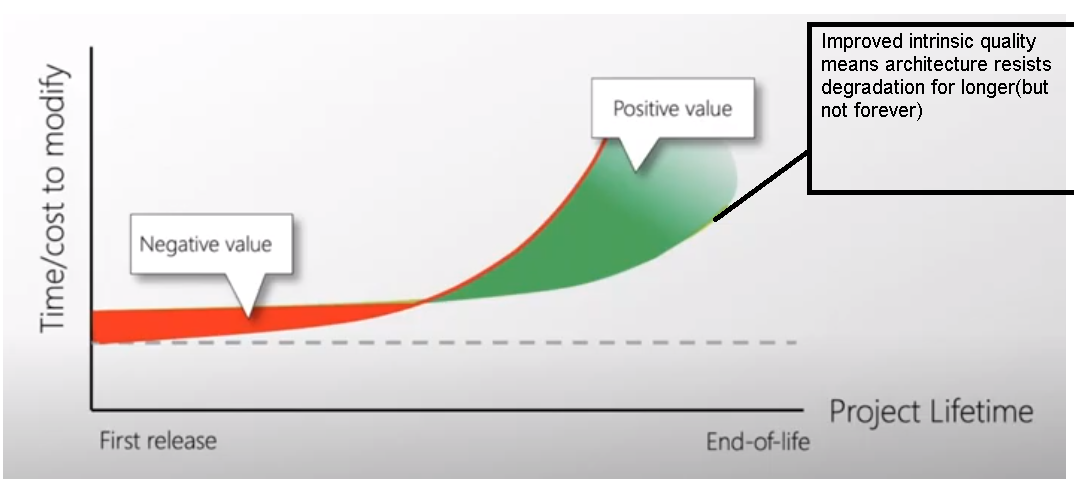
**Solid Principles, Factory Pattern**

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

**Why SOLID Principle is required.**

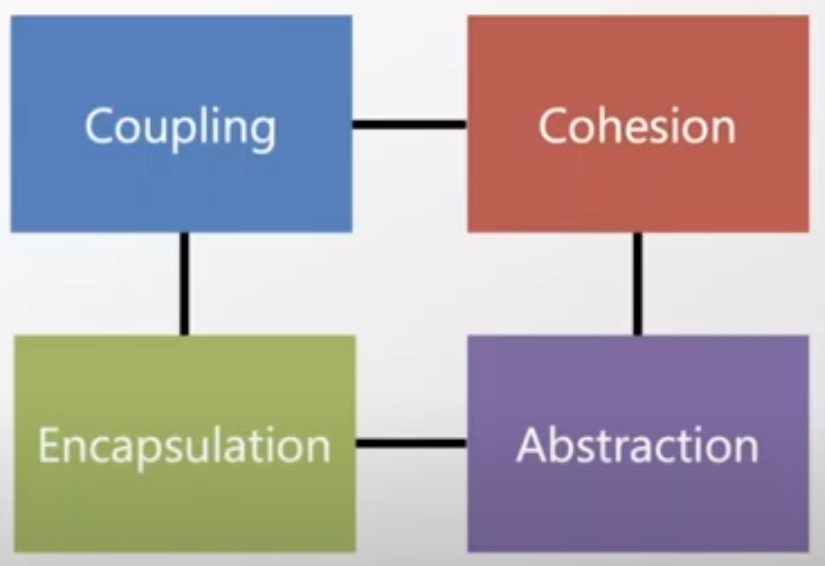
1. The initial cost of making a software will be high due to the S.O.L.I.D. principles applied on the software design but the maintenance and upgrading capability will be high and very cost efficient.

* Negative value – initial cost spend for designing the software.
* Positive value – Low cost of maintenance and upgradation.



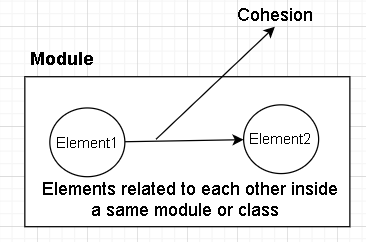
**SRP – Single Responsibility Principle**

1. Modularization: Technique to divide a software into modules.
   1. 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.



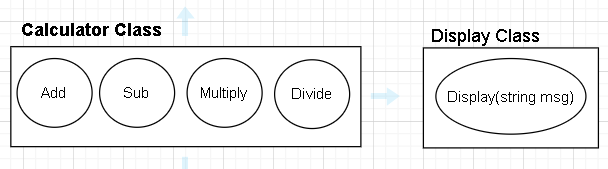
**1. Cohesion (Sticking together)**

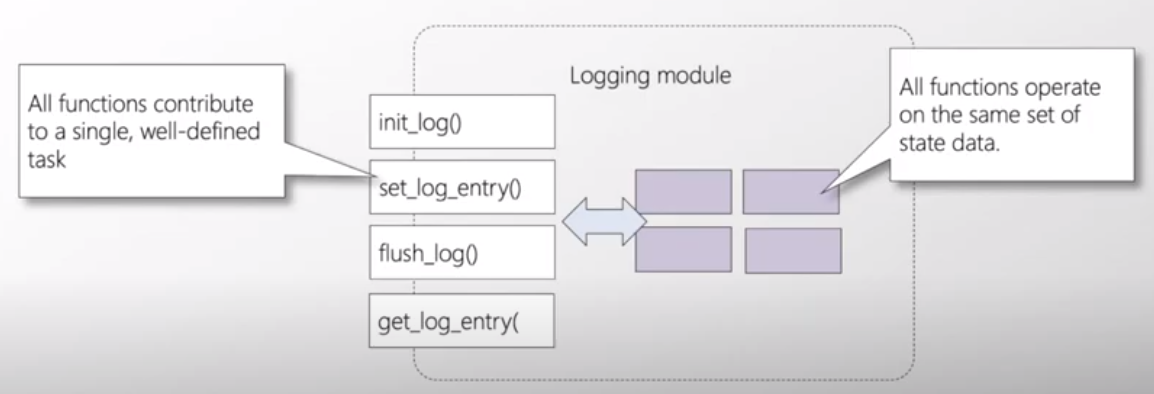
1. Cohesion – Elements of same characteristic should be place near/inside the same elements.
2. A Strongly cohesive module implements functionality focusing on a single feature of the solution.
3. Cohesion is a glue that holds a module together.
4. When a module element changes, it should be changed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



1. Type of Cohesion
   1. Functional Cohesion: If 2 operation present within a module\class perform the same functional task **OR** they are part of the same “Purpose”.

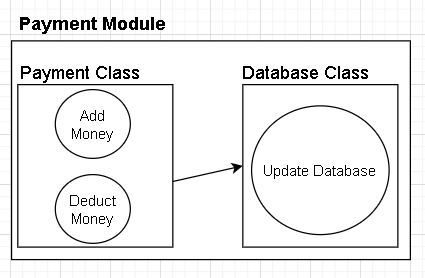
Each Element does some related task. Atomic functions



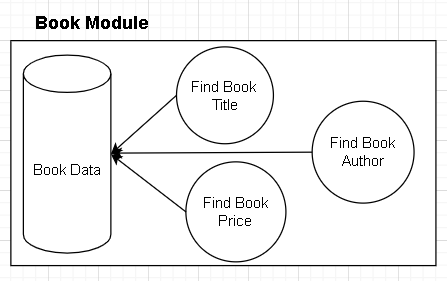
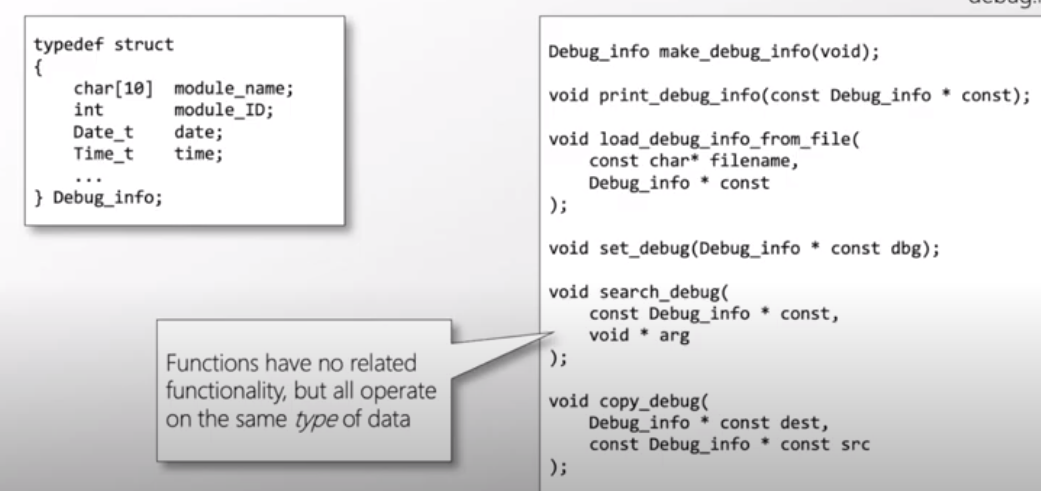


* 1. Sequential Cohesion: In a module, If two operations are such that

X’s Output **is** Y’s Input

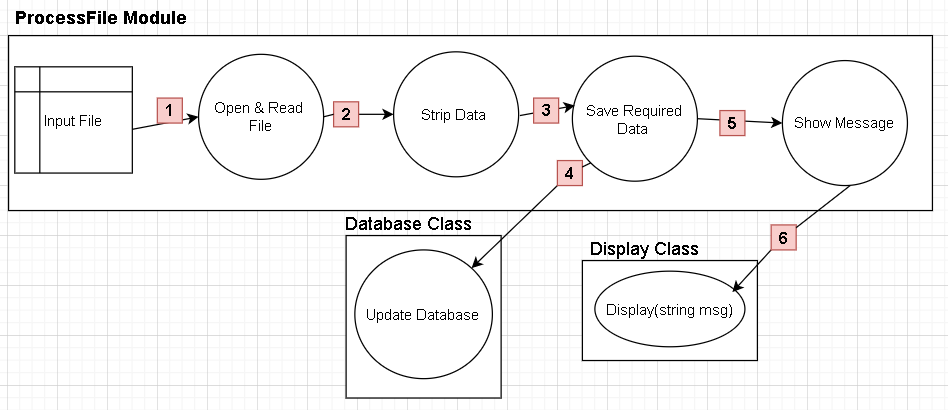


* 1. Communication (Information) Cohesion: A module exhibits “Communication Cohesion” if all the activities it supports uses the same input or output data **OR** access and modifies the same part of a data structure.

