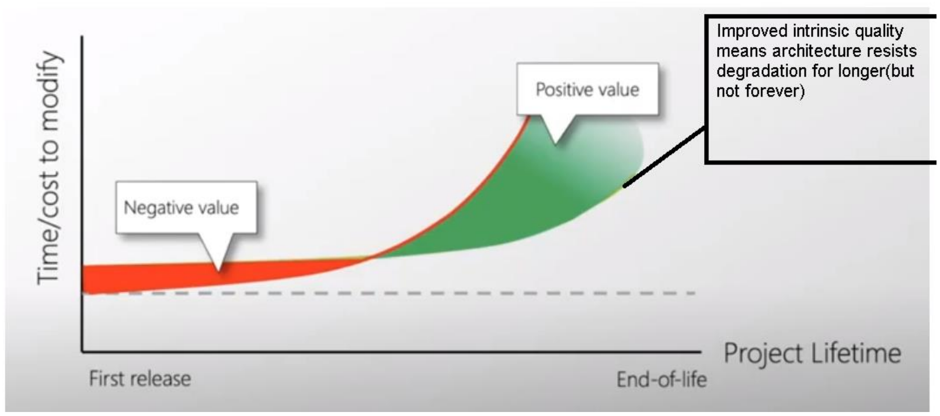
Solid Principles, Factory Pattern

S.O.L.I.D. Principles

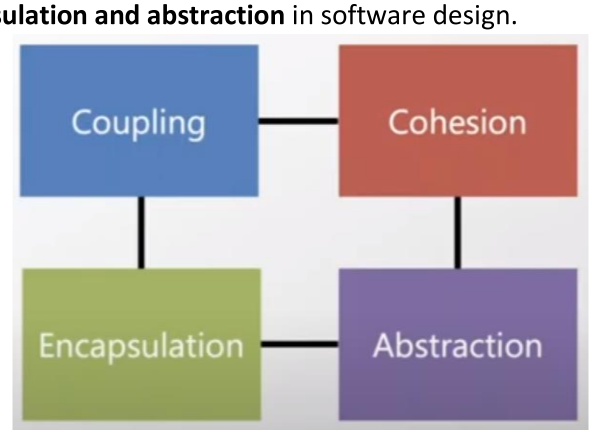
Why SOLID Principle is required.

1. The initial cost of making a software will be high due to the S.O.L.I.D. principles applied on the software design but the maintenance and upgrading capability will be high and very cost efficient.
    - Negative value – initial cost spend for designing the software.
    - Positive value – Low cost of maintenance and upgradation.



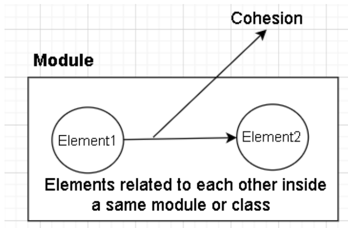
SRP – Single Responsibility Principle

1. Modularization: Technique to divide a software into modules.
    a. An important objective of **Modularization** is to **MAXIMIZE** the **MODULES COHESION** and **MINIMIZE COUPLING** between the **MODULES.**
2. To understand single responsibility principles, first we need to understand **cohesion, coupling**, **encapsulation and abstraction** in software design.

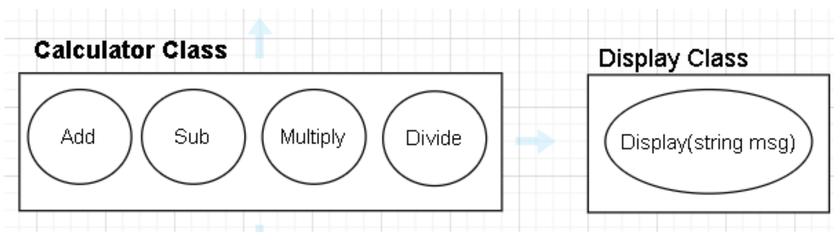


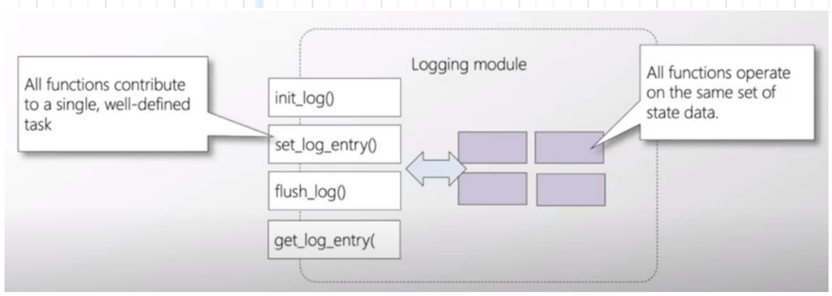
1. Cohesion (Sticking together)

1. Cohesion – Elements of same characteristic should be place near/inside the same elements.
2. A Strongly cohesive module implements functionality focusing on a single feature of the solution.
3. Cohesion is a glue that holds a module together.
4. When a module element changes, it should be changed for the same reason.

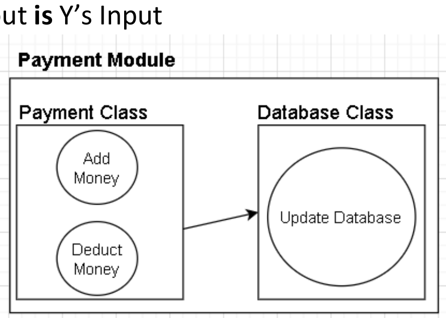


5. Type of Cohesion
   a. Functional Cohesion: If 2 operation present within a module\class perform the same functional task **OR** they are part of the same "Purpose".
   Each Element does some related task. Atomic functions

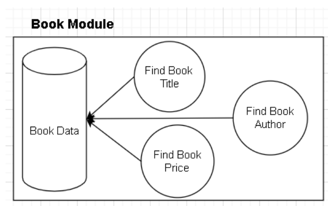


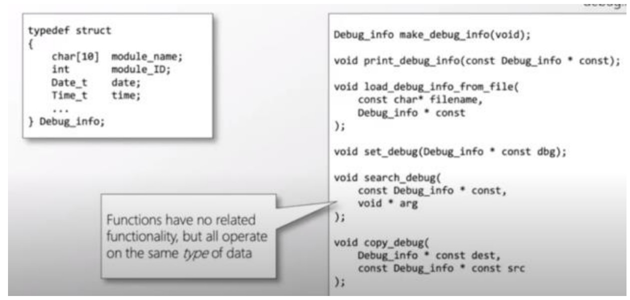


   b. Sequential Cohesion: In a module, If two operations are such that
   X's Output **is** Y's Input

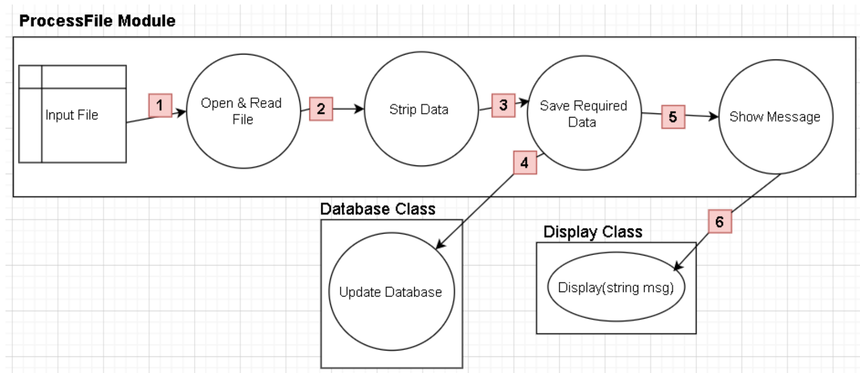


c. Communication (Information) Cohesion: A module exhibits “Communication Cohesion” if all the activities it supports uses the same input or output data **OR** access and modifies the same part of a data structure.



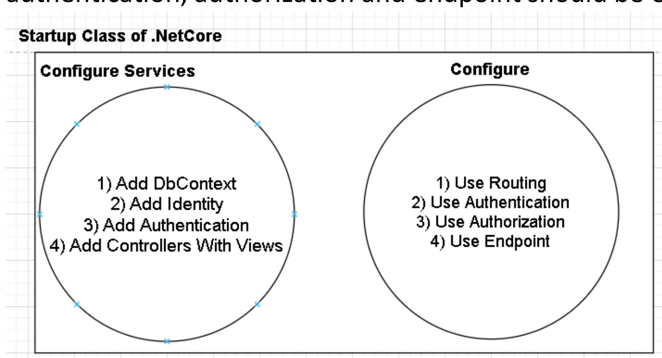


d. Procedural Cohesion: A Module whose instruction do different tasks, but to ensure a particular order in which tasks are performed, they are put into same module
Cons: Poor Maintenance

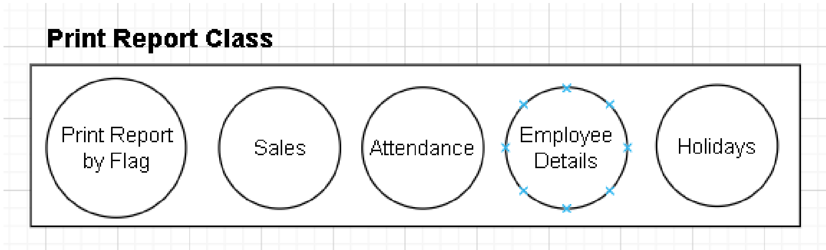


e. Temporal Cohesion: Instruction that must be **executed during same time span** are put together (Activities related in time).

