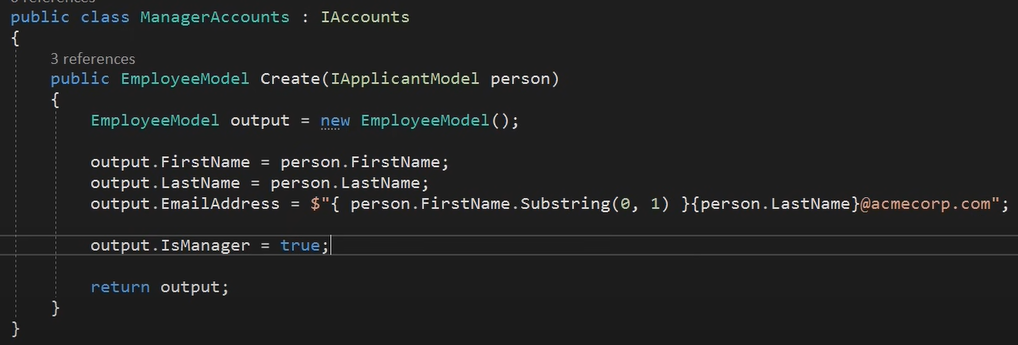
Accounts is a class for normal employees (that are just employees not managers or etc.).

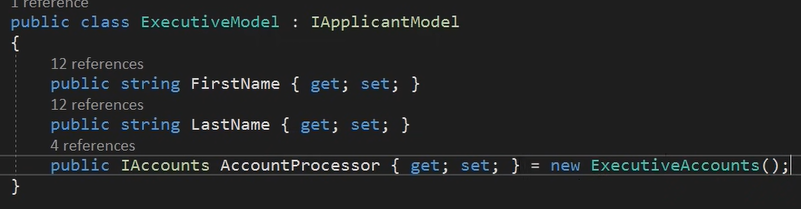


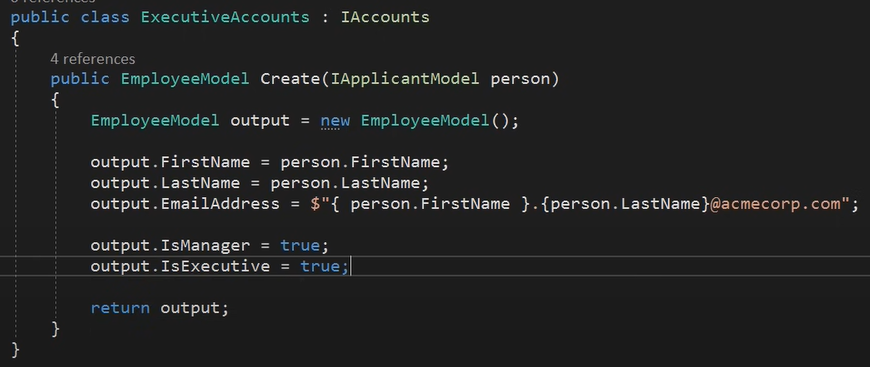
This is for normal employees.



This model is for Managers.



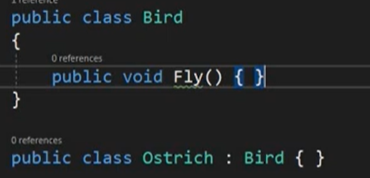




So in the end, if we we wanted to add a new model, for instance, TechnicianModel, then we would create a TechnicianModel class that would inherit from IApplicantModel and we would create Accounts for this Model (TechnicianAccounts) which would be somewhat different. And IAccounts for TechnicianAccounts would point to TechnicianAccounts.

This principle aims to extend a Class’s behaviour without changing the existing behaviour of that Class.

Liskov Substitution Principle- The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application.



For example, here we have an Ostrich class that inherits from Bird class. It doesn’t make sense beacause ostriches cannot fly so LSP is broken. To comply with LSP, we can use OCP. So we can create an interface for FlyingBirds and implement this interface for other birds.

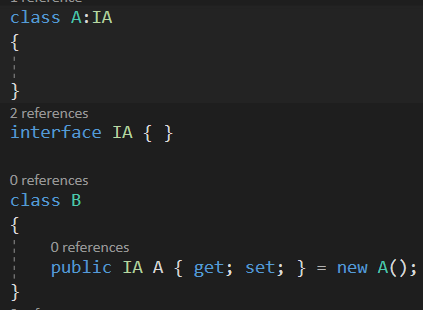
So the goal of this principle is basically providing a proper way for inheritance.