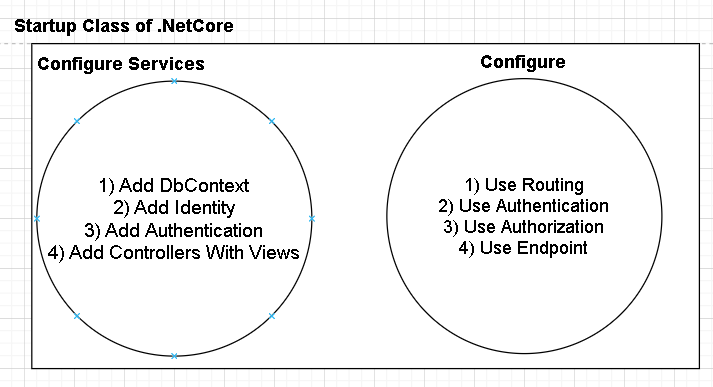
 

* 1. Procedural Cohesion: A Module whose instruction do different tasks, but to ensure a particular order in which tasks are performed, they are put into same module

**Cons: Poor Maintenance**



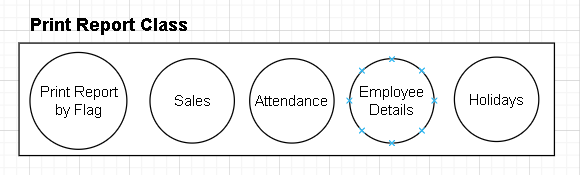
* 1. Temporal Cohesion: Instruction that must be **executed during same time span** are put together (Activities related in time).
     1. Configure & Configure Services: All the unrelated functionality such as routing, authentication, authorization and endpoint should be executed at same time.



* 1. Logical Cohesion: A logical cohesive module is one where elements contribute to activities of the same general category

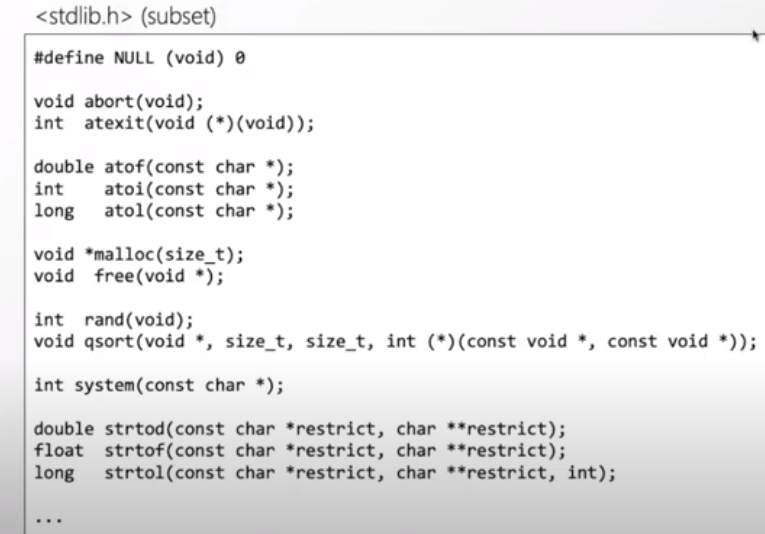
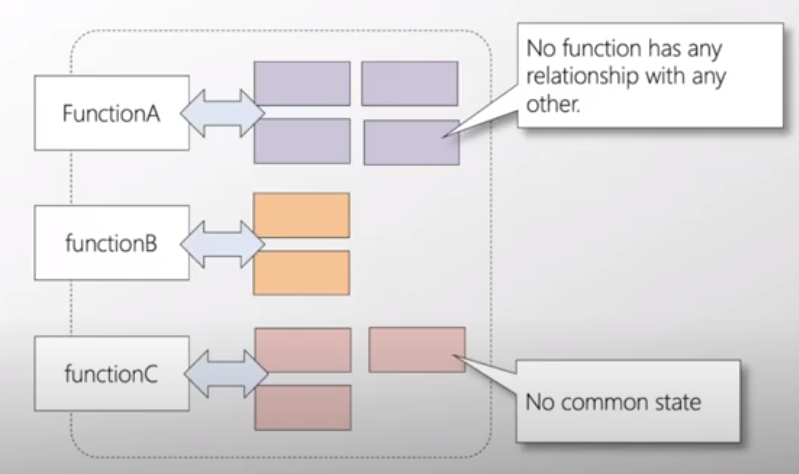
It basically use FLAG for switching and checking which task needs to be performed.

**FLAGS ARE TO BE AVOIDED, BECAUSE IT ADDS COMPLEXTIY TO THE CODE**

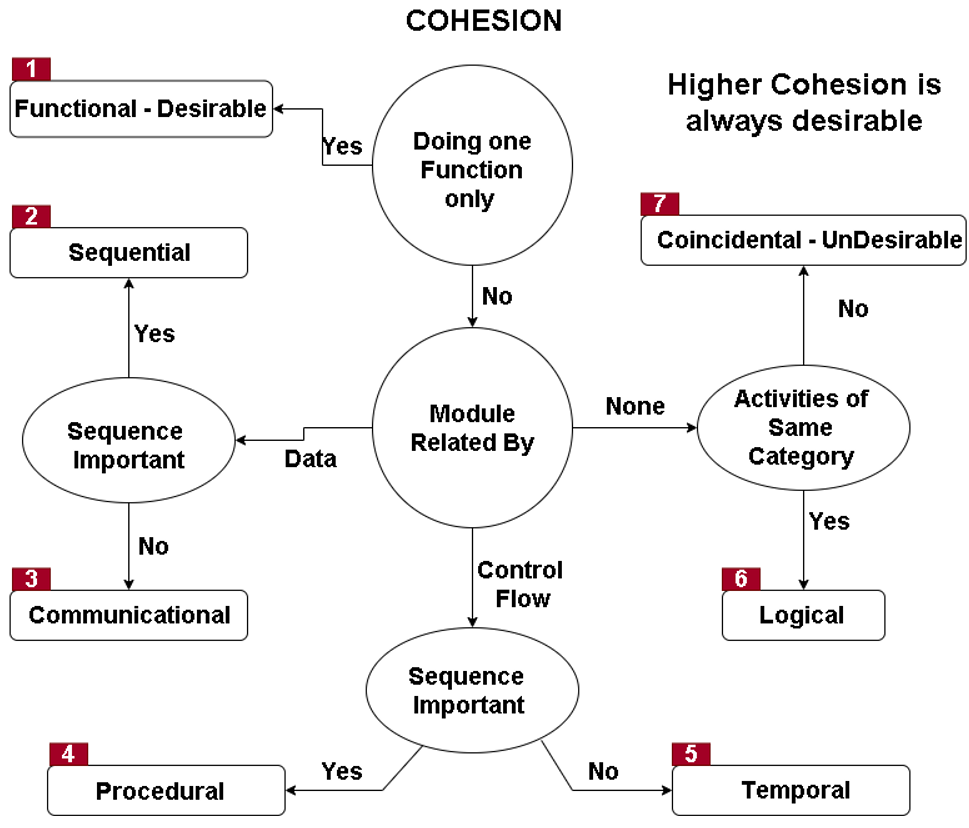


* 1. Coincidental Cohesion: Coincidental Cohesion implies little or no relationship among the statement of code within a procedure (a module).

E.g.



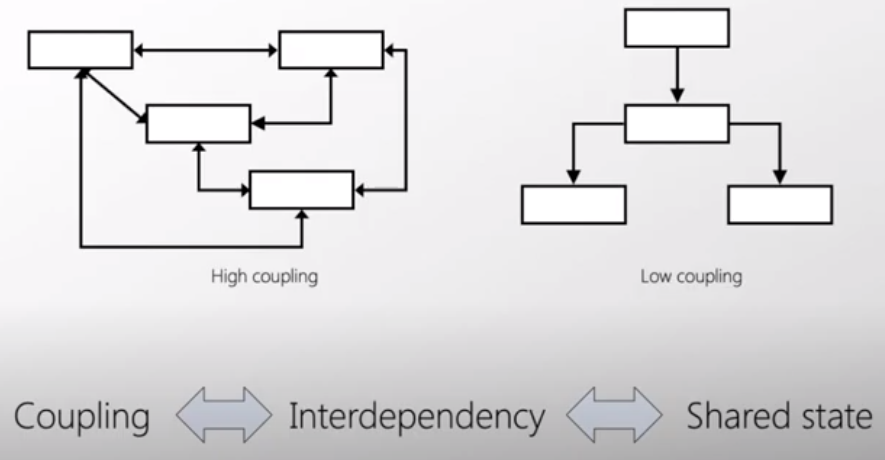
1. Easy way to understand Cohesion



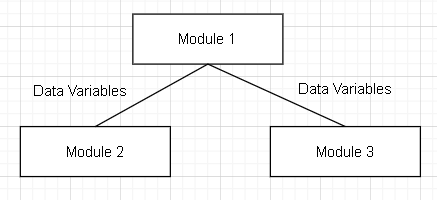
Intentionally left blank…

**2. COUPLING**

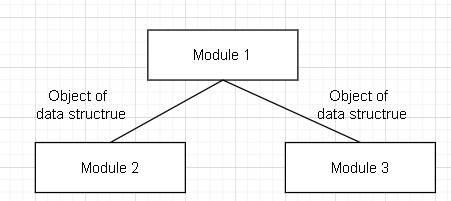
1. Coupling is the measure of degree of interdependencies between modules in the system
2. Coupling is the amount of the state information that modules share between them.
   1. More state information the module share, higher the coupling between them.
   2. High Coupling => Strongly inter-related/Dependents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



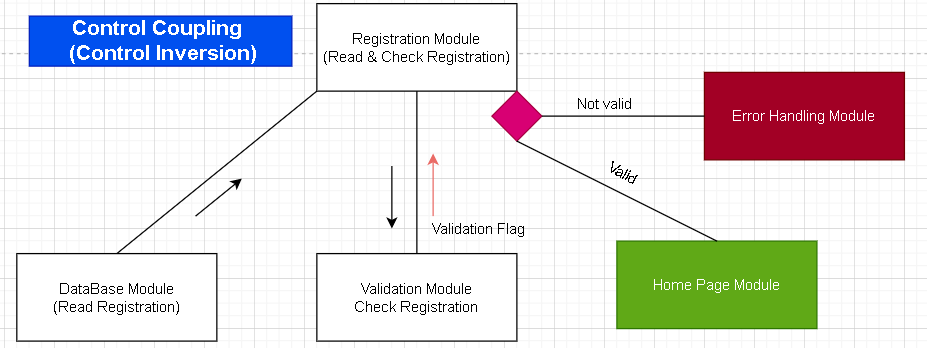
1. **LOW COUPLING IS DESIRED**
2. **HIGH COUPLING LEADS** to more **ERRORS** and IT makes **ISOLATION** (Debugging) of **ERRORS VERY Difficult**
3. Types of Coupling
   1. Data Coupling – Components are independent to each other, they communicate via data only.



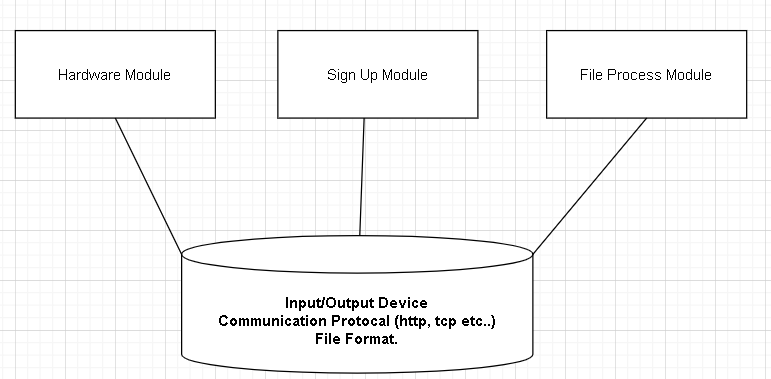
* 1. Stamp Coupling – The module passes whole data structure (objects) to another module.