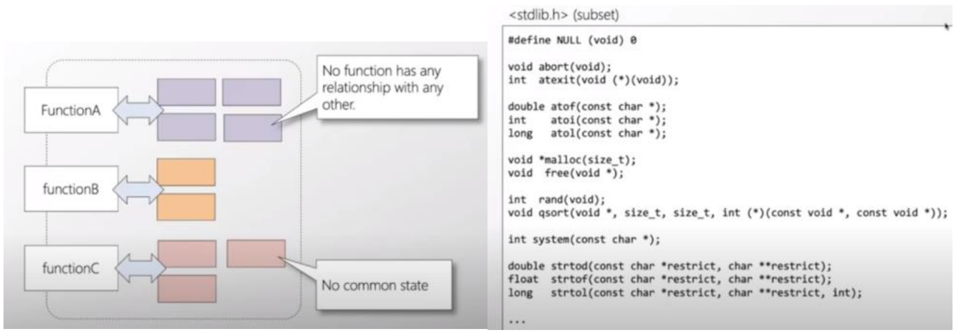
   i. Configure & Configure Services: All the unrelated functionality such as routing, authentication, authorization and endpoint should be executed at same time.



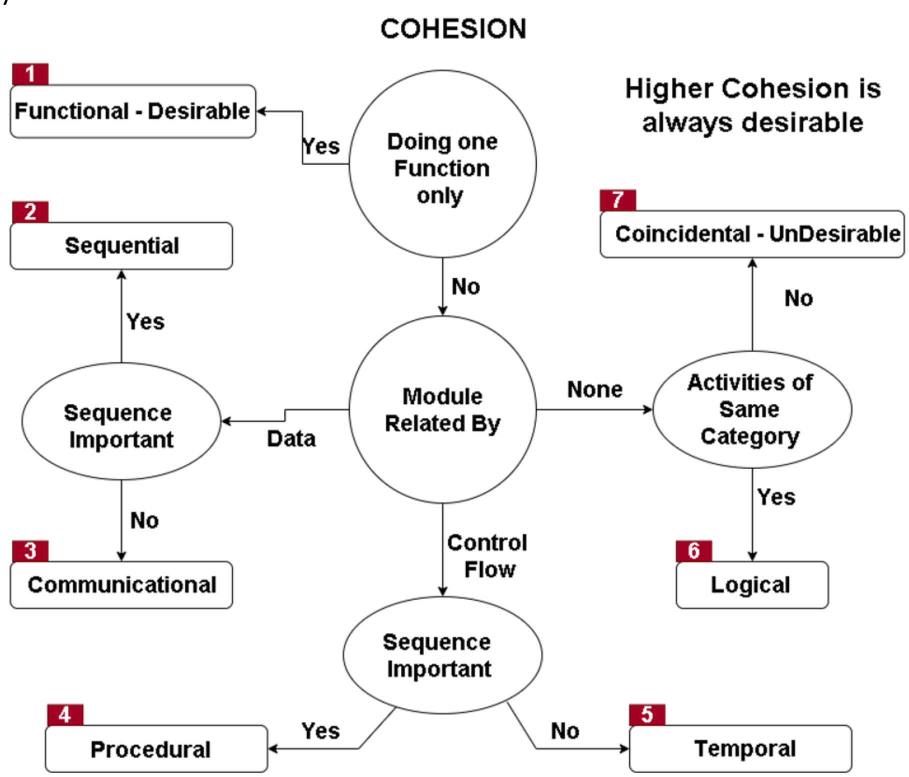
f. Logical Cohesion: A logical cohesive module is one where elements contribute to activities of the same general category
It basically use FLAG for switching and checking which task needs to be performed.
FLAGS ARE TO BE AVOIDED, BECAUSE IT ADDS COMPLEXTIY TO THE CODE



g. Coincidental Cohesion: Coincidental Cohesion implies little or no relationship among the statement of code within a procedure (a module).
E.g.



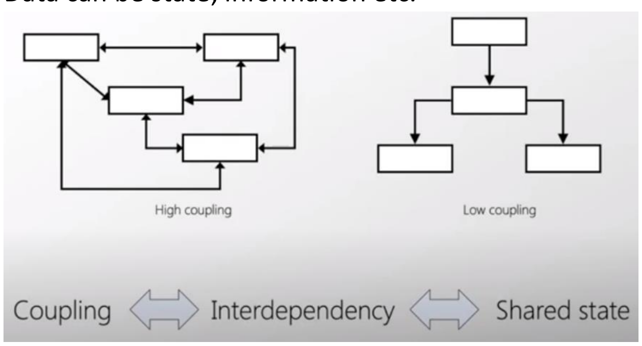
6. Easy way to understand Cohesion



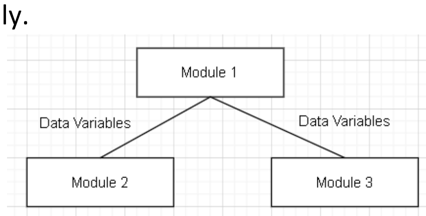
Intentionally left blank…

2. COUPLING

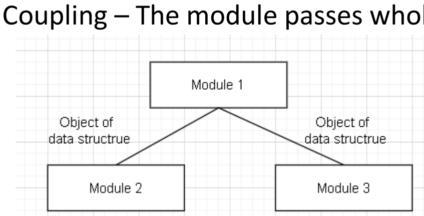
1. Coupling is the measure of degree of interdependencies between modules in the system
2. Coupling is the amount of the state information that modules share between them.
    a. More state information the module share, higher the coupling between them.
    b. High Coupling => Strongly inter-related/Dependents modules.
    c. Low Coupling => Interdependent Modules.
    d. Modules become coupled when it **shares data**, **exchanges data** or **they make function calls to each other.**
    e. Data can be state, information etc.



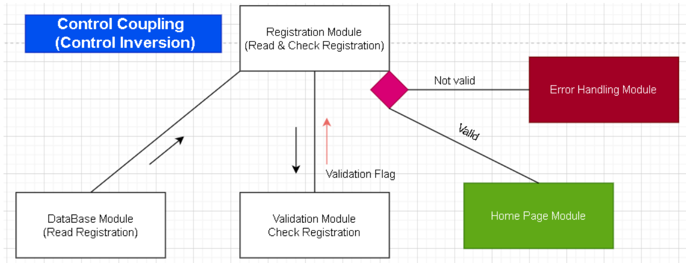
3. **LOW COUPLING IS DESIRED**
4. **HIGH COUPLING LEADS** to more **ERRORS** and IT makes **ISOLATION** (Debugging) of **ERRORS VERY Difficult**
5. Types of Coupling
    a. Data Coupling – Components are independent to each other, they communicate via data only.



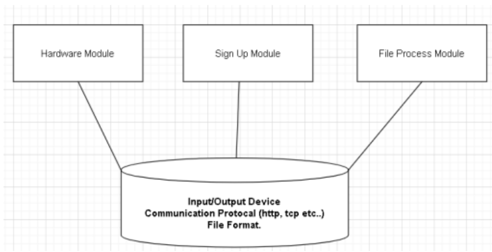
    b. Stamp Coupling – The module passes whole data structure (objects) to another module.



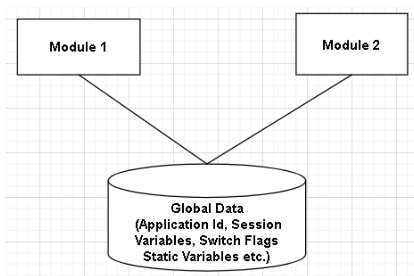
    c. Control Coupling (**control inversion**) – One module is controlling the flow of another module by passing it information on what to do. (e.g. passing a what to do flag)



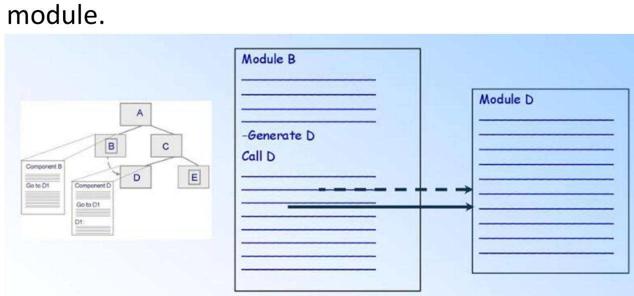
d. External Coupling –Two or more Modules share externally imposed data format, communication protocol or device interface.