Interface Segregation Principle - Clients should not be forced to depend on methods that they do not use. So if we have multiple method declarations in our interface then our class that implements this interface will be forced to implement them. According to this principle, we need to create other interfaces for other tasks (separate interfaces).

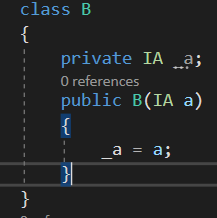
Dependency Inversion Principle - High-level modules should not depend on low-level modules. Both should depend on the abstraction (interfaces). - Abstractions should not depend on details. Details should depend on abstractions.

Bacically, we just should get rid of new keyword or any low module in a high module so that the high module doesn’t depend on that.

Here we can use dependency injection to implement DI.



Here for instcance, even though we used IA in B still we have new A( ). So it breaks DI and in order to make DI we need Dependency injection.



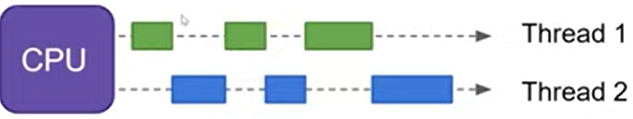
In Dependency injection we just pass that IA as as argument to the constroctor which then assigns that a to our private \_a. Nonetheless, at some point in our application we are gonna have to new up this class.

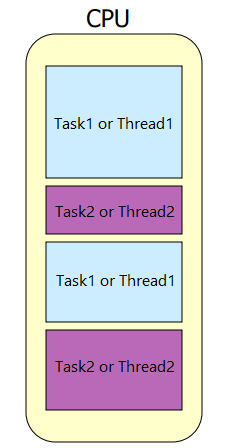
Dependency injection is one of the implementations of DIP.

One of the the benefits of implementing dependency inversion principle is that in unit testing when we test one class it will new up low modules if we have them there and it can take a lot of time or space in memory to new them up. However, if we implement this then we can create mock classes for those interfaces and that’s it.

**Concurrency and Parallelism**

Concurrency- Making progress on more that one task –seemingly at the same time. But this is actually happining one at a time, meaning that firt we do a little bit of task 1 or thread 1 then we go to task 2 or thread 2 then again we go back to task 1 or thread1 to do a little bit of that

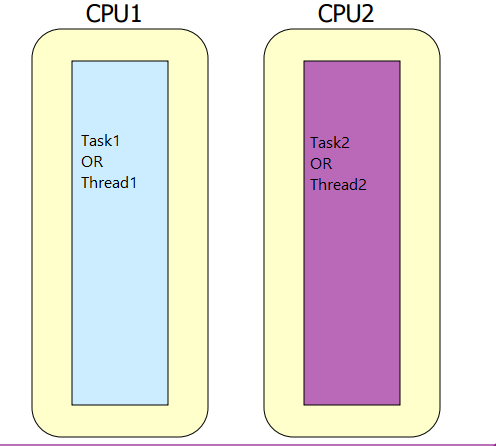




Paralellism is exactly what we think of this, doing multiple things at the same time independently.

Concurrency- our computer takes once core and seperates it into threads. Then these threads are called one at a time.

In c# when we use thread classes, it doesn’t depend on us as to whether it is going to be concurrency or parallelism. Our computer decides it.



**Synchronous vs Asynchronous**

Synchronous or Synchronized means "connected", or "dependent" in some way. In other words, two synchronous tasks must be aware of one another, and one task must execute in some way that is dependent on the other, such as wait to start until the other task has completed.  
Asynchronous means they are totally independent and neither one must consider the other in any way, either in the initiation or in execution.

**Async and Await**

The async keyword only enables the await keyword (and manages the method results).

Async methods can return Task<T>, Task, or void. In almost all cases, you want to return Task<T> or Task, and return void only when you have to.

Why return Task<T> or Task? Because they’re awaitable, and void is not. So if you have an async method returning Task<T> or Task, then you can pass the result to await. With a void method, you don’t have anything to pass to await.

Async methods returning Task or void do not have a return value. Async methods returning Task<T> must return a value of type T:

public async Task<IViewComponentResult> Invoke()

{

ICollection<Product> products = await \_context.Products.Take(8).ToListAsync();

return View(products);

}