* 1. 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)

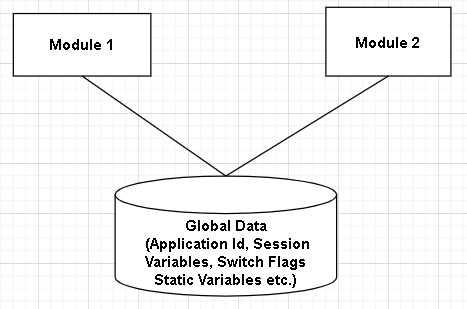


* 1. External Coupling –Two or more Modules share externally imposed data format, communication protocol or device interface.



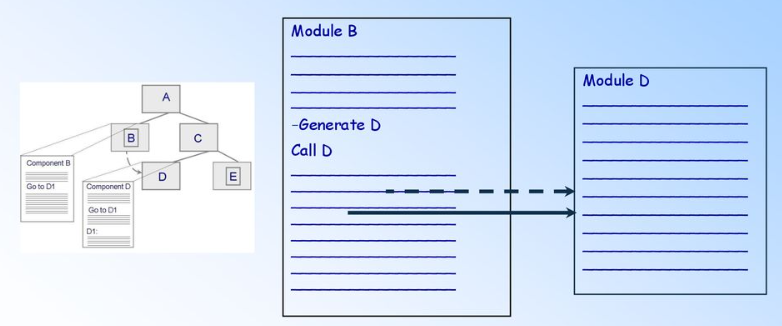
* 1. Common Coupling (global coupling) – when two modules share the same global data.

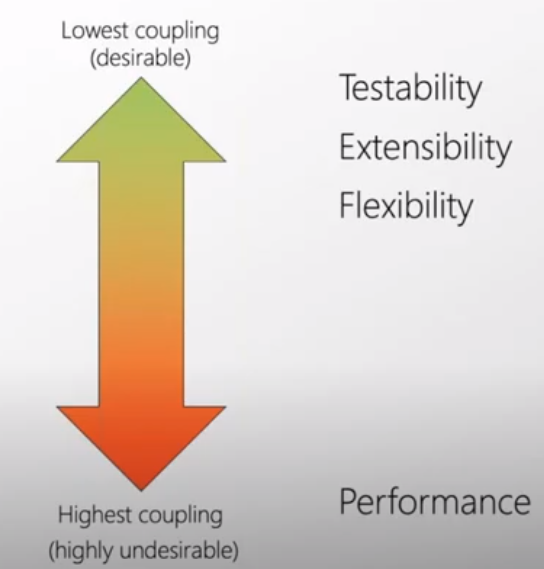
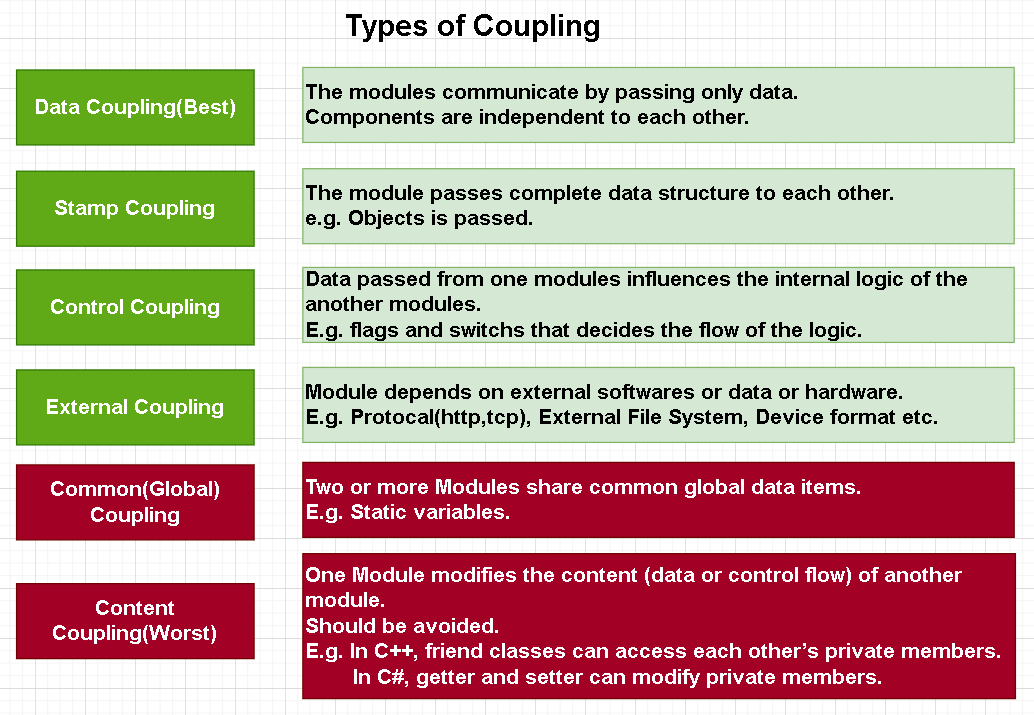
Changing the shared resource might result in some unexpected result for other modules.



* 1. Content Coupling – When one module modifies an internal data item in

Another module.





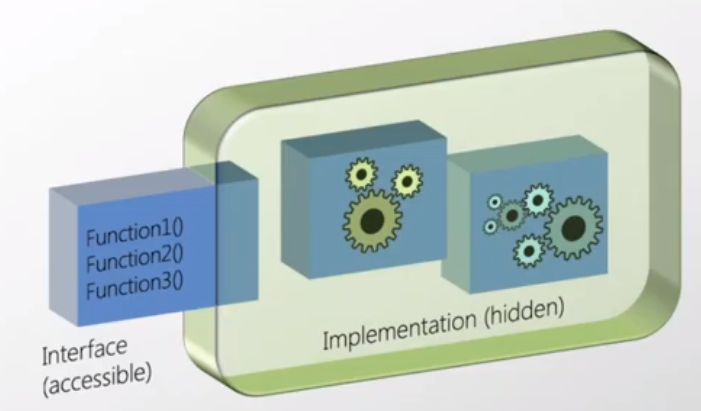
Left Blank intentionally

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)**

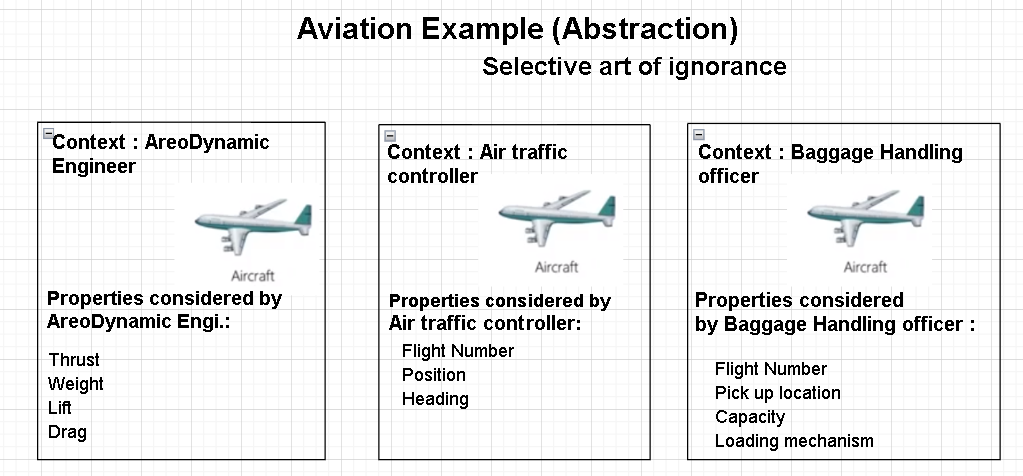
1. Encapsulation – Separating implementation from interface (accessible) and hide the implemented details (state information, internal processing of implementation (algorithm behavior)).



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

**4. Abstraction(Short Version) - Context**

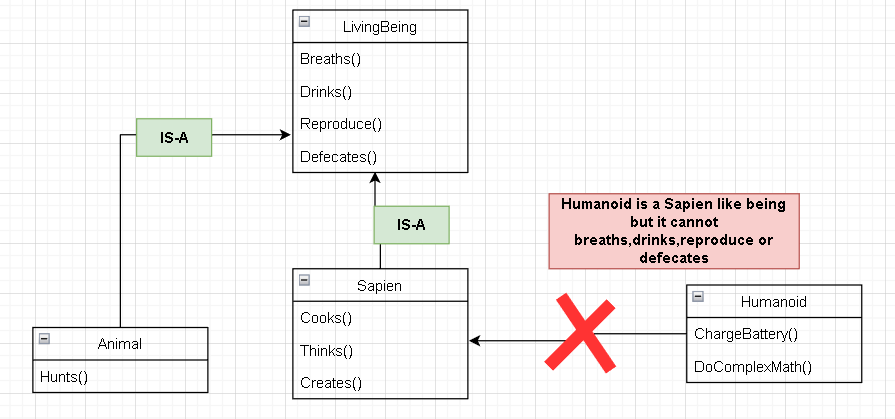
1. Art of Selective ignorance. i.e. Context
   1. User eating Burger King, User doesn’t need to know from where the buns and patties come from or from where it has been bought.
   2. The guy assembling your burger king doesn’t need to know the process of making the buns and patties. He just have to assemble it.
   3. So the context is important
2. Aviation example -
   1. 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.