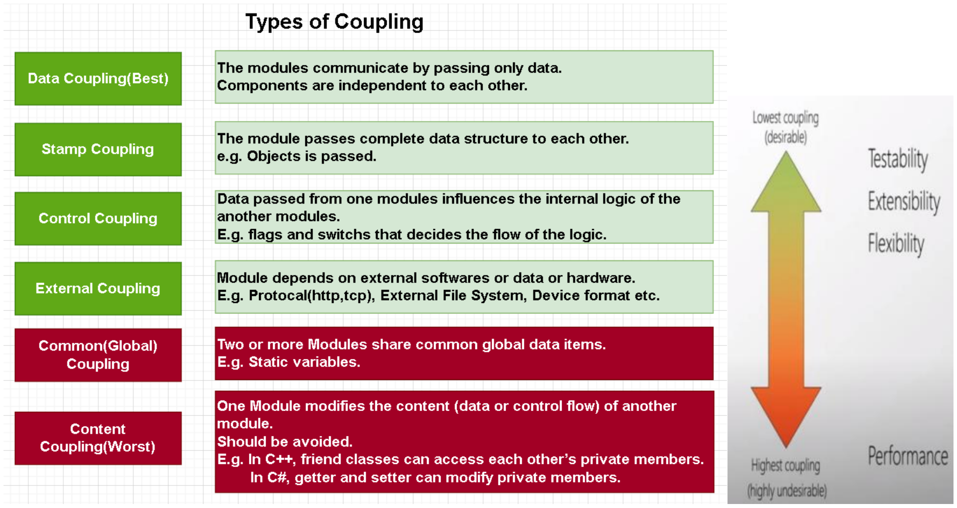


e. Common Coupling (global coupling) – when two modules share the same global data. Changing the shared resource might result in some unexpected result for other modules.



f. Content Coupling – When one module modifies an internal data item in Another module.





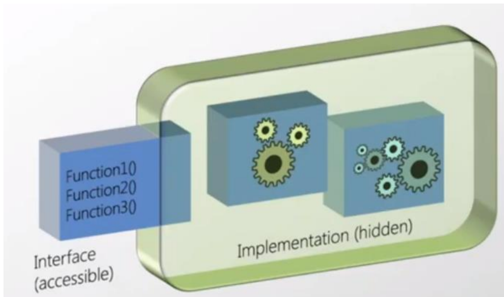
Left Blank intentionally

Reference:

- https://vimeo.com/244842060
- https://www.youtube.com/watch?v=-XJk27254KA&t=744s
- https://www.youtube.com/watch?v=lXUY_uTp6qg
- https://www.javatpoint.com/software-engineering-coupling-and-cohesion
- https://www.slideshare.net/MunaamMunawar/coupling-cohesion-and-there-types
- https://www.educative.io/edpresso/what-are-the-different-types-of-coupling
- https://www.youtube.com/watch?v=9VaZLEW0aTI
- https://www.youtube.com/watch?v=4L7Hj2WaKN8
- https://www.youtube.com/watch?v=Hsc02suLZ8s
- http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC333/Lectures/Design/cohesion.html

3. Encapsulation (Short Version)

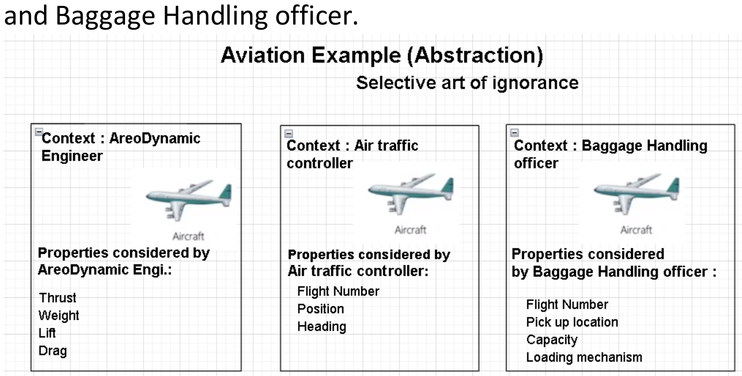
1. Encapsulation – Separating implementation from interface (accessible) and hide the implemented details (state information, internal processing of implementation (algorithm behavior)).



2. Benefits of hiding implementation details (state information, internal processing of implementation)
    a. If user cannot access the state information and implementation then user won't be able to couple it or depend upon it.
    b. Good encapsulation reduces the coupling between the system/components/modules.
        i. Easier to Test the system.
        ii. System becomes more flexible & extensible.

4. Abstraction(Short Version) - Context

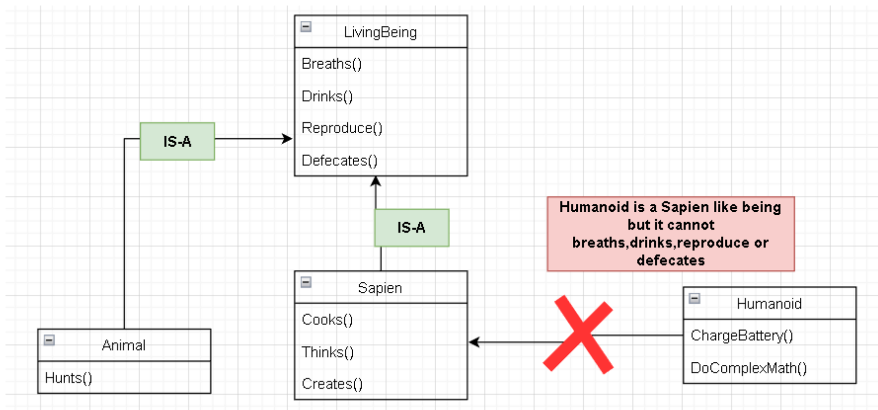
1. Art of Selective ignorance. i.e. Context
    a. User eating Burger King, User doesn't need to know from where the buns and patties come from or from where it has been bought.
    b. The guy assembling your burger king doesn't need to know the process of making the buns and patties. He just have to assemble it.
    c. So the context is important
2. Aviation example -
    a. Context of Aerodynamics Engineer for aircraft will be different than Air traffic officer and Baggage Handling officer.



LISKOV SUBSTITUTION PRINCIPLE(LSP)

[Note: Refer Modern Day Sapiens in examples as they are more evolved than Animals.]

1. LSP makes you follow "IS-A" relationship strictly between the entities.
   - The example of inheriting Humanoid from Sapiens is a bad idea because Humanoid is a being but it cannot Breath, Drink or Defecates. So this will violate LSP as "IS-A" rule has been broken.
   - On other hand, Sapiens & Animals are living being they Breath, Drink, Defecate and do other activities.



2. Liskov Substitution Principle –
   - Definition:

     Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.
   - Simplifying Definition: If S is a subtype of T, then any term of type S can be safely used in context where a term of type T is expected.
   - Normalizing Definition: **LSP** states that object of a super class (parent class) should be safely replaceable with the object of its subclass (child class) without breaking the application.
   - Normalizing Definition 2: When adding a new class in the hierarchy, the existing system shouldn't break when it uses the new class.
3. Liskov Substitution Principle & Inheritance
   - Inheritance means getting (inheriting) properties, functionalities from the parent class to child class.
   - Inheritance is easily over used or say abused by developers.
   - In some projects, the hierarchy of the classes is over 6 level deep and now it is normal to lose the track of what specific task, main parent class was performing.
   - **LSP** help the developers by limiting your use of inheritance by asking question i.e. Can you safely replace the object of parent class with child class without breaking or adding constraints (pre-conditions) in the child class? If YES than go for it and If NO then step back recheck inheritance implementation and high level abstraction for the application.
   - **LSP** is a gate keeper, which stop you from using inheritance wrongly and hence minimize the run-time error or run time surprises from happening.

4. LSP violation is hard to detect but when you detect a violation you should rethink the use of inheritance
   - Does the child class have **meaningful implementation** for all the overridden methods? If so, that's a very good thing.
   - Does the child classes overridden methods behaves differently from the parent class method? If yes then it is bad.
   - Would implementing an overridden method be out of scope and you were made to write throw not implemented exception in the method? If yes then it is bad.
   - Would implementing an overridden method ignore the calls and do nothing?
     - It is bad, if it is only one single method then it is ok if you can justify it.
     - It is very bad, if it has more than one single method doing nothing.
     - Below Page is left blank intentionally.

5. E.g. of Violation of LSP- Sapiens & Humanoid.

```
public class LivingBeing
{
    public virtual void Eat(string food) { Console.WriteLine("Eat " + food); }
    public virtual void Run() { Console.WriteLine("Run"); }
    public virtual void Sit() { Console.WriteLine("Sit"); }
    public virtual void Stand() { Console.WriteLine("Stand"); }
    public virtual void Defecation() { Console.WriteLine("Defecating"); }
}

public class Sapien : LivingBeing
{
    public virtual void Read(string bookName)
    { Console.WriteLine("Reading " + bookName); }
}

public class Animal : LivingBeing
{
    public virtual void Hunt(string preyName)
    { Console.WriteLine("Hunting " + preyName); }
}

public class Humaniod : Sapien
{
    //This were we are Breaking LSP.
    public override void Eat(string food){ ChargeTheHumanoid();}
    private void ChargeTheHumanoid(){Console.WriteLine("I am getting charged.");
}
```

```
public static class Test{
    public static void Check(LivingBeing livingBeing, string food)
    {
        livingBeing.Sit();
        livingBeing.Stand();
        livingBeing.Run();
        // This code will work but it will give you suprises since it violates LSP,
        // if object is Sapien then no issue sapien can eat, defecat
        // but if the object is humaniod then it doesn't eat food or defecat.
        // But if food is electricity humans die if they are exposed to it.
        livingBeing.Eat(food);
        livingBeing.Defecation();
        //All Living Being Cannot Read.. This Breaks LSP..
        //livingBeing.Read("The Orville List");
    }
}

void Main()
{
    Sapien sapiens = new Sapien();
    Test.Check(sapiens,"Pizza");
    Console.WriteLine("---------------");
    Humaniod humaniod = new Humaniod();
    Test.Check(humaniod,"");
    Console.WriteLine("---------------");
    Animal animal = new Animal();
    Test.Check(animal, "Meat");
}
```

