Seminario #7:
Varianza, Covarianza, Herencia, Polimorfismo y Encapsulamiento.

La **herencia**: es uno de los mecanismos de los lenguajes de programación orientados a objetos basados en clases, por medio del cual una clase se deriva de otra de manera que extiende su funcionalidad. La clase de la que se hereda se suele denominar *clase base o clase padre.* En C# solo existe herencia simple por lo que las clases solo pueden heredar de una sola *clase padre,* sin embargo, pueden implementar varias interfaces. La herencia es transitiva, lo que implica que si la clase A hereda de la clase B y la clase B hereda de la clase C, entonces la clase A también hereda de la clase C. Una clase derivada puede agregar miembros de su propia definición, pero no puede eliminar miembros heredados. Los constructores no son heredados.

El **polimorfismo**: es la capacidad de que los objetos de diferentes clases con métodos con una misma signatura se puedan comportar de forma distinta. En C# es obligatorio especificar el comando virtual en la clase base. Además, es obligatorio usar la cláusula override en la clase que está sobrescribiendo (o implementando el método abstracto) de la clase base.

Podemos identificar tres tipos de polimorfismos:

1- Polimorfismo por herencia.
2- Polimorfismo por abstracción.
3- Polimorfismo por interface.

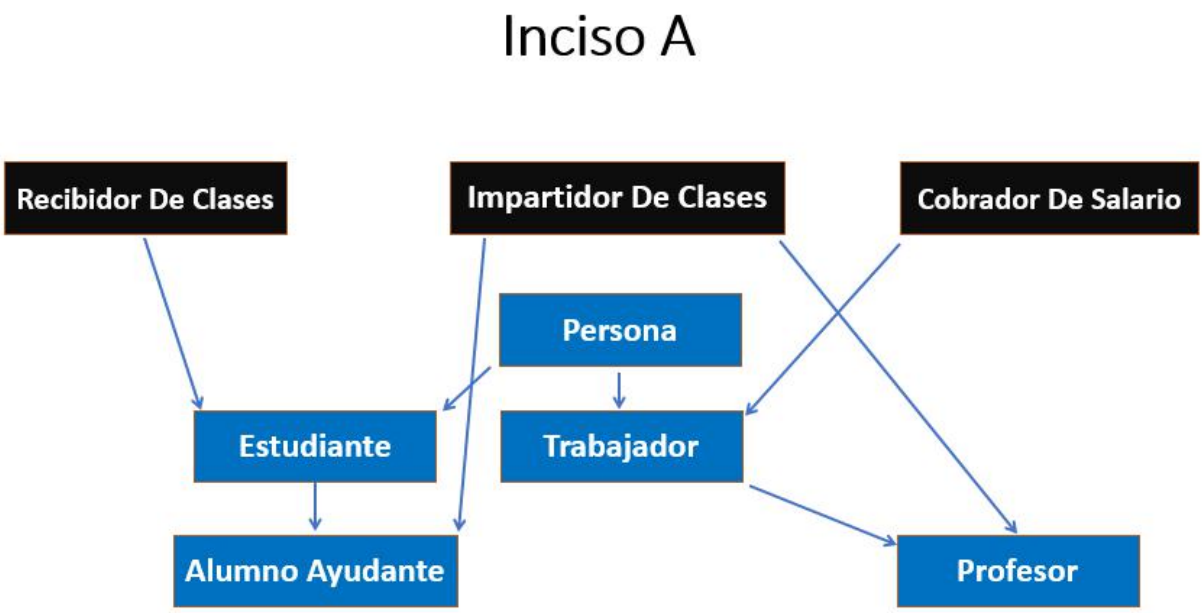
El **polimorfismo por herencia** se refiere a cuando se hereda de una clase normalmente, pero la clase que hereda re-define o mejor dicho sobreescribe (override) los/las métodos/propiedades de la clase base. Podemos verlo en un ejemplo sencillo que se ilustra en el Anexo 1.

El **polimorfismo por abstracción** hace referencia a las clases abstractas, estas clases tienen una característica muy especial, *NO SE PUEDEN INSTANCIAR*, esto quiere decir que no podemos crear un Objeto (hacer new) de una clase abstracta. Un ejemplo sencillo se ilustra en los anexos del trabajo.