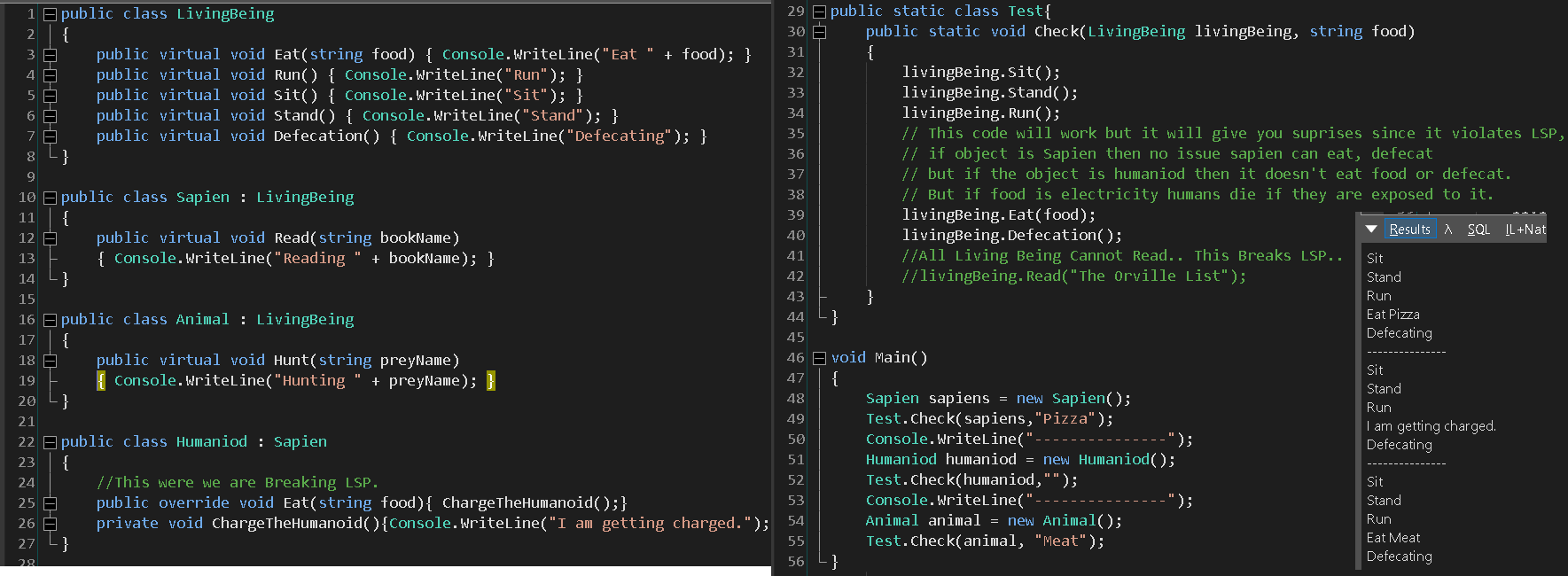


1. Liskov Substitution Principle –
   * Definition:

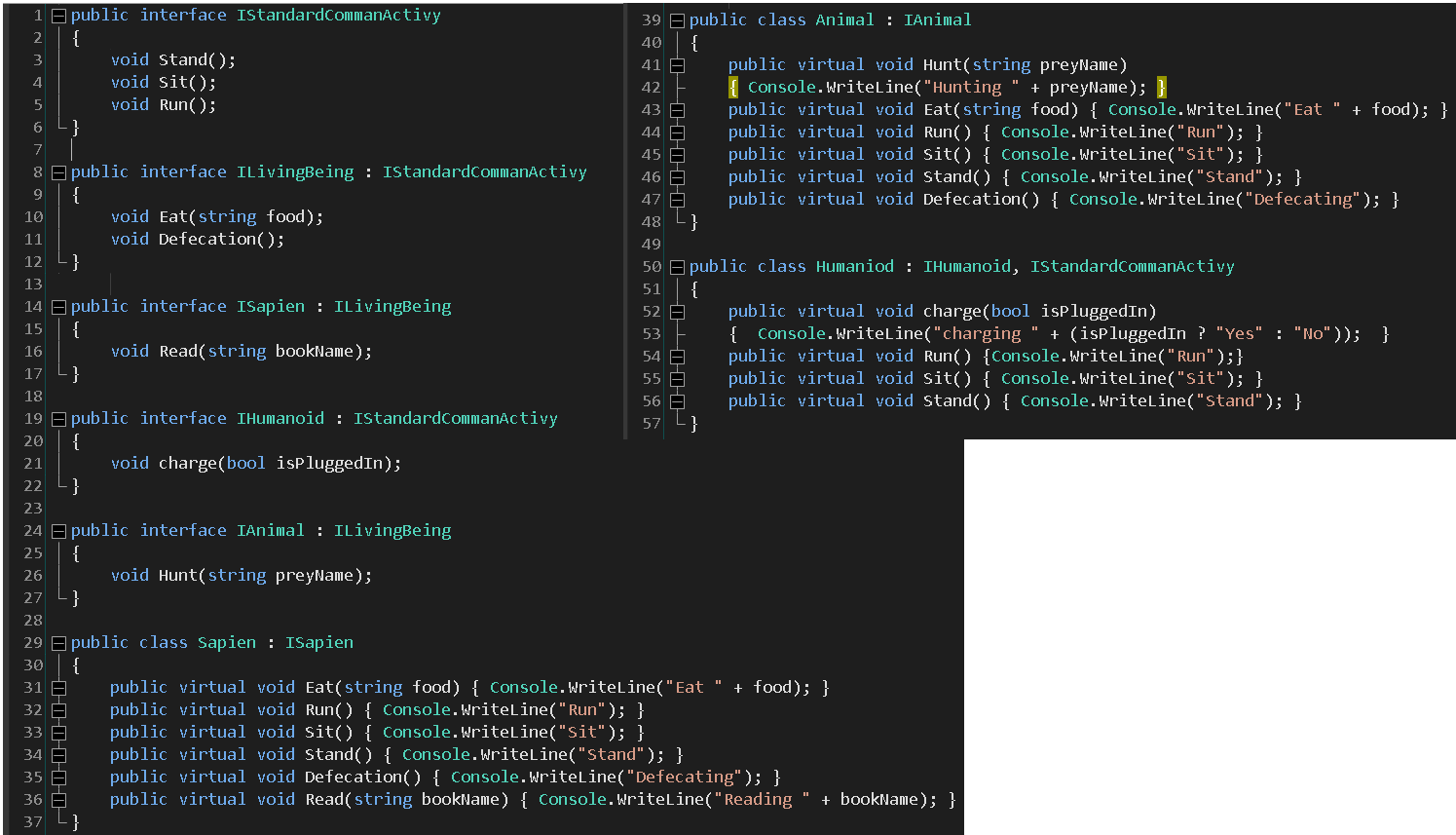


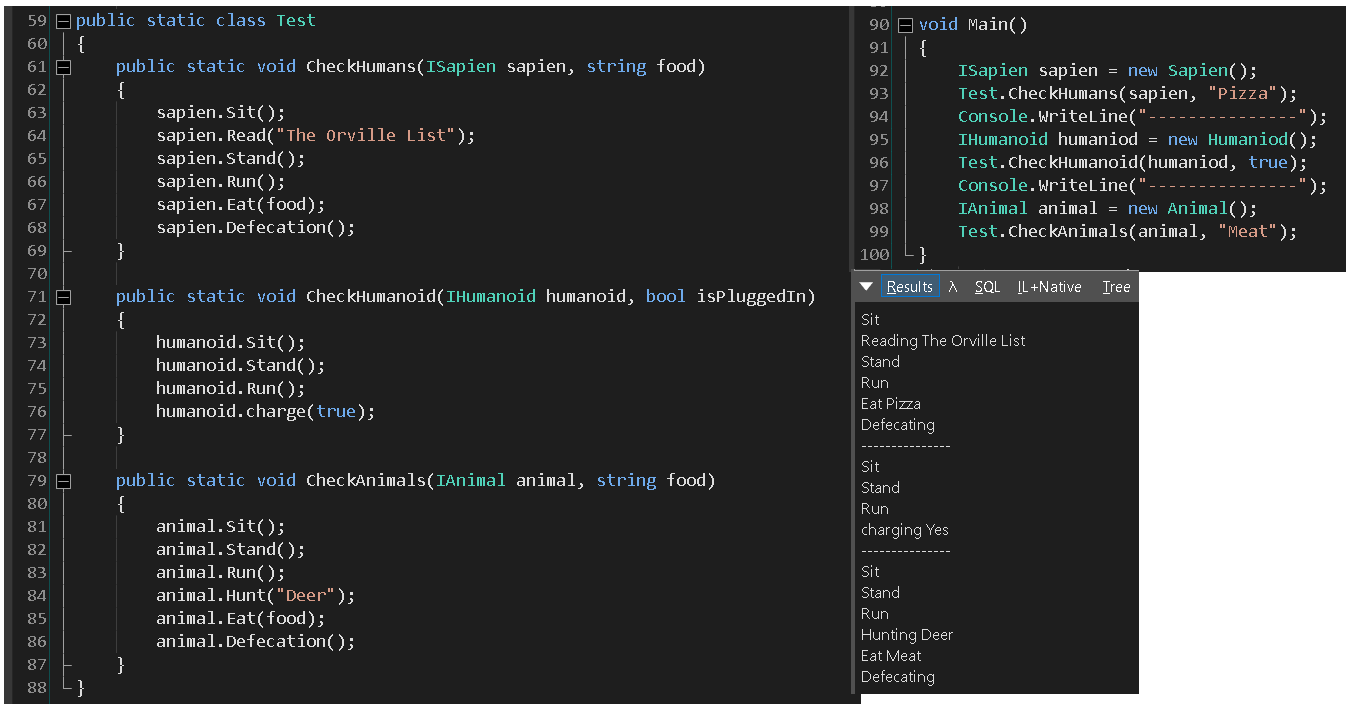
* + Simplifying Definition: If S is a subtype of T, then any term of type S can be safely used in context where a term of type T is expected.
  + Normalizing Definition: **LSP** states that object of a super class (parent class) should be safely replaceable with the object of its subclass (child class) 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.

1. 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 classwithout 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.
2. 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.
3. E.g. of Violation of LSP- Sapiens & Humanoid.

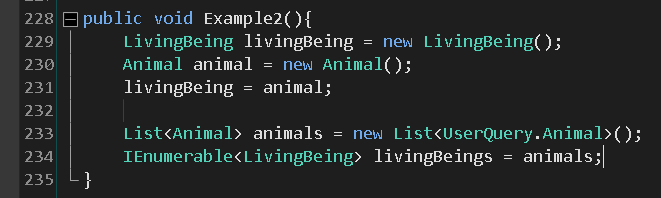
****

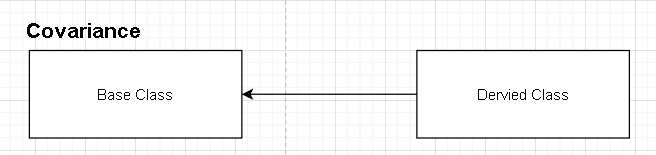
1. Improving the e.g. of Sapiens and Humanoid by implementing “**Interface Segregation Principle**”.

