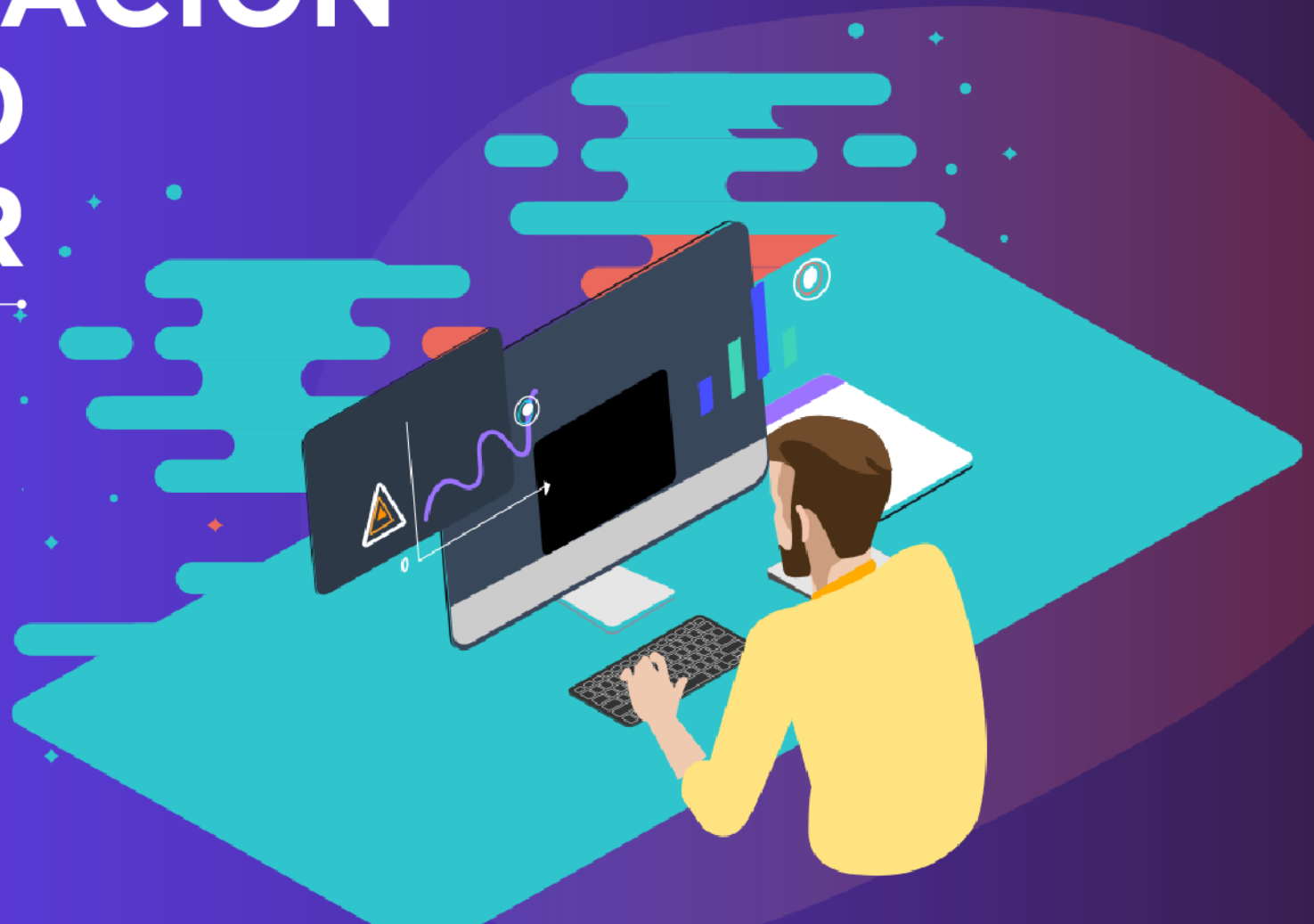
PROGRAMA

.NET ESPECIALIZACIÓN: NET

ESPECIALIZACIÓN ASP.NET 5.0 DEVELOPER



CURSO VIRTUAL

Fundamentos de Programación

Programación orientada a objetos

Instructor: Erick Aróstegui
earostegui@galaxy.edu.pe

GALAXY TRAINING

TEMAS

01	Clases, estructuras, atributos, constructores, métodos
02	Herencia, sobrecarga de métodos y constructores
03	Interfaces y métodos genéricos
04	Clases internas., abstractas y finales
05	Asociación de clases