Results λ SQL IL+Nat

```
Sit
Stand
Run
Eat Pizza
Defecating
---------------
Sit
Stand
Run
I am getting charged.
Defecating
---------------
Sit
Stand
Run
Eat Meat
Defecating
```

6. Improving the e.g. of Sapiens and Humanoid by implementing "**Interface Segregation Principle**".

```
public interface IStandardCommanActivy
{
    void Stand();
    void Sit();
    void Run();
}

public interface ILivingBeing : IStandardCommanActivy
{
    void Eat(string food);
    void Defecation();
}

public interface ISapien : ILivingBeing
{
    void Read(string bookName);
}

public interface IHumanoid : IStandardCommanActivy
{
    void charge(bool isPluggedIn);
}

public interface IAnimal : ILivingBeing
{
    void Hunt(string preyName);
}

public class Sapien : ISapien
{
    public virtual void Eat(string food) { Console.WriteLine("Eat " + food); }
    public virtual void Run() { Console.WriteLine("Run"); }
    public virtual void Sit() { Console.WriteLine("Sit"); }
    public virtual void Stand() { Console.WriteLine("Stand"); }
    public virtual void Defecation() { Console.WriteLine("Defecating"); }
    public virtual void Read(string bookName) { Console.WriteLine("Reading " + bookName); }
}
```

```
public class Animal : IAnimal
{
    public virtual void Hunt(string preyName)
    { Console.WriteLine("Hunting " + preyName); }
    public virtual void Eat(string food) { Console.WriteLine("Eat " + food); }
    public virtual void Run() { Console.WriteLine("Run"); }
    public virtual void Sit() { Console.WriteLine("Sit"); }
    public virtual void Stand() { Console.WriteLine("Stand"); }
    public virtual void Defecation() { Console.WriteLine("Defecating"); }
}

public class Humaniod : IHumanoid, IStandardCommanActivy
{
    public virtual void charge(bool isPluggedIn)
    {  Console.WriteLine("charging " + (isPluggedIn ? "Yes" : "No"));  }
    public virtual void Run() {Console.WriteLine("Run");}
    public virtual void Sit() { Console.WriteLine("Sit"); }
    public virtual void Stand() { Console.WriteLine("Stand"); }
}
```

```
public static class Test
{
    public static void CheckHumans(ISapien sapien, string food)
    {
        sapien.Sit();
        sapien.Read("The Orville List");
        sapien.Stand();
        sapien.Run();
        sapien.Eat(food);
        sapien.Defecation();
    }

    public static void CheckHumanoid(IHumanoid humanoid, bool isPluggedIn)
    {
        humanoid.Sit();
        humanoid.Stand();
        humanoid.Run();
        humanoid.charge(true);
    }

    public static void CheckAnimals(IAnimal animal, string food)
    {
        animal.Sit();
        animal.Stand();
        animal.Run();
        animal.Hunt("Deer");
        animal.Eat(food);
        animal.Defecation();
    }
}
```

```
void Main()
{
    ISapien sapien = new Sapien();
    Test.CheckHumans(sapien, "Pizza");
    Console.WriteLine("---------------");
    IHumanoid humaniod = new Humaniod();
    Test.CheckHumanoid(humaniod, true);
    Console.WriteLine("---------------");
    IAnimal animal = new Animal();
    Test.CheckAnimals(animal, "Meat");
}
```

Results λ SQL IL+Native Tree

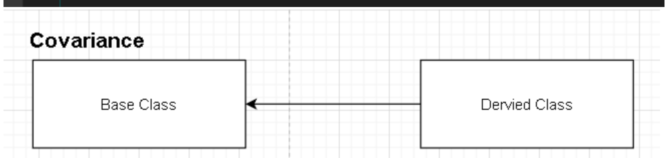
```
Sit
Reading The Orville List
Stand
Run
Eat Pizza
Defecating
---------------
Sit
Stand
Run
charging Yes
---------------
Sit
Stand
Run
Hunting Deer
Eat Meat
Defecating
```

7. One of the way to avoid violation of LSP is via Code Contract but mostly it is up to the programmer to think before apply inheritance left and right.

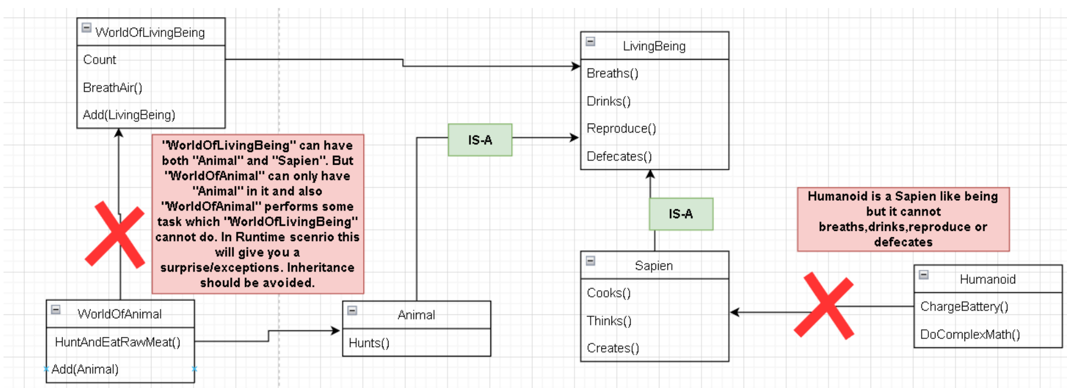
8. Covariance – Preserves assignment compatibility

- Covariance -> Enables you to pass or assign derived type where base type is expected.
  - If LivingBeing is Base Class & Animal is Derived Class then it is ok to assign object of animal to object of livingbeing.
  - In case of assigning List of Animals to IEnumerable of LivingBeing, The Generic IEnumerable interface should have a keyword “out” to signify that it is a Covariant Type

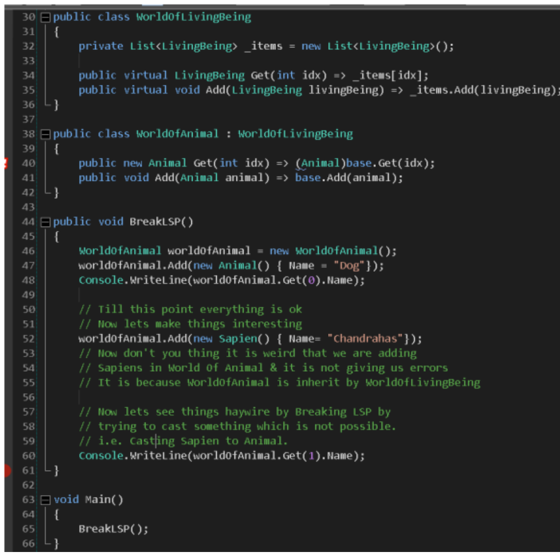
```
public void Example2(){
    LivingBeing livingBeing = new LivingBeing();
    Animal animal = new Animal();
    livingBeing = animal;

    List<Animal> animals = new List<UserQuery.Animal>();
    IEnumerable<LivingBeing> livingBeings = animals;
}
```



- Not an Ideal Situation of LSP

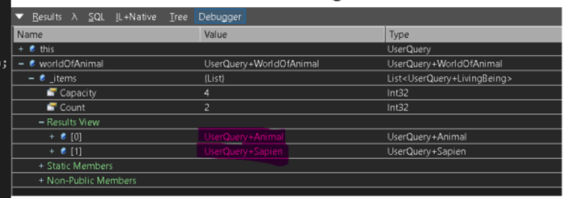


- We can have inheritance between “LivingBeing” and “Animal” because “IS-A” relationship can be formed i.e. Animal is a Living Being. Animal is a Living Being and can perform some additional task other than Breathing, Drinking & Reproducing.
- World of Living being can contain all type of living being such as Animals, Sapiens & others but World of Animal can only contain Animals. So if we are trying to inherit them we need to ask a question i.e.
  - **Is any special task “WorldOfAnimal” can perform but “WorldOfLivingBeing” cannot?**
    - **Yes, not all Living Being can hunt & eat raw meat.**
    - **Therefore, we should not inherit them.**
    - **We won’t be able to Cast Sapiens type to Animal type.**
    - **If in-case, we try to inherit them. We will break LSP.**

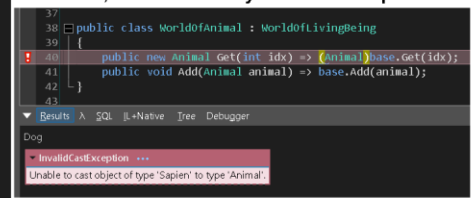


```
30 public class WorldOfLivingBeing
31 {
32     private List<LivingBeing> _items = new List<LivingBeing>();
33
34     public virtual LivingBeing Get(int idx) => _items[idx];
35     public virtual void Add(LivingBeing livingBeing) => _items.Add(livingBeing);
36 }
37
38 public class WorldOfAnimal : WorldOfLivingBeing
39 {
40     public new Animal Get(int idx) => (Animal)base.Get(idx);
41     public void Add(Animal animal) => base.Add(animal);
42 }
43
44 public void BreakLSP()
45 {
46     WorldOfAnimal worldOfAnimal = new WorldOfAnimal();
47     worldOfAnimal.Add(new Animal() { Name = "Dog"});
48     Console.WriteLine(worldOfAnimal.Get(0).Name);
49
50     // Till this point everything is ok
51     // Now lets make things interesting
52     worldOfAnimal.Add(new Sapien() { Name= "Chandrahas"});
53     // Now don't you thing it is weird that we are adding
54     // Sapiens in World Of Animal & it is not giving us errors
55     // It is because WorldOfAnimal is inherit by WorldOfLivingBeing
56
57     // Now lets see things haywire by Breaking LSP by
58     // trying to cast something which is not possible.
59     // i.e. Casting Sapien to Animal.
60     Console.WriteLine(worldOfAnimal.Get(1).Name);
61 }
62
63 void Main()
64 {
65     BreakLSP();
66 }
```

Till Line Number 52 everthing seems ok.