****

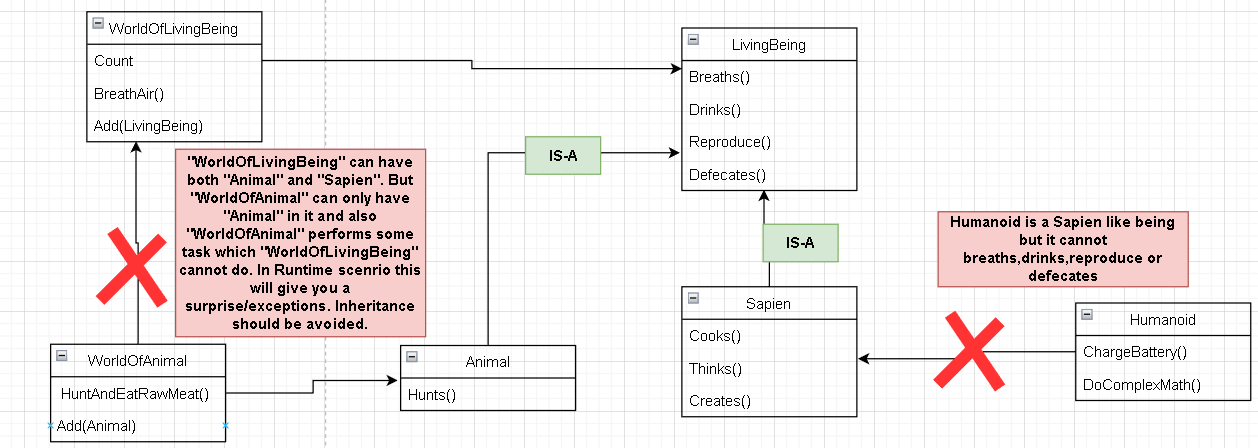
****

1. 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.
2. **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

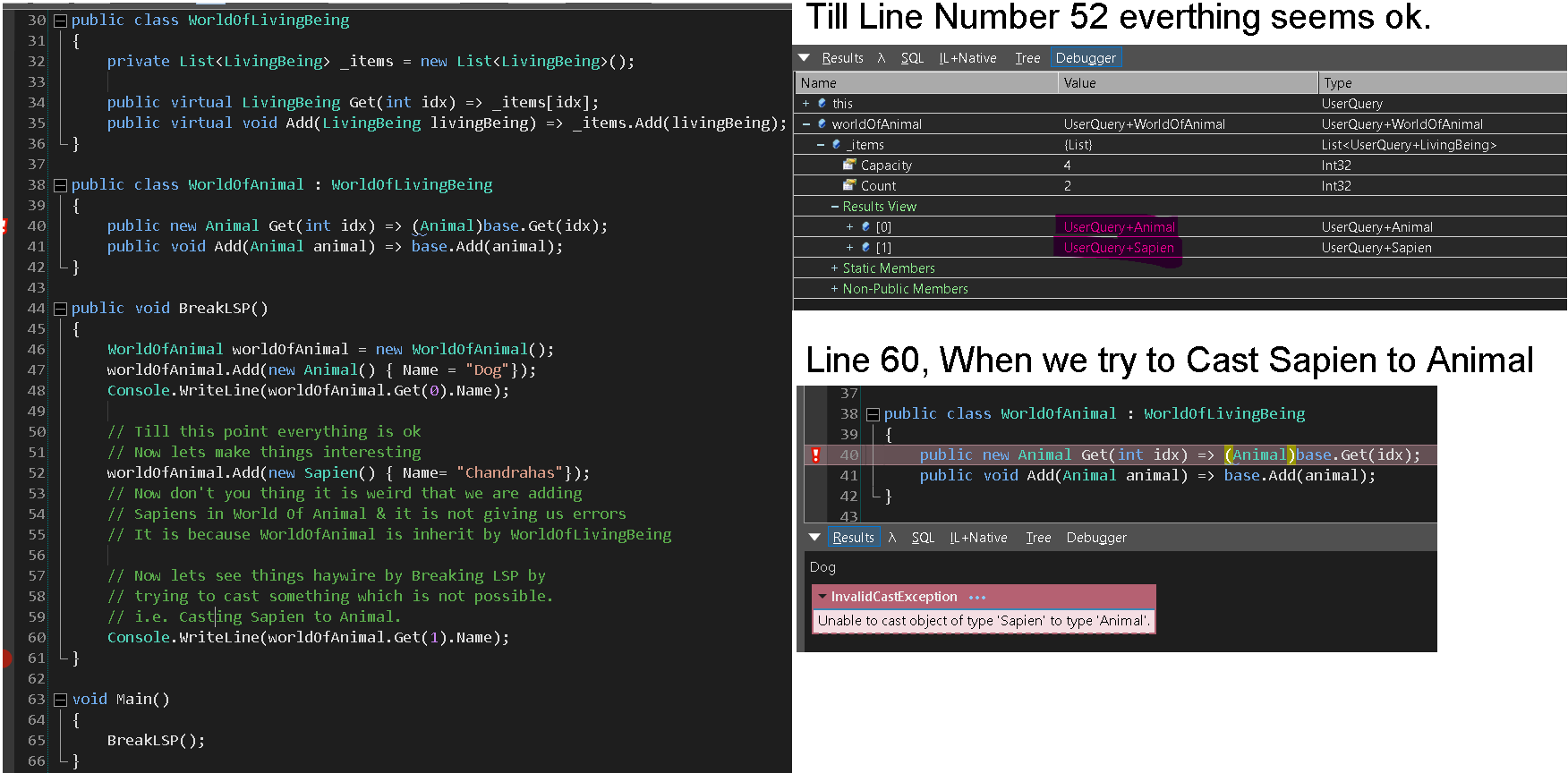




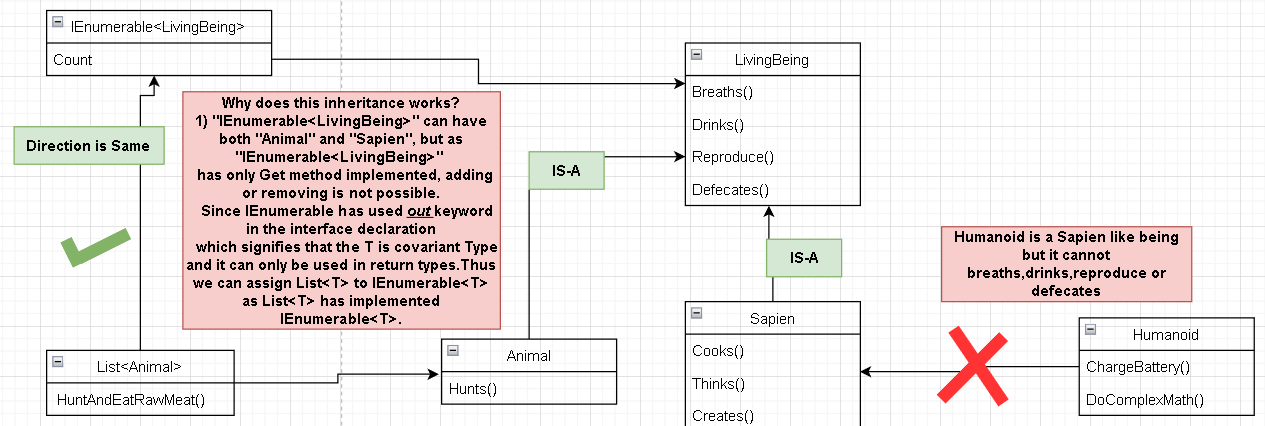
* + Not an Ideal Situation of LSP

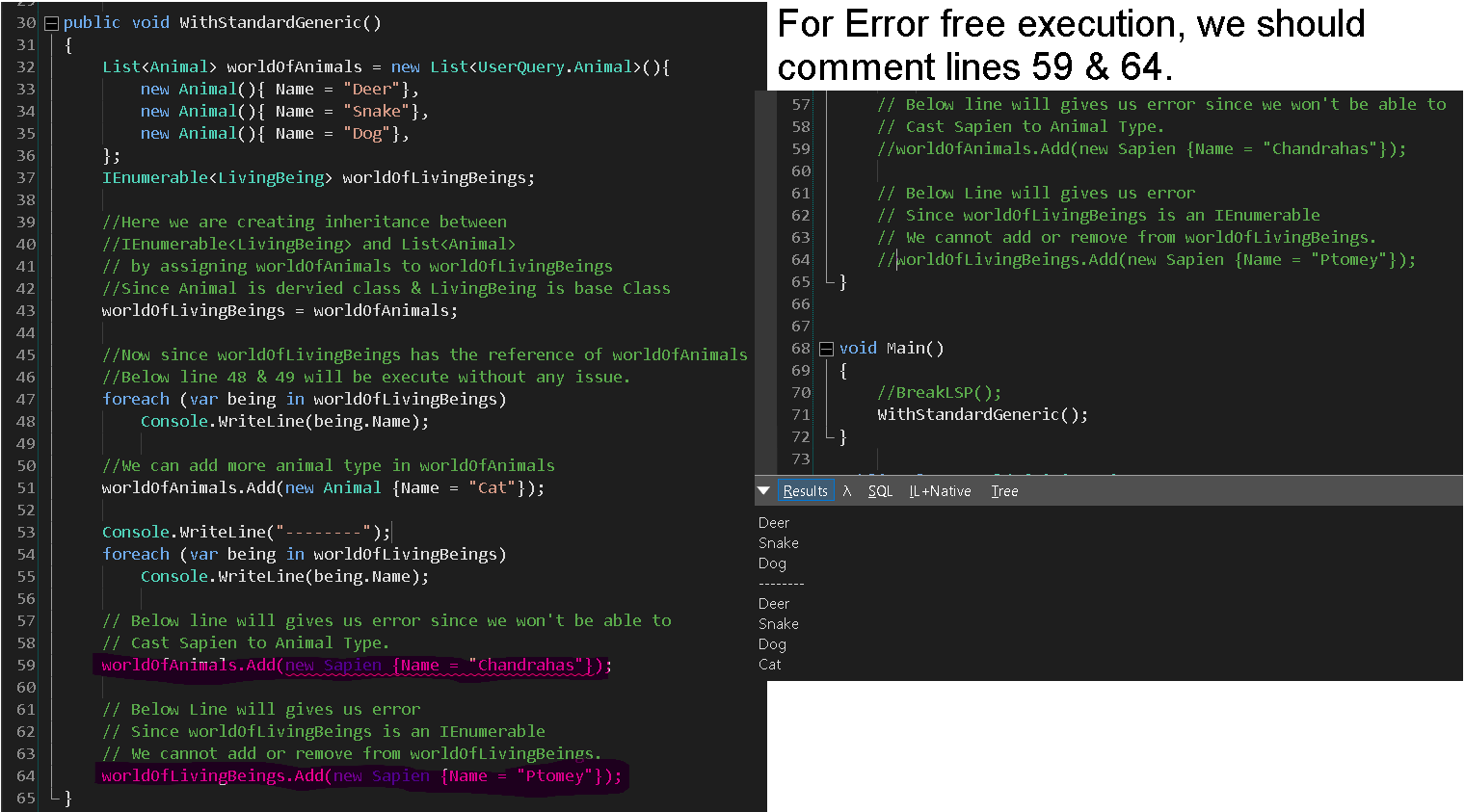
****

* 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.**

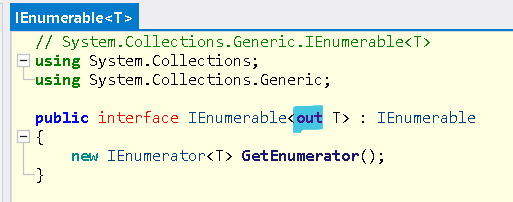
****

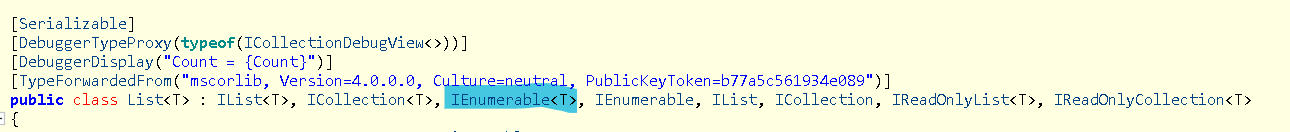
* + In the above example, we are hiding the base class method “Get” by using new keyword and **overriding** the “Get” method **entirely** in the child class according to our convenience. This breaks the LSP in **Line 40.**
  + Hence it is Said Breaking LSP will give you surprises i.e. run time exception. 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>**

****

****

* + - 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.