En el **polimorfismo por interface** las instancias permiten utilizar métodos comunes a varias clases que no necesariamente tengan que heredar de clases superiores, esto quiere decir que permiten representar métodos de las

clases que no tienen relación entre ellos. Un ejemplo sencillo se ilustra en el Anexo 3.

Encapsulamiento: Cuando se diseñan clases, hay que determinar que miembros pueden ser visibles para el usuario y cuáles no. El encapsulamiento es el ocultamiento de los datos miembro de un objeto de manera que sólo se puedan cambiar mediante las operaciones definidas para ese objeto. Esto permite usar el objeto solamente de la manera que fue concebido inicialmente. En C# esto se logra utilizando los modificadores de acceso (public, protected, etc…)



Varianza:

La covarianza y la contravarianza se denominan colectivamente varianza. Un parámetro de tipo genérico que no esta marcado como covariante ni contravariante se denomina invariante. En .NET Framework 4, los parámetros de tipo variante están restringidos a los tipos de interfaz genérica o de delegado genérico. Un tipo de interfaz genérica o de delegado genérico puede tener parámetro de tipo covariante y contravariante. La varianza se aplica únicamente a tipos por referencia, si se especifica un tipo por valor para un parámetro de tipo variante, ese parámetro de tipo es invariante para el tipo construido reultante. La varianza no se aplica a combinación de delegados, es decir, si hay dos delegados de tipo *Action<Derived>* y de *Action<Base>* no se pude combinar el segundo con el

primero aunque el reultado tuviese seguridad de tipos. La varianza permite la asignación del segundo delegado a una variable de tipo *Action<Derived>*, pero los delegados solo se pueden combinar si tienen exactamente el mimo tipo.

En las versiones anteriores de .NET, todas las interfaces eran invariantes, lo que significa que no se podía convertir parámetros de tipo en otros tipos en la jerarquía de herencia. Esto ha cambiado en .Net Framework 4, que ahora incluye compatibilidad con la varianza para varias interfaces genéricas ya existentes. La compatibilidad con la varianza habilita la conversión implícita de las clases que implementan estas interfaces.

Las interfaces siguientes son ahora variantes:

1. IEnumerable(T): T es covariante

2. Ienumerator(T): T es covariante

3. IQueryable(T): T es covariante

4.IGrouping(Tkey, TElement): ambos son covariantes

5.IComparer(T): T es contravariante

6.IEqualityComparer(T): T es contravariante

7.IComparable(T): T es contravariante

En C# no existe la varianza de tipos de manera general. Por ejemplo, el siguiente código da error en tiempo de compilación:

```
IList<string> strings = new List<string>();

IList<object> objects = strings; //ERROR DE COMPILACIÓN
```

Esto ocurre porque sino podríamos hacer algo como:

```
objects[0] = 5;

string s = strings[0];
```

Lo cual constituye una violación de la política de tipos.

Sin embargo, en algunas interfaces como IEnumerable<T> no hay forma de modificar el estado de los objetos a los que se hace referencia, por lo que hacer esto

```
IEnumerable<object> objects = strings;
```

no implica los conflictos del ejemplo anterior.

Covarianza:

La covarianza permite que un método devuelva un tipo más derivado, que el definido por el parámetro del tipo genérico de la interfaz. Dicho de otra forma, es la propiedad por la cual podemos utilizar instancias de clases más pequeñas como si fueran instancia de una clase mayor. Un ejemplo de esto sería utilizar este código para imprimir una lista de profesores, aquí se evidencia la covarianza ya que al pasarle a este método una lista de profesores, estos que son una clase más es específica son tratados como Persona una clase más genérica.

```
public static void PrintPeople(IEnumerable<Persona> people)
{
        foreach (var item in people)
        {
           Console.WriteLine(item.Nombre);
        }
}
```

La ***covarianza*** también pude ser usada en un ***delegado*** por ejemplo :

```
Func<Profesor>func = ()=> { return new Profeor(); };

Persona p = func();
```

La función devuelve un objeto profesor que puede ser asignado a una persona sin tener que hacer ninguna conversión.