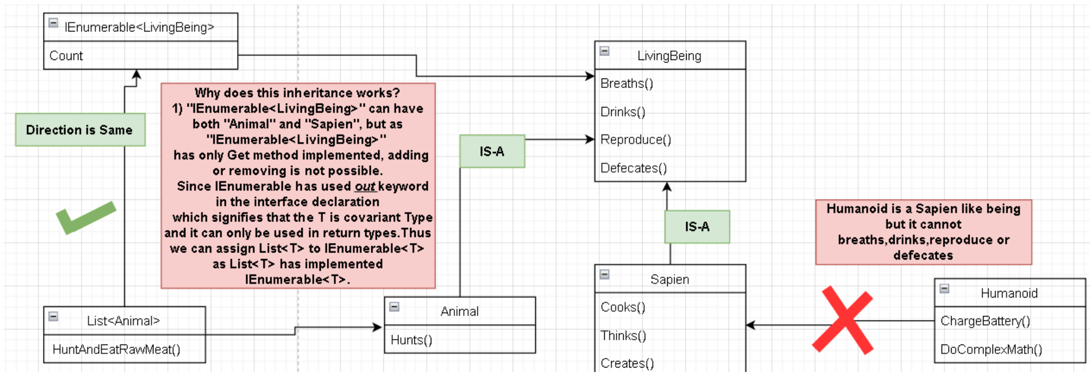


Line 60, When we try to Cast Sapien to Animal



- In the above example, we are hiding the base class method "Get" by using new keyword and **overriding** the "Get" method **entirely** in the child class according to our convenience. This breaks the LSP in **Line 40.**
- Hence it is Said Breaking LSP will give you surprises i.e. run time exception. We were able to compile it but we got an un-excepted runtime exception.
- The above example breaks the LSP. So if we don't think in the perspective of class and segregate the common functionalities into small interfaces then we can achieve inheritance between them and that to without breaking the LSP.

- **Example of IEnumerable<LivingBeing> & List<Animal>**




```
30 public void WithStandardGeneric()
31 {
32     List<Animal> worldOfAnimals = new List<UserQuery.Animal>(){
33         new Animal(){ Name = "Deer"},
34         new Animal(){ Name = "Snake"},
35         new Animal(){ Name = "Dog"},
36     };
37     IEnumerable<LivingBeing> worldOfLivingBeings;
38
39     //Here we are creating inheritance between
40     //IEnumerable<LivingBeing> and List<Animal>
41     // by assigning worldOfAnimals to worldOfLivingBeings
42     //Since Animal is dervied class & LivingBeing is base Class
43     worldOfLivingBeings = worldOfAnimals;
44
45     //Now since worldOfLivingBeings has the reference of worldOfAnimals
46     //Below line 48 & 49 will be execute without any issue.
47     foreach (var being in worldOfLivingBeings)
48         Console.WriteLine(being.Name);
49
50     //We can add more animal type in worldOfAnimals
51     worldOfAnimals.Add(new Animal {Name = "Cat"});
52
53     Console.WriteLine("--------");
54     foreach (var being in worldOfLivingBeings)
55         Console.WriteLine(being.Name);
56
57     // Below line will gives us error since we won't be able to
58     // Cast Sapien to Animal Type.
59     worldOfAnimals.Add(new Sapien {Name = "Chandrahas"});
60
61     // Below Line will gives us error
62     // Since worldOfLivingBeings is an IEnumerable
63     // We cannot add or remove from worldOfLivingBeings.
64     worldOfLivingBeings.Add(new Sapien {Name = "Ptomey"});
65 }
```

For Error free execution, we should comment lines 59 & 64.

```
57     // Below line will gives us error since we won't be able to
58     // Cast Sapien to Animal Type.
59     //worldOfAnimals.Add(new Sapien {Name = "Chandrahas"});
60
61     // Below Line will gives us error
62     // Since worldOfLivingBeings is an IEnumerable
63     // We cannot add or remove from worldOfLivingBeings.
64     //worldOfLivingBeings.Add(new Sapien {Name = "Ptomey"});
65 }
66
67
68 void Main()
69 {
70     //BreakLSP();
71     WithStandardGeneric();
72 }
73
```

Results λ SQL IL+Native Tree

```
Deer
Snake
Dog
--------
Deer
Snake
Dog
Cat
```

- So as you have seen we are not able to add anything other than Animal in List<Animal> and also we are not able to add anything in IEnumerable of LivingBeing. Hence we are not violating the LSP.
- The Line 43 i.e. where we say IEnumerable<LivingBeing> = List<Animal> we are creating the reference of List of Animal in IEnumerable of LivingBeing and that is perfectly valid. Since Animal is inherited by LivingBeing. There is one important reason hidden in IEnumerable interface.

- How we were able to assign List of Animal to IEnumerable of LivingBeing in other words we are able to create inheritance between them should be the question asked?
    - For that we need to check IEnumerable Interface.

```
IEnumerable<T>
// System.Collections.Generic.IEnumerable<T>
using System.Collections;
using System.Collections.Generic;

public interface IEnumerable<out T> : IEnumerable
{
    new IEnumerator<T> GetEnumerator();
}
```

```
[Serializable]
[DebuggerTypeProxy(typeof(ICollectionDebugView<>))]
[DebuggerDisplay("Count = {Count}")]
[TypeForwardedFrom("mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089")]
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IEnumerable, IList, ICollection, IReadOnlyList<T>, IReadOnlyCollection<T>
{
```

    - In generic declaration, we have keyword **out** and it is the reason we were able to inherit or assign List of Animal to IEnumerable of LivingBeing.
    - It tells that, if Derived class can be assigned to Base Class, then Generic Derived class can also be assigned to Generic interface of Base class
        - If generic interface have prefixed **out** keyword while declaring generic type parameter
            - e.g. public interface IWorld<out T> {}
        - If Generic Derived Class has implemented Generic interface
            - e.g. public class World<T> : IWorld <T>,…{}

> **A generic type G<T> is covariant with respect to T if G<X> is convertible to G<Y> given that X is convertible to Y**

    - Let's see example for Living Being & Animal with Homemade Generic.

```
// out make the T as covariant parameter.
// The out keyword can only be defined in Generic Interfaces
// The out keyword tell that it can be used only for return types.
// If we remove the out keyword we will loose the ability of inherit.
public interface IWorld<out T>
{
    T Get(int idx);
    IEnumerable<T> GetAll();
}

public class World<T> : IWorld<T>
{
    private List<T> _items = new List<T>();

    public T Get(int idx) => _items[idx];
    public IEnumerable<T> GetAll() => _items;
    public void Add(T t) => _items.Add(t);
}
```

```
Results λ
Deer
Snake
Dog
--------
Deer
Snake
Dog
Cat
```