****

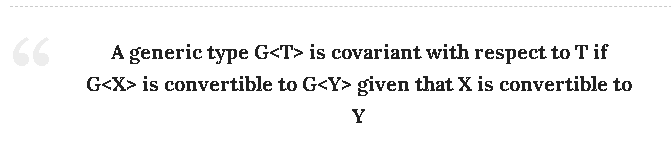
****

* + - * 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

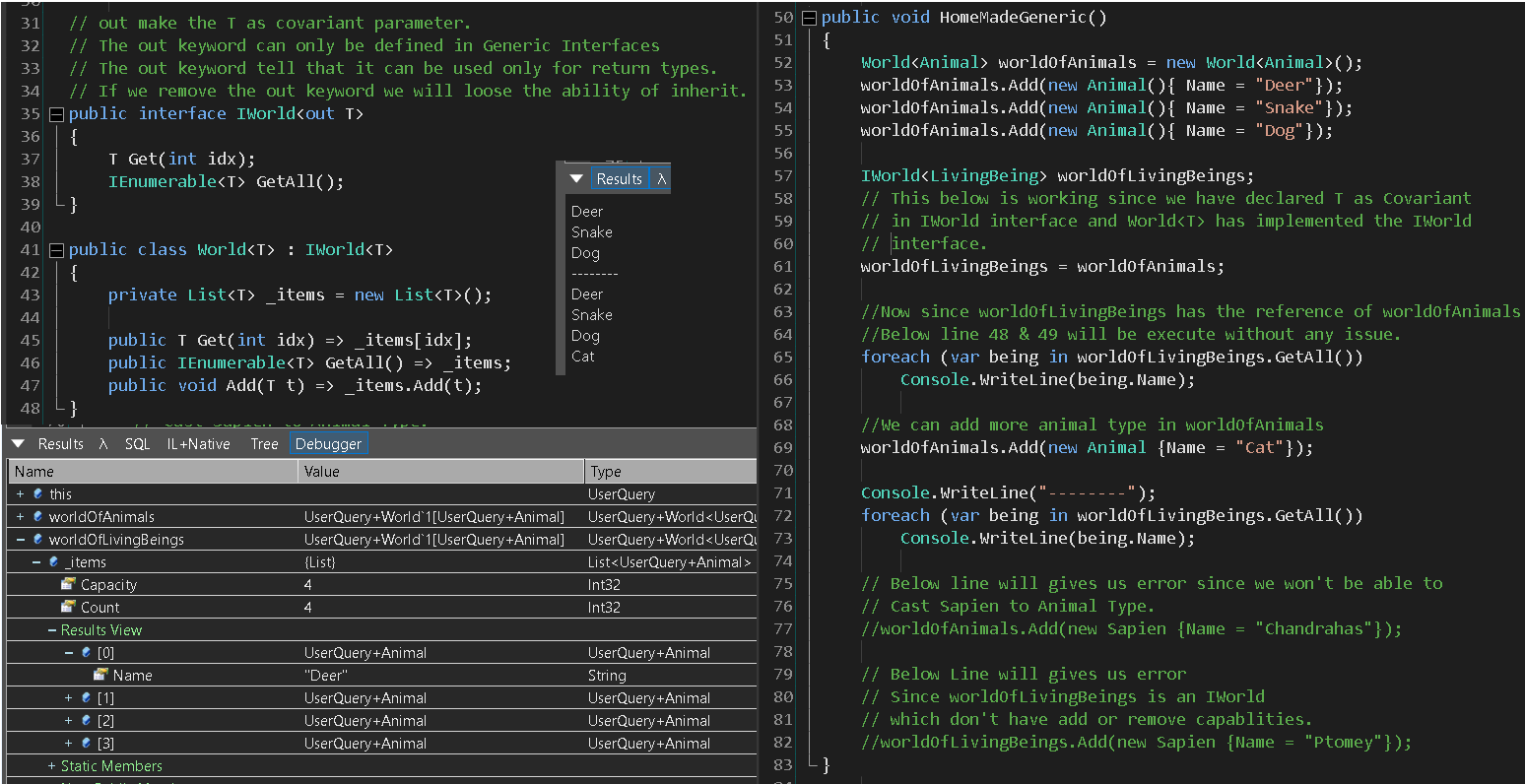
e.g. public interface IWorld<out T> {}

* + - * + If Generic Derived Class has implemented Generic interface

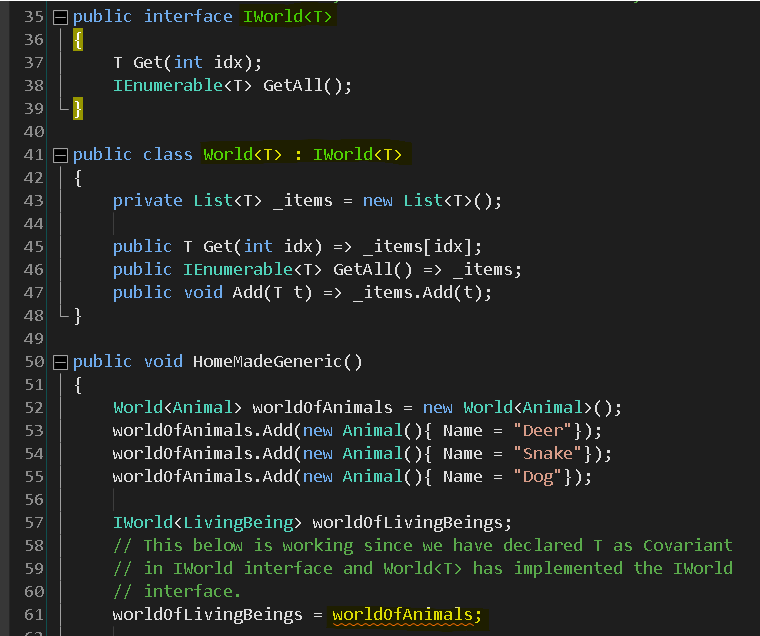
e.g. public class World<T> : IWorld <T>,…{}

****

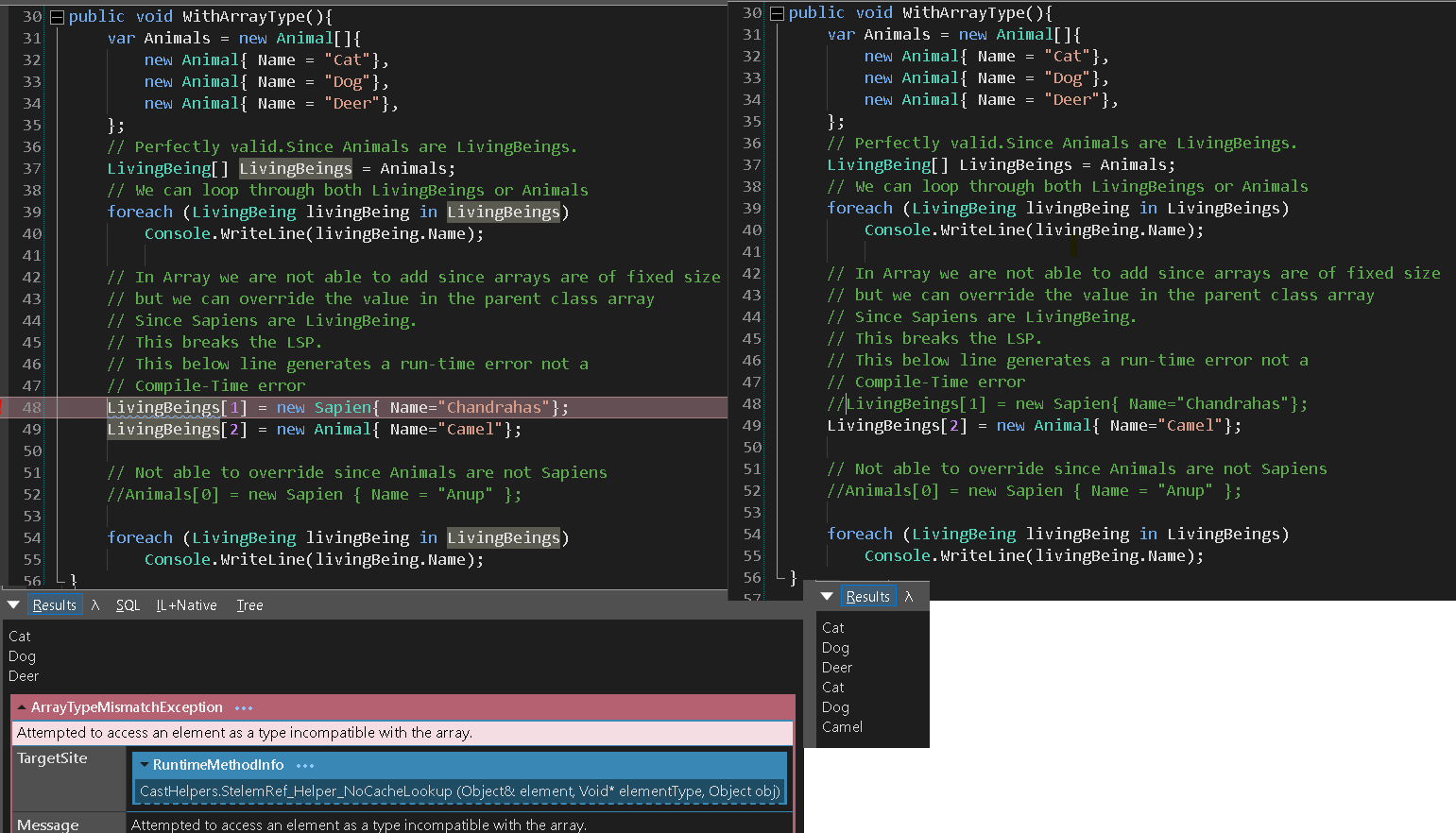
* + - * Let’s see example for Living Being & Animal with Homemade Generic.

****

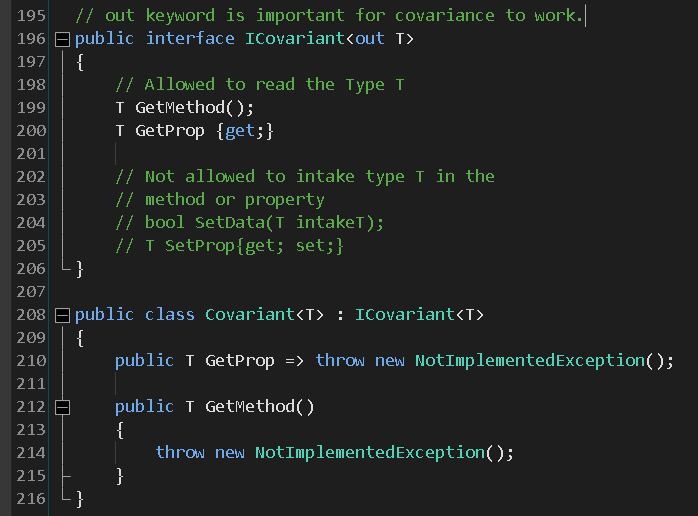
* + - * So **out** keyword mentioned in the interface tells us that this interface is has a **Covariant Type.**
      * The “T” in the covariant interfaces can only be used in the methods and properties with return type. It cannot be used as method parameters such as Add(T obj), if we have mentioned **out** keyword in the generic interface declaration.
      * So this means we can use thistype 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.

