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*, *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 refiere a cuando hereda de una clase normalmente, pero la clase que hereda re-define o mejor dicho

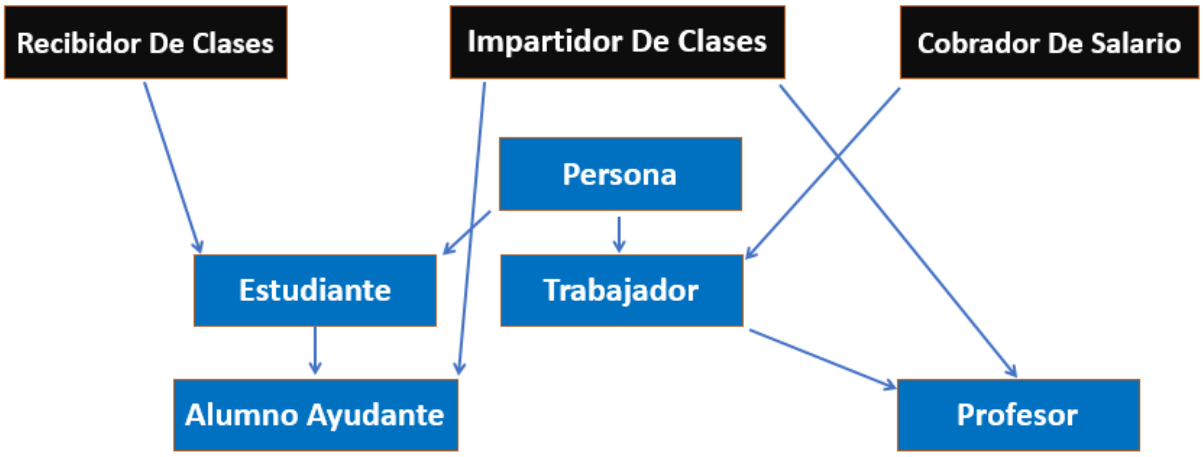
sobreescribe (override) a él/los métodos/propiedades de la clase base. Podemos verlo en un ejemplo sencillo que se ilustra en el Anexo 1.

El polimorfismo por abstracción hace referencia a las clases abstractas, estas clases no 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 el Anexo 2 donde cómo podemos ver tanto la clase Estudiante que hereda de Persona como en la clase Persona existen un método común llamado Actividad. El mismo en la clase derivada Estudiante sobreescribe el método abstracto con el mismo nombre (nótese que el método no contiene nada en su interior) de la clase Persona.

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…)

Inciso A



Varianza:

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. 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. 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.

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

ANEXO 1:

```
class Persona
{
    protected string nombre;

    1 reference
    public Persona(string nombre)
    {
        this.nombre = nombre;
    }

    1 reference
    public virtual string Identficar()
    {
        return nombre + " es persona";
    }
}

class Estudiante : Persona
{
    0 references
    public Estudiante(string nombre) : base(nombre) { }

    1 reference
    public override string Identficar()
    {
        return base.nombre + " es estudiante";
    }

}
```

ANEXO 2:

```
abstract class Persona
{
    protected string nombre;

    1 reference
    public Persona(string nombre)
    {
        this.nombre = nombre;
    }

    1 reference
    public abstract string Actividad();
}

class Estudiate: Persona
{
    0 references
    public Estudiate(string nombre): base(nombre) { }

    1 reference
    public override string Actividad()
    {
        return base.nombre + " es estudiante";
    }
}
```

ANEXO 3:

```
interface IEtiqueta
{
    2 references
    string Es();
    2 references
    string QueHace();
}

abstract class Persona
{
    protected string nombre;

    2 references
    public Persona(string nombre)
    {
        this.nombre = nombre;
    }
}
```

```
class Estudiante : Persona, IEtiqueta
{
    0 references
    public Estudiante(string nombre) : base(nombre) { }

    2 references
    public string Es()
    {
        return this.nombre + " es estudiante";
    }

    2 references
    public string QueHace()
    {
        return ", estudia";
    }
}

class Profesor : Persona, IEtiqueta
{

    0 references
    public Profesor(string nombre) : base(nombre) { }

    2 references
    public string Es()
    {
        return nombre + " es profesor";
    }

    2 references
    public string QueHace()
    {
        return ", enseña";
    }
}
```