Clases

Es un *tipo de referencia*. Al declarar una variable de un tipo de referencia en tiempo de ejecución, esta contendrá el valor null hasta que se cree una instancia de la clase mediante el operador new o se le asigne un objeto de un tipo compatible que se ha creado en otro lugar

```
//Declarando un objeto de tipo myclass
MiClase mc = new MiClase();
//Declarando otros objetos del mismo tipo
MiClase mc2 = mc;
```

Declarar Clases

Las clases se declaran mediante la palabra clave class seguida por un identificador único. La palabra clave class va precedida del nivel de acceso.

Como en este caso se usa public, cualquier usuario puede crear instancias de esta clase. El nombre de la clase sigue a la palabra clave

```
//[modificador de acceso] - [identificador de clase]
0 referencias
public class Clases
{
    // campos, propiedades, métodos y eventos van aquí..
}
```

Estructuras

Un tipo struct es una construcción de programación utilizado para definir tipos personalizados. Los tipos struct se usan para encapsular pequeños grupos de variables relacionadas y representadas como un solo elemento.

Se inicializa struct con la palabra clave new, se llama al constructor predeterminado sin parámetros y, a continuación, se establecen los miembros de la instancia.

```
{
    int ID, contraseña;
    0 referencias
    public Cliente(int clienteID, int clienteContraseña)
    {
        ID = clienteID;
            contraseña = clienteContraseña;
    }
}
0 referencias
class TestCliente
{
    0 referencias
    static void Main()
    {
        Cliente c1 = new Cliente();// usando el constructor predeterminado
    }
```

Recordatorio de Estructuras

La palabra clave struct puede ser precedida por un modificador de acceso. En la declaración podemos utilizar los siguientes modificadores de acceso. **Public**, esté disponible para el código de cualquier Assembly. **Internal**, estará disponible dentro del mismo Assembly. **Private**, permite que el código solo esté disponible dentro de la estructura de datos que contiene.

- Los struct son de tipo valor y las clases son tipo referencias.
- Las estructuras pueden tener métodos, propiedades, operadores, eventos.
- Una estructura puede implementar una o más interfaces
- Los miembros de una estructura no pueden ser especificados como abstract, virtual o protected.
- Las estructuras no pueden ser heredadas, así como tampoco, pueden heredar.

Atributos

Los atributos son clases que se heredan de la clase base Attribute. Cualquier clase que se hereda de Attribute puede usarse como una especie de "etiqueta" en otros fragmentos de código.

Los atributos proporcionan una manera de asociar la información con el código de manera declarativa. También pueden proporcionar un elemento reutilizable que se puede aplicar a diversos destinos.

Crear Atributos

Crear un atributo es tan sencillo como heredarlo de la clase base Attribute

```
public class MySpecialAttribute: Attribute
{

}
```

Con lo anterior, ahora puedo usar ([MySpecial] o [MySpecialAttribute]) como un atributo en otra parte de la base de código.

```
[MySpecial]
0 referencias
public class SomeOtherClass
{

}
```

Constructores

Cada vez que se crea una clase o estructura, se llama a su constructor. Una clase o estructura puede tener varios constructores. Los constructores permiten al programador establecer valores predeterminados, limitar la creación de instancias y escribir código flexible y fácil de leer.

CARACTERÍSTICAS

- Tiene el mismo nombre de la clase.
- Es el primer método que se ejecuta.
- Se ejecuta en forma automática.
- No puede retornar datos.
- Se ejecuta una única vez.

```
class Constructor
{
    int a, b;
    //DECLARAMOS EN CONSTRUCTOR
    1 referencia
    public Constructor(int x, int y)
    {
        a = x;
        b = y;
    }
    1 referencia
    public int Suma()
    {
        return a + b;
    }
    1 referencia
    public int multipli()
    {
        return a * b;
    }
    0 referencias
    class principal
      0 referencias
    { static void Main(string[] args)
        {
            //CREAMOS EL OBJETO DE LA CLASE Y LE PASAMOS LOS PARAMETROS AL CONSTRUCTOR
            Constructor obj = new Constructor(10, 20);
            Console.WriteLine("La suma es: " + obj.Suma());
            Console.WriteLine("La multiplicacion es: " + obj.multipli());
            Console.ReadKey();
        }
```

Métodos

Un método es un bloque de código que contiene una serie de instrucciones. Un programa hace que se ejecuten las instrucciones al llamar al método y especificando los argumentos de método necesarios.