En .Net 4.0, un ejemplo de la covarianza en C# es la interface IEnumerable, la cual se declarará así:

```
public interface IEnumerable<out T> : IEnumerable

{

     IEnumerator<T> GetEnumerator();

}

 public interface IEnumerator<out T> : IEnumerator

{

    bool MoveNext();

    T Current { get; }

}
```

Con el modificador *out* expresamos que los elementos de tipo **T** se utilizan para ser accedidos mediante la interface. Esto nos permite considerar un IEnumerable<A> como un IEnumerable<B> si A tiene conversión por referencia a B.

Contravarianza:

Una instancia de un tipo genérico contravariante con un parámetro de tipo dado se puede convertir implícitamente al mismo tipo genérico con un parámetro de tipo más derivado. Dicho de otra forma podemos utilizar instancias de clase más grandes como si fueran instancia de clases más pequeñas. Para los parámetros de tipo existe el modificador "in" que indica que estos solo ocurrirán en puestos de entrada. Un ejemplo de esto

```
public interface IComparer<in T>

{
    public int Compare(T left, T right);
}
```

Gracias a esto podemos utilizar un IComparer<object> como un IComparer<string> debido a que string contiene todos los miembros de Object.

Otro ejemplo de contravarianza seria tratar de utilizar el siguiente comparador para poder comparar cualquier clase que herede de persona.

```
public class CompararPorNombre : IComparer<Persona>
  {
    public int Compare(Persona x, Persona y)
    {
       return x.Nombre.CompareTo(y.Nombre);
    }
  }
```

Aquí estaríamos utilizando el comparador de Persona como el comparador de Profesor lo cual una instancia de una clase más grande está siendo tomada como una instancia de una clase más pequeña

La contravarianza también se pude dar en delegados un ejemplo de estos es el siguiente

```
Action<persona> act = (newPersona) => {

        Console.WriteLine (newpersona.Nombre) ;

} ;

Action<Profesor> act2 = act;
```

En este caso crearemos una función delegada que recibe como parámetro una persona y luego se lo asignamos a otra función que acepta parámetro de tipo Profesor como Action es contravariante la operación es permitida

El resto de los incisos que se proponía realizar están resueltos en el Solution que se adjunta este archivo.

Explicar el funcionamiento del siguiente código:

```
static void PrintByConsole(Action<Action<Person>> person) {
   person(x => Console.WriteLine(x.Name))
}
....
PrintByConsole(x => new Student() { Name = 'Pedro' });
```

El método recibe un delegado Action el cuál es contravariante, y este recibe otro delegado Action como entrada. Los delegados anidados aceptarán cualquier argumento de tipo Person. El método invoca al delegado y pone como parémetro una función lambda que recibe un Person e imprime su nombre. Lo que nos preguntámos es si podemos usar Action<Action<Student>> como argumento de PrintByConsole.

Si escribimos el código y lo corremos en C# superior a la versión 4, este funcionará correctamente. Esto se debe a que estamos usando Action<T> dos veces y como este es contravariante al anidarlo con otro Action<T> el functor o morfismo Action<Action<T>> se vuelve "covariante", es decir, podemos pasar tipos mas derivados como parámetro.