****

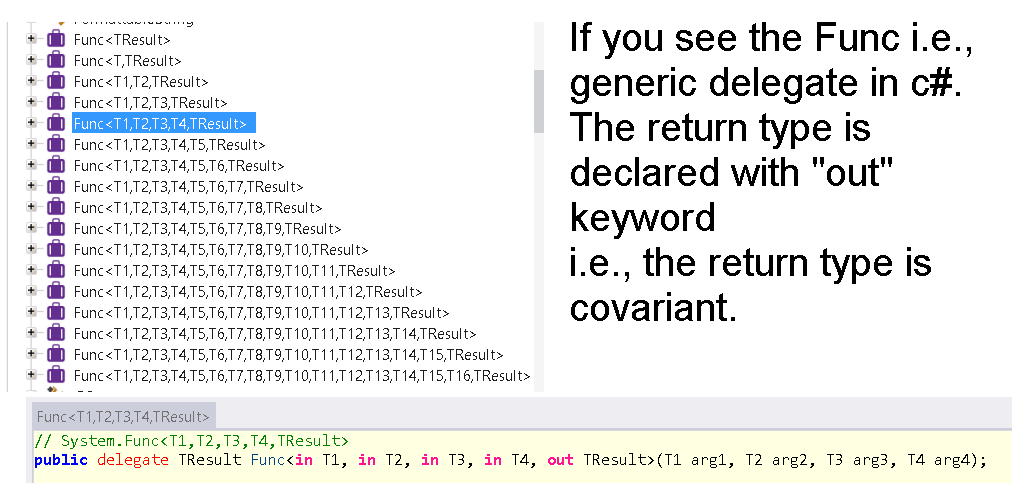
* + - * In primitive type **Array**, Covariance doesn’t seems to work perfectly though we cannot add a new value in array as it is of fixed size. We can override a value in array and this might lead to runtime error.
      * For Example

****

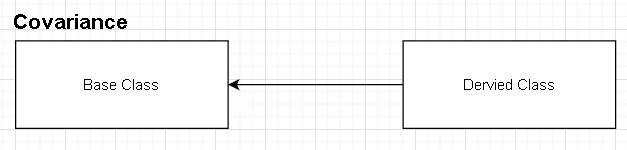
* + Example of How to Declare Covariant interface & Implement Covariance interface

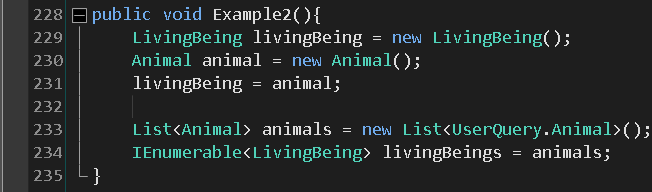
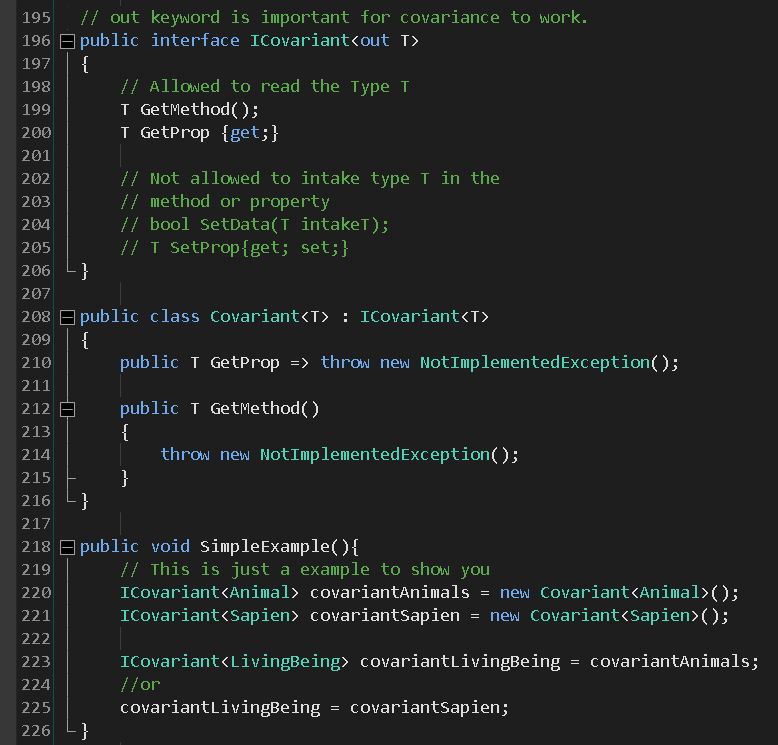
****

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

****

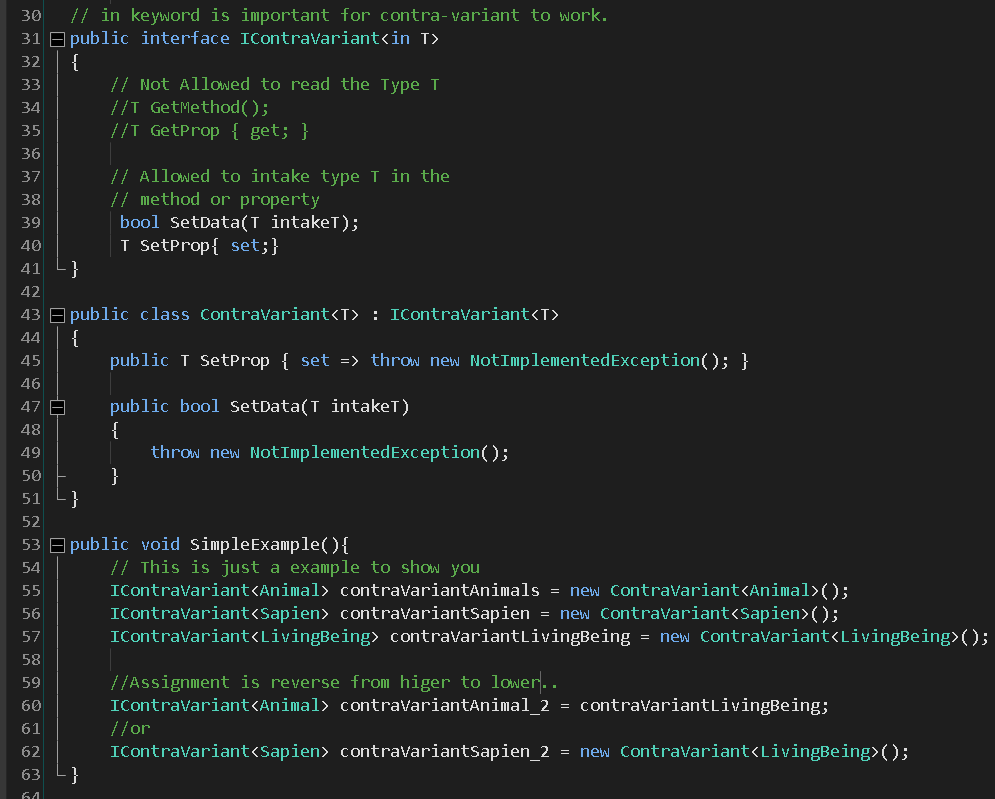
1. **Contravariance – Reverses the assignment compatibility** 
   * So in covariance, we have seen that we can assign derived type to base type.

****

****

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

****

****

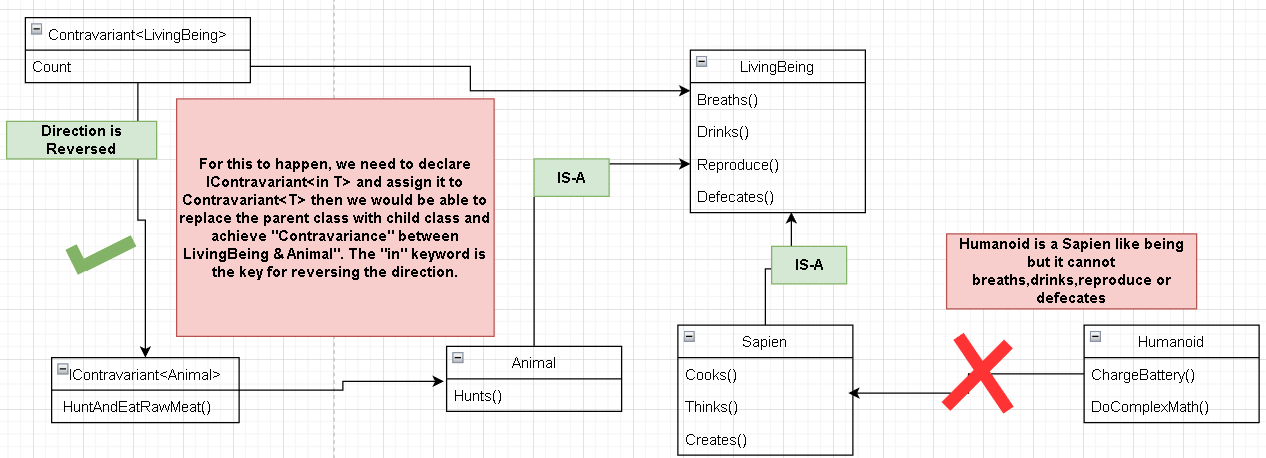
* + In above example,
    - We can see that we are able to assign object of Contravariant of LivingBeing to IContravariant of Animal.
    - We are allowed to assign instantiated interface of base class to the interface of derived class.

i.e. IContravariant<DerivedClass> objectName = new Contravariant<BaseClass>();

OR

IContravariant<BaseClass> baseClassObjName = new Contravariant<BaseClass>();

IContravariant<DerivedClass> objectName = baseClassObjName;

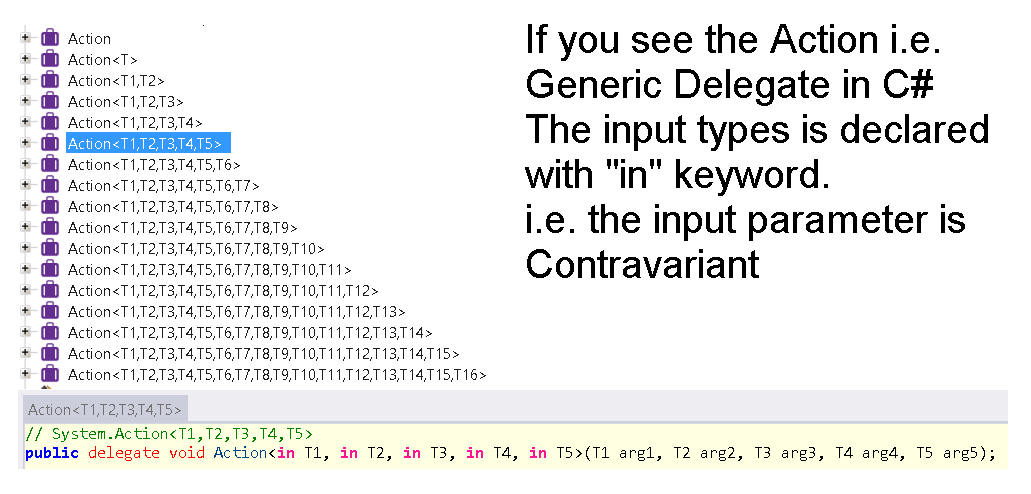
* + - ****
  + When we define only Contravariant interface it means
    - We cannot have a method that returns T in the method signature

i.e. T Get(int x); is not valid.