Los métodos se declaran en una clase, struct o interfaz especificando el nivel de acceso, como public o private , modificadores opcionales como abstract o sealed, el valor devuelto, el nombre del método y cualquier parámetro de método.

Firmas de Métodos

Los parámetros de método se encierran entre paréntesis y se separan por comas. Los paréntesis vacíos indican que el método no requiere parámetros. Esta clase contiene cuatro métodos:

```
abstract class Motocicleta
{
    //cualquiera puede llamar a esto
    0 referencias
    public void Motor() {/*declaraciones de método aquí*/ }

    //solo las clases derivadas pueden llamar a esto
    0 referencias
    protected void Gas(int galones) {/*declaraciones de método aquí*/ }

    //las clases derivadas pueden anular la implementación de la clase base
    0 referencias
    public virtual int  Manejar(int millas, int velocidad)
    {/*declaraciones de método aquí*/ return 1; }

    //las clases derivadas deben implementar esto
    0 referencias
    public abstract double VelociMaxi();
}
```

Acceso a Métodos

Llamar a un método en un objeto es como acceder a un campo. Después del nombre del objeto, agregue el nombre del método y paréntesis. Los argumentos se enumeran entre paréntesis y están separados por comas.

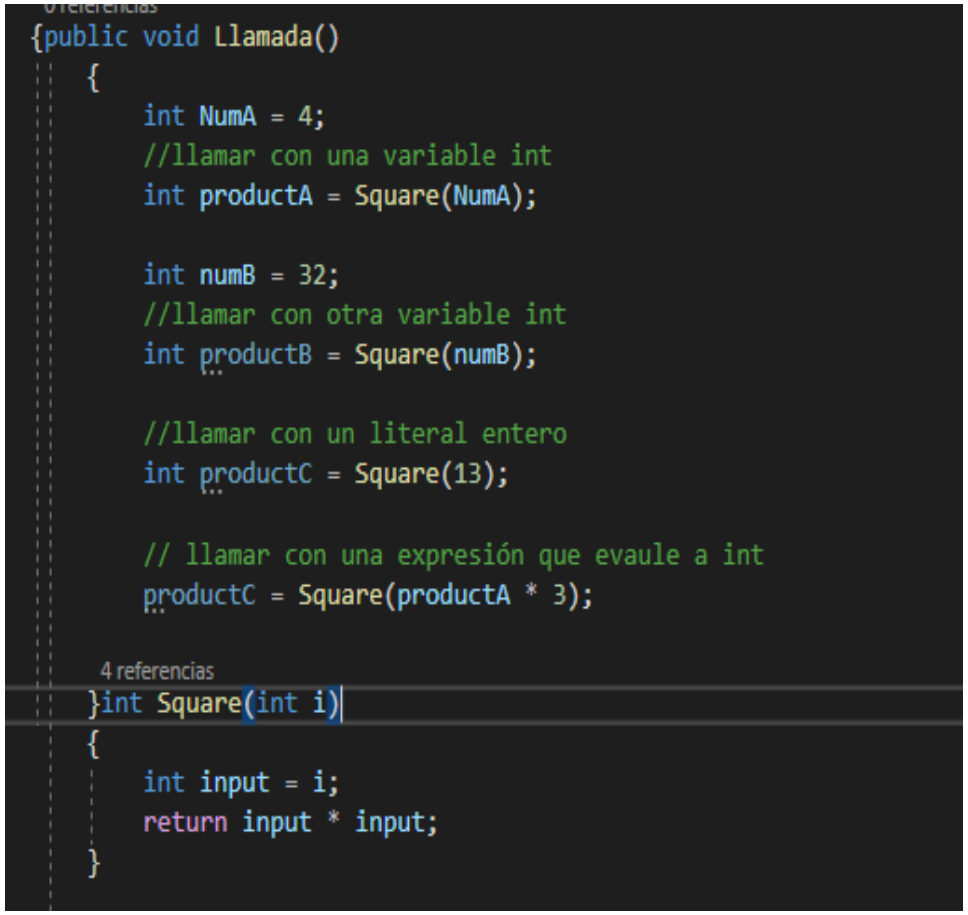
```
class TestMotocicleta: Motocicleta
{
    2 referencias
    public override double VelociMaxi()
    {
        return 108.4 ;
    }
    0 referencias
    static void Main()
    {
        TestMotocicleta moto = new TestMotocicleta();

        moto.Motor();
        moto.Gas(25);
        moto.Manejar(7, 15);
        double speed = moto.VelociMaxi();
        Console.WriteLine("Mi velocidad maxima es: ", speed);

    }
```