Results λ SQL IL+Native Tree Debugger

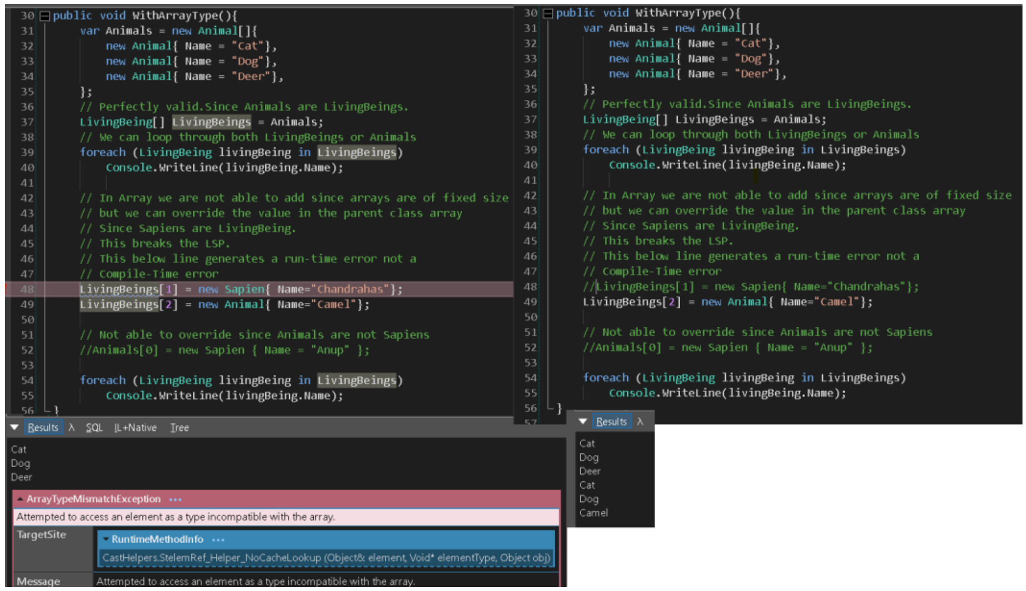
Name	Value	Type
+ this		UserQuery
+ worldOfAnimals	UserQuery+World`1[UserQuery+Animal]	UserQuery+World<UserQu
− worldOfLivingBeings	UserQuery+World`1[UserQuery+Animal]	UserQuery+World<UserQu
− _items	{List}	List<UserQuery+Animal>
Capacity	4	Int32
Count	4	Int32
− Results View		
− [0]	UserQuery+Animal	UserQuery+Animal
Name	"Deer"	String
+ [1]	UserQuery+Animal	UserQuery+Animal
+ [2]	UserQuery+Animal	UserQuery+Animal
+ [3]	UserQuery+Animal	UserQuery+Animal
+ Static Members		

```
public void HomeMadeGeneric()
{
    World<Animal> worldOfAnimals = new World<Animal>();
    worldOfAnimals.Add(new Animal(){ Name = "Deer"});
    worldOfAnimals.Add(new Animal(){ Name = "Snake"});
    worldOfAnimals.Add(new Animal(){ Name = "Dog"});

    IWorld<LivingBeing> worldOfLivingBeings;
    // This below is working since we have declared T as Covariant
    // in IWorld interface and World<T> has implemented the IWorld
    // interface.
    worldOfLivingBeings = worldOfAnimals;

    //Now since worldOfLivingBeings has the reference of worldOfAnimals
    //Below line 48 & 49 will be execute without any issue.
    foreach (var being in worldOfLivingBeings.GetAll())
        Console.WriteLine(being.Name);

    //We can add more animal type in worldOfAnimals
    worldOfAnimals.Add(new Animal {Name = "Cat"});

    Console.WriteLine("--------");
    foreach (var being in worldOfLivingBeings.GetAll())
        Console.WriteLine(being.Name);

    // Below line will gives us error since we won't be able to
    // Cast Sapien to Animal Type.
    //worldOfAnimals.Add(new Sapien {Name = "Chandrahas"});

    // Below Line will gives us error
    // Since worldOfLivingBeings is an IWorld
    // which don't have add or remove capablities.
    //worldOfLivingBeings.Add(new Sapien {Name = "Ptomey"});
}
```

- So **out** keyword mentioned in the interface tells us that this interface is has a **Covariant Type.**
- The "T" in the covariant interfaces can only be used in the methods and properties with return type. It cannot be used as method parameters such as Add(T obj), if we have mentioned **out** keyword in the generic interface declaration.
- So this means we can use this type of interfaces only for **reading** the data not for **writing** the data.
- **"out"** keyword can only be used in generic interface declaration.
- If we removing **out** keyword from the generic interface declaration then
    - We will be able to implement the interface with class.
    - We won't be able to form the inheritance relationship between the generic interface and the class to which it has been implemented.
    - We won't be able to assign the class to the interface it will give us compile time error as shown below.

```csharp
public interface IWorld<T>
{
    T Get(int idx);
    IEnumerable<T> GetAll();
}

public class World<T> : IWorld<T>
{
    private List<T> _items = new List<T>();

    public T Get(int idx) => _items[idx];
    public IEnumerable<T> GetAll() => _items;
    public void Add(T t) => _items.Add(t);
}

public void HomeMadeGeneric()
{
    World<Animal> worldOfAnimals = new World<Animal>();
    worldOfAnimals.Add(new Animal(){ Name = "Deer"});
    worldOfAnimals.Add(new Animal(){ Name = "Snake"});
    worldOfAnimals.Add(new Animal(){ Name = "Dog"});

    IWorld<LivingBeing> worldOfLivingBeings;
    // This below is working since we have declared T as Covariant
    // in IWorld interface and World<T> has implemented the IWorld
    // interface.
    worldOfLivingBeings = worldOfAnimals;
```

- In primitive type **Array**, Covariance doesn't seems to work perfectly though we cannot add a new value in array as it is of fixed size. We can override a value in array and this might lead to runtime error.
- For Example

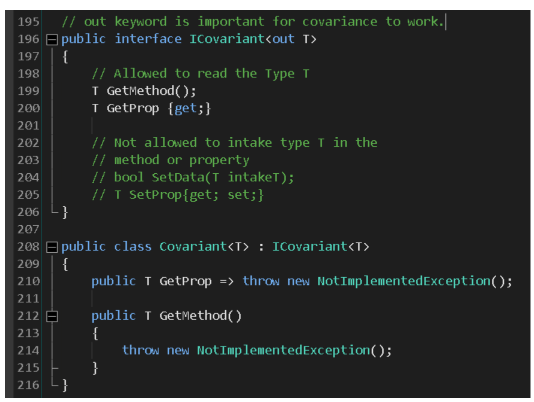


```csharp
public void WithArrayType(){
    var Animals = new Animal[]{
        new Animal{ Name = "Cat"},
        new Animal{ Name = "Dog"},
        new Animal{ Name = "Deer"},
    };
    // Perfectly valid.Since Animals are LivingBeings.
    LivingBeing[] LivingBeings = Animals;
    // We can loop through both LivingBeings or Animals
    foreach (LivingBeing livingBeing in LivingBeings)
        Console.WriteLine(livingBeing.Name);

    // In Array we are not able to add since arrays are of fixed size
    // but we can override the value in the parent class array
    // Since Sapiens are LivingBeing.
    // This breaks the LSP.
    // This below line generates a run-time error not a
    // Compile-Time error
    LivingBeings[1] = new Sapien{ Name="Chandrahas"};
    LivingBeings[2] = new Animal{ Name="Camel"};

    // Not able to override since Animals are not Sapiens
    //Animals[0] = new Sapien { Name = "Anup" };

    foreach (LivingBeing livingBeing in LivingBeings)
        Console.WriteLine(livingBeing.Name);
}
```



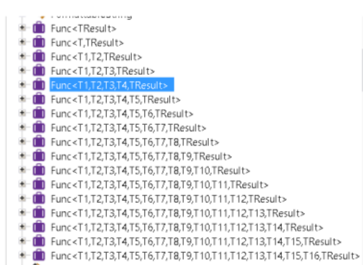
```csharp
public void WithArrayType(){
    var Animals = new Animal[]{
        new Animal{ Name = "Cat"},
        new Animal{ Name = "Dog"},
        new Animal{ Name = "Deer"},
    };
    // Perfectly valid.Since Animals are LivingBeings.
    LivingBeing[] LivingBeings = Animals;
    // We can loop through both LivingBeings or Animals
    foreach (LivingBeing livingBeing in LivingBeings)
        Console.WriteLine(livingBeing.Name);

    // In Array we are not able to add since arrays are of fixed size
    // but we can override the value in the parent class array
    // Since Sapiens are LivingBeing.
    // This breaks the LSP.
    // This below line generates a run-time error not a
    // Compile-Time error
    //LivingBeings[1] = new Sapien{ Name="Chandrahas"};
    LivingBeings[2] = new Animal{ Name="Camel"};

    // Not able to override since Animals are not Sapiens
    //Animals[0] = new Sapien { Name = "Anup" };

    foreach (LivingBeing livingBeing in LivingBeings)
        Console.WriteLine(livingBeing.Name);
}
```


  - Example of How to Declare Covariant interface & Implement Covariance interface



```csharp
// out keyword is important for covariance to work.
public interface ICovariant<out T>
{
    // Allowed to read the Type T
    T GetMethod();
    T GetProp {get;}

    // Not allowed to intake type T in the
    // method or property
    // bool SetData(T intakeT);
    // T SetProp{get; set;}
}

public class Covariant<T> : ICovariant<T>
{
    public T GetProp => throw new NotImplementedException();

    public T GetMethod()
    {
        throw new NotImplementedException();
    }
}
```

  - **Func<out TResult> is Covariant.**

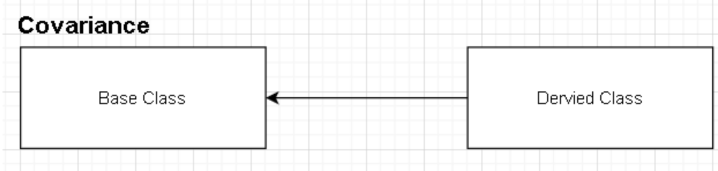


If you see the Func i.e., generic delegate in c#. The return type is declared with "out" keyword i.e., the return type is covariant.

```csharp
// System.Func<T1,T2,T3,T4,TResult>
public delegate TResult Func<in T1, in T2, in T3, in T4, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4);
```

9. Contravariance – Reverses the assignment compatibility

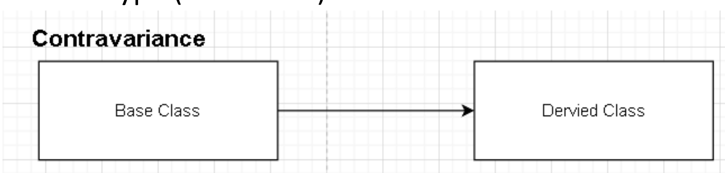
- So in covariance, we have seen that we can assign derived type to base type.