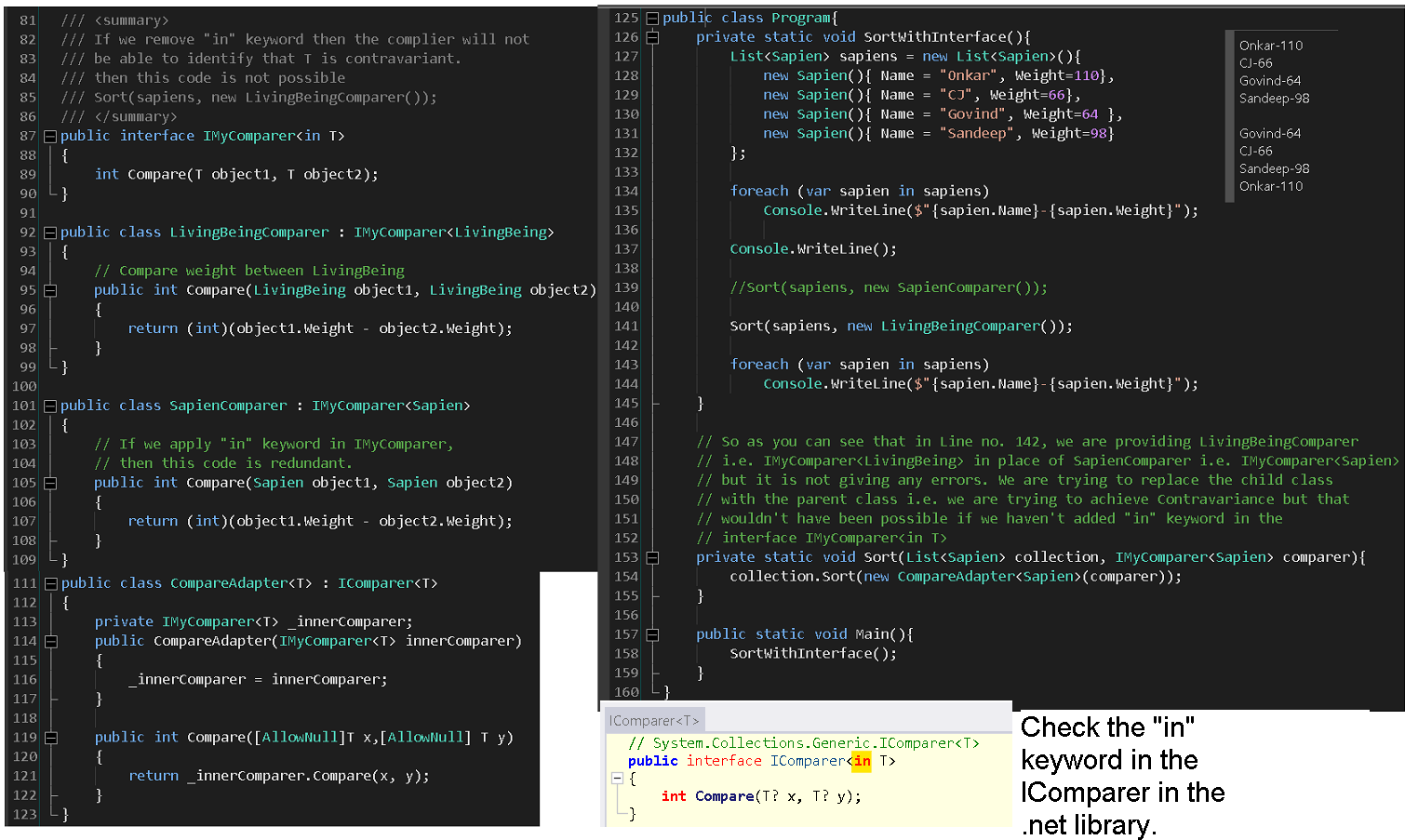
* + - We can have a method that take T as input parameter.

i.e. void Set(T t); is valid.

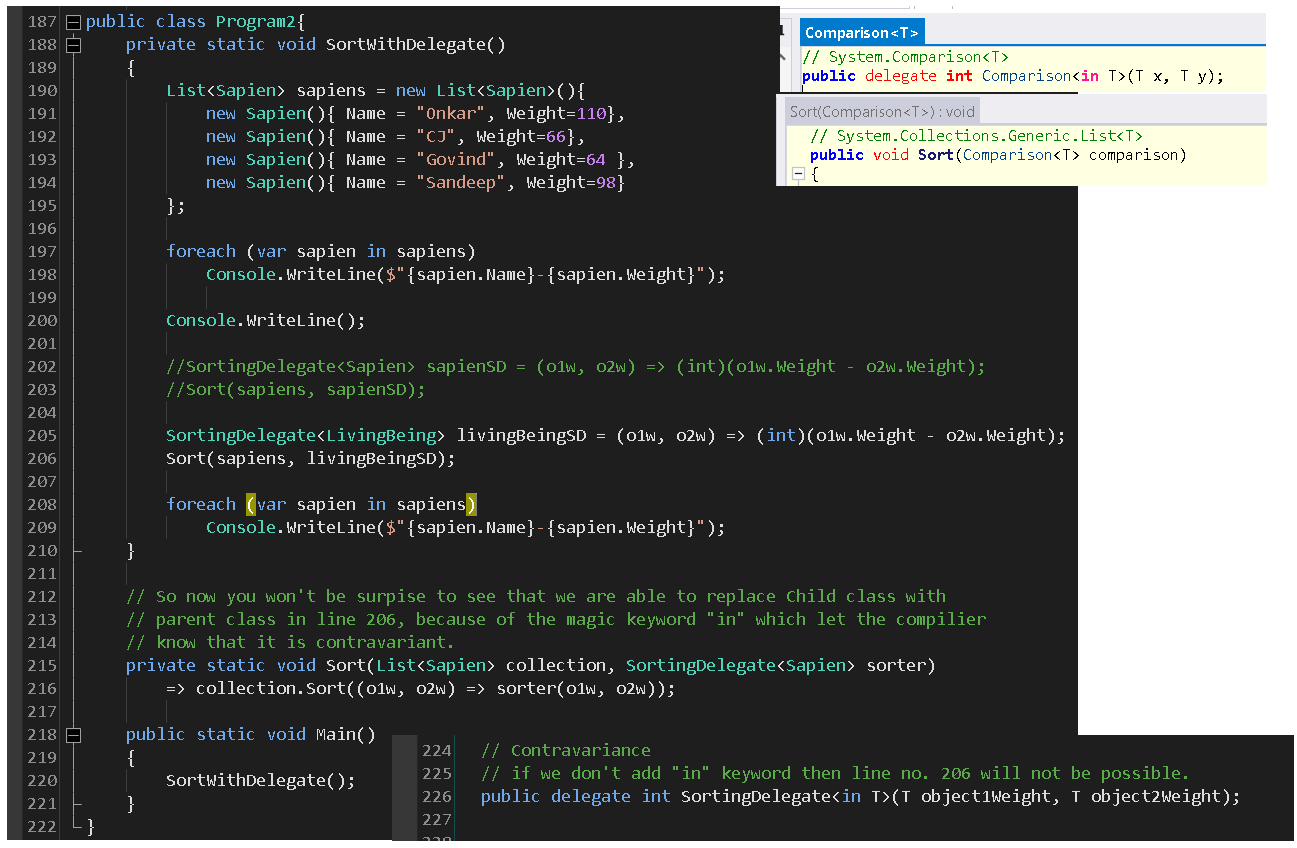
* + - For Properties, we can only have set only properties.
  + Normally, Contravariant interface are used only for writing the data not reading the data.
  + **Action<in TInput1>** is **Contravariant**.

****

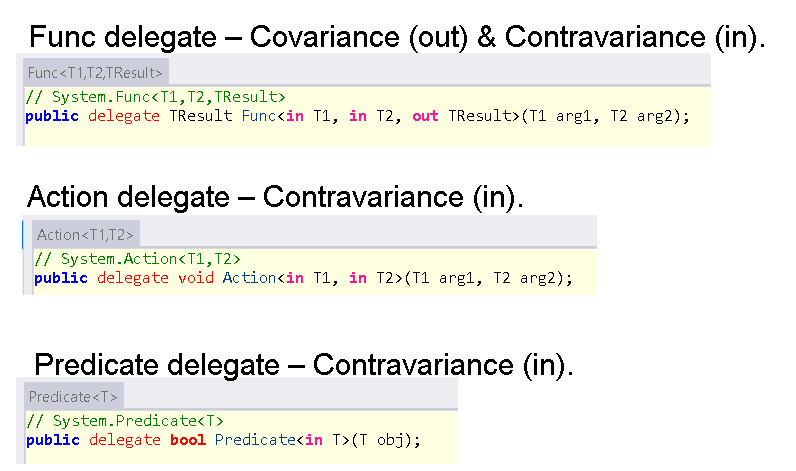
* + Let’s understand Contravariant via code, this example is transformed from “Coding Tutorials” video

****

* + 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.

****

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

****

1. **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**

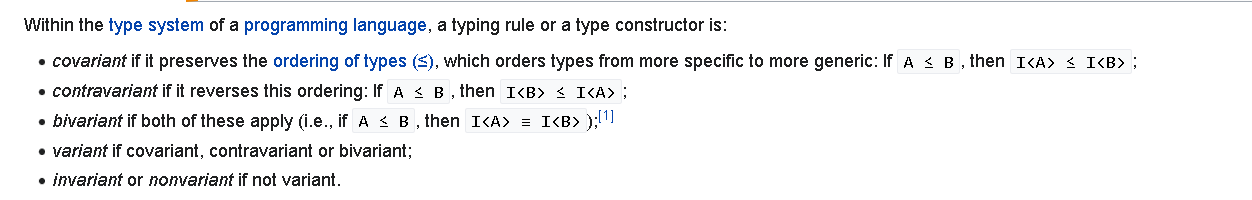
* + - * **Let’s reverse (x <= y) = (Square(y) <= Square(x)) -> No**
      * Replace x = 1 & y = 2 then

**(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

****

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=OBrS1L0Hlcs>

**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/>