Parámetros de Métodos

Si el código de llama al método, proporciona valores concretos denominados argumentos para cada parámetro. Deben ser compatibles con el tipo de parámetro, pero el nombre del argumento (si existe) utilizado en el código de llamada no tiene que ser el mismo que el parámetro con nombre definido en el método.



Pasar por referencias frente a pasar por valor

Ahora, si se pasa un objeto basado en este tipo a un método, también se pasa una referencia al objeto. En el ejemplo siguiente se pasa un objeto de tipo SampleRefType al método ModifyObject

```
public static void TestRefType()
{
    RefeMuestra rt = new RefeMuestra();
    rt.value = 44;
    ObjeModifi(rt);
    Console.WriteLine(rt.value);
}
1 referencia
static void ObjeModifi(RefeMuestra obj)
{
    obj.value = 33;
}
```

Delegados

Un delegado es un tipo que representa referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Cuando se crea una instancia de un delegado, puede asociar su instancia a cualquier método mediante una signatura compatible y un tipo de valor devuelto.

Los delegados se utilizan para pasar métodos como argumentos a otros métodos.

```
public delegate int RealizaCalculo(int x, int y);
```

Información general Delegados

_Los delegados permiten pasar los métodos como parámetros.
_Los delegados pueden usarse para definir métodos de devolución de llamada.
_No es necesario que los métodos coincidan exactamente con el tipo de delegado. Para obtener más información, consulte Usar varianza en delegados

Herencias

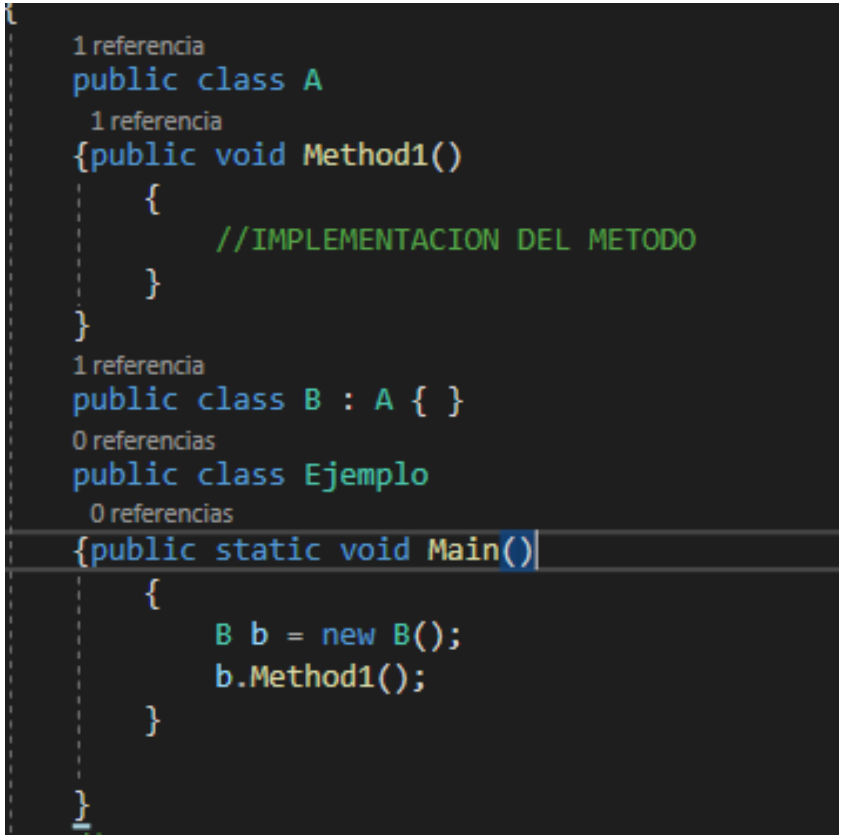
Permite definir una clase base, que proporciona funcionalidad específica (datos y comportamiento), así como clases derivadas, que heredan o invalidan esa funcionalidad. Permite definir una clase secundaria que reutiliza (hereda), amplía o modifica el comportamiento de una clase primaria.