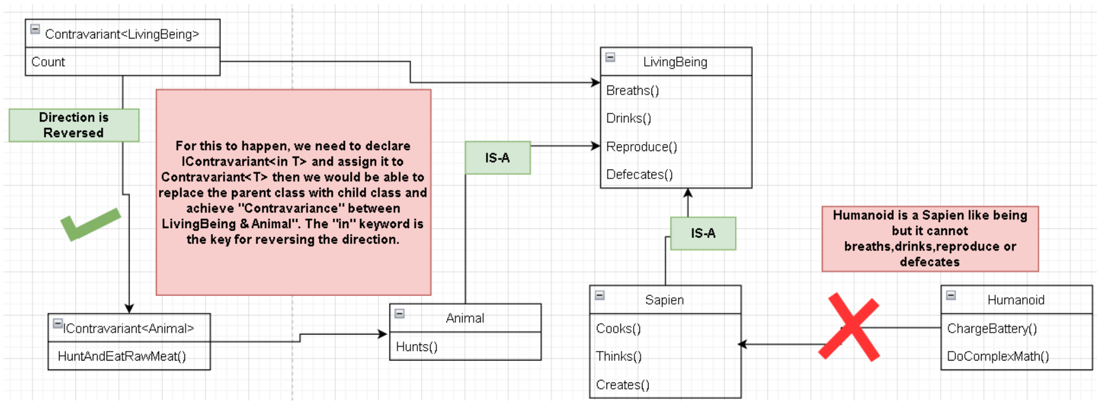

```csharp
// out keyword is important for covariance to work.
public interface ICovariant<out T>
{
    // Allowed to read the Type T
    T GetMethod();
    T GetProp {get;}

    // Not allowed to intake type T in the
    // method or property
    // bool SetData(T intakeT);
    // T SetProp{get; set;}
}

public class Covariant<T> : ICovariant<T>
{
    public T GetProp => throw new NotImplementedException();

    public T GetMethod()
    {
        throw new NotImplementedException();
    }
}

public void SimpleExample(){
    // This is just a example to show you
    ICovariant<Animal> covariantAnimals = new Covariant<Animal>();
    ICovariant<Sapien> covariantSapien = new Covariant<Sapien>();

    ICovariant<LivingBeing> covariantLivingBeing = covariantAnimals;
    //or
    covariantLivingBeing = covariantSapien;
}
```

```csharp
public void Example2(){
    LivingBeing livingBeing = new LivingBeing();
    Animal animal = new Animal();
    livingBeing = animal;

    List<Animal> animals = new List<UserQuery.Animal>();
    IEnumerable<LivingBeing> livingBeings = animals;
}
```

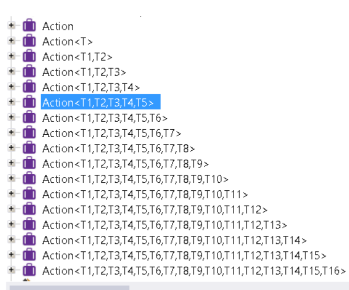
- In Contravariance, we do opposite i.e. we can assign base type (Parent class) to derived type (Child class).




```csharp
// in keyword is important for contra-variant to work.
public interface IContraVariant<in T>
{
    // Not Allowed to read the Type T
    //T GetMethod();
    //T GetProp { get; }

    // Allowed to intake type T in the
    // method or property
     bool SetData(T intakeT);
     T SetProp{ set;}
}

public class ContraVariant<T> : IContraVariant<T>
{
    public T SetProp { set => throw new NotImplementedException(); }

    public bool SetData(T intakeT)
    {
        throw new NotImplementedException();
    }
}

public void SimpleExample(){
    // This is just a example to show you
    IContraVariant<Animal> contraVariantAnimals = new ContraVariant<Animal>();
    IContraVariant<Sapien> contraVariantSapien = new ContraVariant<Sapien>();
    IContraVariant<LivingBeing> contraVariantLivingBeing = new ContraVariant<LivingBeing>();

    //Assignment is reverse from higer to lower..
    IContraVariant<Animal> contraVariantAnimal_2 = contraVariantLivingBeing;
    //or
    IContraVariant<Sapien> contraVariantSapien_2 = new ContraVariant<LivingBeing>();
}
```

- In above example,
  - We can see that we are able to assign object of Contravariant of LivingBeing to IContravariant of Animal.
  - We are allowed to assign instantiated interface of base class to the interface of derived class.
    i.e. IContravariant<DerivedClass> objectName = new Contravariant<BaseClass>();
    OR
    IContravariant<BaseClass> baseClassObjName = new Contravariant<BaseClass>();
    IContravariant<DerivedClass> objectName = baseClassObjName;
  - 

- When we define only Contravariant interface it means
  - We cannot have a method that returns T in the method signature
    i.e. T Get(int x); is not valid.
  - We can have a method that take T as input parameter.
    i.e. void Set(T t); is valid.
  - For Properties, we can only have set only properties.
- Normally, Contravariant interface are used only for writing the data not reading the data.
- **Action<in TInput1>** is **Contravariant**.



If you see the Action i.e. Generic Delegate in C#
The input types is declared with "in" keyword.
i.e. the input parameter is Contravariant

```
// System.Action<T1,T2,T3,T4,T5>
public delegate void Action<in T1, in T2, in T3, in T4, in T5>(T1 arg1, T2 arg2, T3 arg3, T4 arg4, T5 arg5);
```

- Let’s understand Contravariant via code, this example is transformed from “Coding Tutorials” video

```csharp
81  /// <summary>
82  /// If we remove "in" keyword then the complier will not
83  /// be able to identify that T is contravariant.
84  /// then this code is not possible
85  /// Sort(sapiens, new LivingBeingComparer());
86  /// </summary>
87  public interface IMyComparer<in T>
88  {
89      int Compare(T object1, T object2);
90  }
91
92  public class LivingBeingComparer : IMyComparer<LivingBeing>
93  {
94      // Compare weight between LivingBeing
95      public int Compare(LivingBeing object1, LivingBeing object2)
96      {
97          return (int)(object1.Weight - object2.Weight);
98      }
99  }
100
101 public class SapienComparer : IMyComparer<Sapien>
102 {
103     // If we apply "in" keyword in IMyComparer,
104     // then this code is redundant.
105     public int Compare(Sapien object1, Sapien object2)
106     {
107         return (int)(object1.Weight - object2.Weight);
108     }
109 }
111 public class CompareAdapter<T> : IComparer<T>
112 {
113     private IMyComparer<T> _innerComparer;
114     public CompareAdapter(IMyComparer<T> innerComparer)
115     {
116         _innerComparer = innerComparer;
117     }
118
119     public int Compare([AllowNull]T x,[AllowNull] T y)
120     {
121         return _innerComparer.Compare(x, y);
122     }
123 }
```

```csharp
125 public class Program{
126     private static void SortWithInterface(){
127         List<Sapien> sapiens = new List<Sapien>(){
128             new Sapien(){ Name = "Onkar", Weight=110},
129             new Sapien(){ Name = "CJ", Weight=66},
130             new Sapien(){ Name = "Govind", Weight=64 },
131             new Sapien(){ Name = "Sandeep", Weight=98}
132         };
133
134         foreach (var sapien in sapiens)
135             Console.WriteLine($"{sapien.Name}-{sapien.Weight}");
136
137         Console.WriteLine();
138
139         //Sort(sapiens, new SapienComparer());
140
141         Sort(sapiens, new LivingBeingComparer());
142
143         foreach (var sapien in sapiens)
144             Console.WriteLine($"{sapien.Name}-{sapien.Weight}");
145     }
146
147     // So as you can see that in Line no. 142, we are providing LivingBeingComparer
148     // i.e. IMyComparer<LivingBeing> in place of SapienComparer i.e. IMyComparer<Sapien>
149     // but it is not giving any errors. We are trying to replace the child class
150     // with the parent class i.e. we are trying to achieve Contravariance but that
151     // wouldn't have been possible if we haven't added "in" keyword in the
152     // interface IMyComparer<in T>
153     private static void Sort(List<Sapien> collection, IMyComparer<Sapien> comparer){
154         collection.Sort(new CompareAdapter<Sapien>(comparer));
155     }
156
157     public static void Main(){
158         SortWithInterface();
159     }
160 }
```

```
Onkar-110
CJ-66
Govind-64
Sandeep-98

Govind-64
CJ-66
Sandeep-98
Onkar-110
```

```csharp
IComparer<T>
// System.Collections.Generic.IComparer<T>
public interface IComparer<in T>
{
    int Compare(T? x, T? y);
}
```

Check the "in" keyword in the IComparer in the .net library.