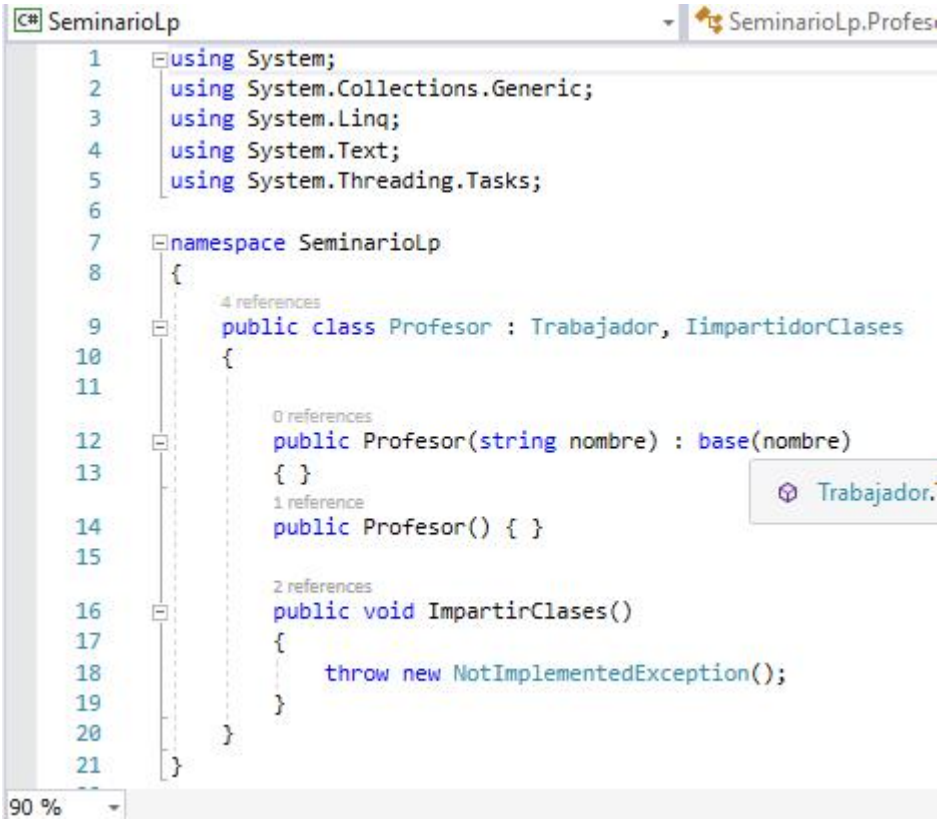
ANEXO 1:

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace SeminarioLp
8   {
9       public class Persona
10      {
11          private string nombre;
12          public string Nombre { get => nombre; set => nombre = value; }
13
14          public Persona(string _nombre)
15          {
16              nombre = _nombre;
17          }
18          public Persona() { }
19      }
20  }
21
```

Anexo 2:

```
SeminarioLp                                   SeminarioLp.Estudiant
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace SeminarioLp
8   {
        6 references
9       public class Estudiante : Persona, IrecibidorClases
10      {
            1 reference
11          public Estudiante(string nombre) : base(nombre)
12          {
13          }
            0 references
14          public Estudiante() { }
15
            1 reference
16          public void RecibirClases()
17          {
18          }
19      }
20  }
21
90 %
```

ANEXO 3:


SeminarioLp
SeminarioLp.Profes
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace SeminarioLp
{
4 references
public class Profesor : Trabajador, IimpartidorClases
{
0 references
public Profesor(string nombre) : base(nombre)
{ }
Trabajador.
1 reference
public Profesor() { }
2 references
public void ImpartirClases()
{
throw new NotImplementedException();
}
}
}
90 %

Anexo 4:

SeminarioLp — SeminarioLp.Trabaja

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace SeminarioLp
8   {
        3 references
9       public class Trabajador : Persona, IcobradorSalario
10      {
            1 reference
11          public Trabajador(string nombre) : base(nombre) { }
            0 references
12          public Trabajador() { }
            2 references
13          public void CobrarSalario()
14          {
15              throw new NotImplementedException();
16          }
17      }
18  }
19
```

90 %

Anexo 5:

SeminarioLp — SeminarioLp.AlumnoAyudante

```
7   namespace SeminarioLp
8   {
        2 references
9       class AlumnoAyudante : Estudiante, IimpartidorClases, IcobradorSalario
10      {
            0 references
11          public AlumnoAyudante(string nombre) : base(nombre) { }
            0 references
12          public AlumnoAyudante() { }
            2 references
13          public void CobrarSalario()
14          {
15              throw new NotImplementedException();
16          }
17
            2 references
18          public void ImpartirClases()
19          {
20              throw new NotImplementedException();
21          }
22      }
23  }
24
```

Anexo 6:

SeminarioLp

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace SeminarioLp
8  {
       2 references
9      public interface IcobradorSalario
10     {
           2 references
11         void CobrarSalario();
12     }
13 }
14
```

Anexo 7:

SeminarioLp

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace SeminarioLp
8  {
       2 references
9      public interface IimpartidorClases
10     {
           2 references
11         void ImpartirClases();
12     }
13 }
14
```

Anexo 8:

SeminarioLp

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace SeminarioLp
8  {
       1 reference
9      public interface IrecibidorClases
10     {
           1 reference
11         void RecibirClases();
12     }
13 }
14
```