- So if we remove the “**in**” keyword from line no. 87, we won’t be able to replace the parent class with the child class i.e. Contravariance won’t have been possible.
- So simply applying the “in” keyword helps us to replace parent class with the child class without breaking the application. [“**HAHAHAHA**” inside joke Hai.]

- Let’s look at one more code example.

```
187 public class Program2{
188     private static void SortWithDelegate()
189     {
190         List<Sapien> sapiens = new List<Sapien>(){
191             new Sapien(){ Name = "Onkar", Weight=110},
192             new Sapien(){ Name = "CJ", Weight=66},
193             new Sapien(){ Name = "Govind", Weight=64 },
194             new Sapien(){ Name = "Sandeep", Weight=98}
195         };
196
197         foreach (var sapien in sapiens)
198             Console.WriteLine($"{sapien.Name}-{sapien.Weight}");
199
200         Console.WriteLine();
201
202         //SortingDelegate<Sapien> sapienSD = (o1w, o2w) => (int)(o1w.Weight - o2w.Weight);
203         //Sort(sapiens, sapienSD);
204
205         SortingDelegate<LivingBeing> livingBeingSD = (o1w, o2w) => (int)(o1w.Weight - o2w.Weight);
206         Sort(sapiens, livingBeingSD);
207
208         foreach (var sapien in sapiens)
209             Console.WriteLine($"{sapien.Name}-{sapien.Weight}");
210     }
211
212     // So now you won't be surpise to see that we are able to replace Child class with
213     // parent class in line 206, because of the magic keyword "in" which let the compilier
214     // know that it is contravariant.
215     private static void Sort(List<Sapien> collection, SortingDelegate<Sapien> sorter)
216         => collection.Sort((o1w, o2w) => sorter(o1w, o2w));
217
218     public static void Main()
219     {
220         SortWithDelegate();
221     }
222 }
```

```
Comparison<T>
// System.Comparison<T>
public delegate int Comparison<in T>(T x, T y);
```

```
Sort(Comparison<T>) : void
// System.Collections.Generic.List<T>
public void Sort(Comparison<T> comparison)
{
```

```
224 // Contravariance
225 // if we don't add "in" keyword then line no. 206 will not be possible.
226 public delegate int SortingDelegate<in T>(T object1Weight, T object2Weight);
227
```

- Generic delegates such as Func, Action & Predicate already have **Covariance (out** keyword) & **Contravariance (in** keyword).

Func delegate – Covariance (out) & Contravariance (in).

```
Func<T1,T2,TResult>
// System.Func<T1,T2,TResult>
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
```

Action delegate – Contravariance (in).

```
Action<T1,T2>
// System.Action<T1,T2>
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
```

Predicate delegate – Contravariance (in).

```
Predicate<T>
// System.Predicate<T>
public delegate bool Predicate<in T>(T obj);
```

10. Mathematical understanding of Covariance, Contravariance & Invariance.

- What is Projection?
  - A projection is function which takes a single integer and returns a new integer.
  - For Example
    - If z is passed to “Double” projection function then z + z will be returned.
    - If z is passed to “Negate” projection function then 0 – z will be returned.
    - If z is passed to “Square” projection function then z * z will be returned.
- Covariance
  - Let us evaluate
    - **(x <= y) = (Double(x) <= Double(y)) -> Yes**
    - Replace x = 1 & y = 2 then

      (1 <= 2) = ((1 +1) <= (2 + 2))
      (1 <= 2) = (2 <= 4)
      True = True
    - Now Let’s replace x = 2 & y = 2 then

      (2 <= 2) = ((2 + 2) <= (2 + 2))
      (2 <= 2) = (4 <= 4)
      True = True
    - Now let’s replace x = 4 & y = 2 then

      (4 <= 2) = ((4 + 4) <= (2+2))
      (4 <= 2) = (8 <= 4)
      False = False
    - **So as you can see Projection Double perverse the direction of size.**
- **Contravariance**
  - Let us evaluate
    - **(x <= y) = (Negate(x) <= Negate(y)) -> No**
    - Replace x = 1 & y = 2 then

      (1 <= 2) = ((0-1) <= (0-2))
      (1 <= 2) = (-1 <= -2)
      True = False
    - Let’s replace x = 2 & y = 1 then

      (2 <= 1) = ((0-2) <= (0-1))
      (2 <= 1) = (-2 <= -1)
      False = True
    - **Let’s Reverse (x <= y) = (Negate(y) <= Negate(x)) -> Yes**
    - Replace x = 1 & y = 2 then

      (1 <= 2) = ((0-2) <= (0-1))
      (1 <= 2) = (-2 <= -1)
      True = True
    - Let’s replace x = 2 & y = 1 then

      (2 <= 1) = ((0-1) <= (0-2))
      (2 <= 1) = (-1 <= -2)
      False = False

        - **So as you can see when Projection Negate was reversed then both left & right become equal i.e. (1<=2) = True & also (-2<=-1) = True hence we can say then Projection Negate reverse's the direction of size.**
- **Invariant**
    - Let us evaluate
        - **(x <= y) = (Square(x) <= Square(y)) -> No**
        - Replace x = -1 & y = 0 then

          (-1 <= 0) = ((-1*-1) <= (0*0))

          (-1 <= 0) = (1 <= 0)

          True = False
        - **Let's reverse (x <= y) = (Square(y) <= Square(x)) -> No**
        - Replace x = 1 & y = 2 then

          (1 <= 2) = ((2 * 2) <= (1 * 1))

          (1 <= 2) = (4 <= 1)

          True = False
        - **So as you can see Projection Square was not able to preserve nor reverse the direction of the size.**
- Source : Wikipedia

Within the type system of a programming language, a typing rule or a type constructor is:

- *covariant* if it preserves the ordering of types (≤), which orders types from more specific to more generic: If `A ≤ B`, then `I<A> ≤ I<B>`;
- *contravariant* if it reverses this ordering: If `A ≤ B`, then `I<B> ≤ I<A>`;
- *bivariant* if both of these apply (i.e., if `A ≤ B`, then `I<A> ≡ I<B>`);[1]
- *variant* if covariant, contravariant or bivariant;
- *invariant* or *nonvariant* if not variant.

References:

- https://reflectoring.io/lsp-explained/ - very well explained..
- https://blog.knoldus.com/what-is-liskov-substitution-principle-lsp-with-real-world-examples/
- https://www.youtube.com/watch?v=x82wNJTzu28
- https://openclassrooms.com/en/courses/5684096-use-mvc-solid-principles-and-design-patterns-in-java/6417806-l-for-the-liskov-substitution-principle
- https://www.youtube.com/watch?v=oeknOtb0gzQ
- https://www.codeproject.com/Articles/648987/Violating-Liskov-Substitution-Principle-LSP
- https://www.codeproject.com/Articles/1084933/Detecting-Liskov-Substitution-Principle-Violations
- https://www.netguru.com/blog/solid-principles-3-lsp
- https://docs.microsoft.com/en-us/archive/msdn-magazine/2011/july/msdn-magazine-cutting-edge-code-contracts-inheritance-and-the-liskov-principle
- https://www.youtube.com/watch?v=oLUkB076yz0
- https://www.youtube.com/watch?v=S81Xl8sATBI
- https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts
- https://docs.microsoft.com/en-us/dotnet/csharp/nullable-references
- https://www.cs.unh.edu/~cs619/slides/DesignByContract.pdf
- https://learn.microsoft.com/en-us/archive/blogs/ericlippert/whats-the-difference-between-covariance-and-assignment-compatibility - Both explained in terms of math's.
- https://www.youtube.com/watch?v=Wp5iYQqHspg
- https://www.youtube.com/watch?v=LsKIhuRJbtk
- Covariance
  - https://www.youtube.com/watch?v=DOCNiTEiaBk&list=PLQB-TSatJvw5cwcKKecTG4EwfhZcbAkm0&index=3
  - https://www.youtube.com/watch?v=3FTvHnhmd88
  - https://levelup.gitconnected.com/covariance-and-contravariance-in-net-c-c2b8576b2155 -
  - https://vkontech.com/a-practical-intro-to-covariance-and-contravariance-in-c/
- Contravariance
  - https://www.youtube.com/watch?v=OBrS1L0HIcs

OCP - OPEN CLOSE PRINICPLE

1. T

Reference:

- https://levelup.gitconnected.com/4-ways-to-achieve-open-closed-principle-in-c-d2d0ebea4b86
- https://blogs.siliconorchid.com/post/coding-inspiration/implementing-open-closed/
- https://codeblog.jonskeet.uk/2013/03/15/the-open-closed-principle-in-review/
- https://learnwithshahriar.wordpress.com/2015/12/09/solid-principle-ocp/
- https://www.youtube.com/watch?v=En9xIW5LAzg
- https://www.youtube.com/watch?v=oxRd4cHZqwc
- https://www.youtube.com/watch?v=zpgOpegZXJg
- https://www.youtube.com/watch?v=fspR_pJanOg
- https://www.youtube.com/watch?v=J55tD3QPAQY

GRASP Pattern

Reference:

- https://www.youtube.com/watch?v=oU54ETOBcxU&t=226s

Some Software engineering Principles you would require to know.

Reference:

- https://methodpoet.com/top-software-engineering-principles/#SOLID_principles
- https://methodpoet.com/worst-anti-patterns/