Los miembros privados solo son visible en las clases derivadas que están anidadas en su clase base. De lo contrario, no son visibles en las clases derivadas.

```
0 referencias
class Hrencias
{
    2 referencias
    public class A
    {
        private int value = 10;

        0 referencias
        public class B: A
        {
            1 referencia
            public int GetValue()
            {
                return this.value;
            }
        }
    }
    0 referencias
    public class Ejem
    {
        0 referencias
        public static void Main(string[] args)
        {
            var b = new A.B();
            Console.WriteLine(b.GetValue());
        }
    }
```

Los miembros públicos son visibles en las clases derivadas y forman parte de la interfaz pública de dichas clases. Los miembros públicos heredados se pueden llamar como si se definieran en la clase derivada.



Sobrecarga de Métodos

Es la creación de varios métodos con el mismo nombre, pero con diferentes firmas y definiciones. Se utiliza el numero y tipo de argumentos para seleccionar que definición de método ejecutar

```
public static int Nume(int num1, int num2)
{
    return Nume(num1, num2 ,0);
}
1 referencia
public static int Nume(int num1, int num2, int num3)
{
    return Nume(num1, num2, num3 ,0);
}
1 referencia
public static int Nume(int num1, int num2, int num3, int num4)
{
    return num1+ num2+ num3+ num4;
}
/* METODOS
```

Sobrecarga de Constructores

Como hemos visto el constructor es un método y como tal podemos sobrecargarlo, es decir definir varios constructores con distintas cantidades o tipos de parámetros.

```
class Titulo
   {
       private string titu;
       private int colum, fila;
       1 referencia
       public Titulo(string t)
       {
           titu = t;
           colum = 1;
           fila = 1;
       }
       1 referencia
       public Titulo(string t, int col, int fil)
       {
           titu = t;
           colum = col;
           fila = fil;
       }
       2 referencias
       public void imprimir()
       {
           Console.SetCursorPosition(colum, fila);
           Console.Write(titu);
       }
       0 referencias
       static void Main(string[] args)
       {
           Titulo t1 = new Titulo("HOLA MUNDO");
           t1.imprimir();
           Titulo t2 = new Titulo("HOLA MUNDO", 40, 12);
           t2.imprimir();
           Console.ReadKey();
       }
```

Interfaces

Una interfaz contiene definiciones para un grupo de funcionalidades relacionadas que debe implementar una clase o estructura no abstracta .
Además, debe usar una interfaz si desea simular la herencia de estructuras, porque en realidad no pueden heredar de otra estructura o clase.

```
0 referencias
interface IEquatable<T>
{
    0 referencias
    bool Equals(T obj);
}
```

Una interfaz se puede declarar una propiedad que tiene un descriptor de acceso get . La clase que implementa la interfaz puede declarar la misma propiedad con un descriptor de acceso get y set

```
2 referencias
public class Carro : IEquatable<Carro>
{
    2 referencias
    public string Marca { get; set; }
    2 referencias
    public string Modelo { get; set; }
    2 referencias
    public string Año { get; set; }
    //implementación de IEquatable<T> Interface
    0 referencias
    public bool Equals(Carro carro)
    {
        return (this.Marca, this.Modelo, this.Año) ==
            (carro.Marca, carro.Modelo, carro.Año);
    }

}
```

RESUMEN DE INTERFACES

- Una interfaz suele ser como una clase base abstracta con solo miembros abstractos. Cualquier clase o estructura que implemente la interfaz debe implementar todos sus miembros.
- No se puede crear una instancia de una interfaz directamente. Sus miembros son implementados por cualquier clase o estructura que implemente la interfaz.
- Una clase o estructura puede implementar múltiples interfaces. Una clase puede heredar una clase base y también implementar una o más interfaces.

Métodos Genéricos

Un método genérico es un método que se declara con parámetros de tipo

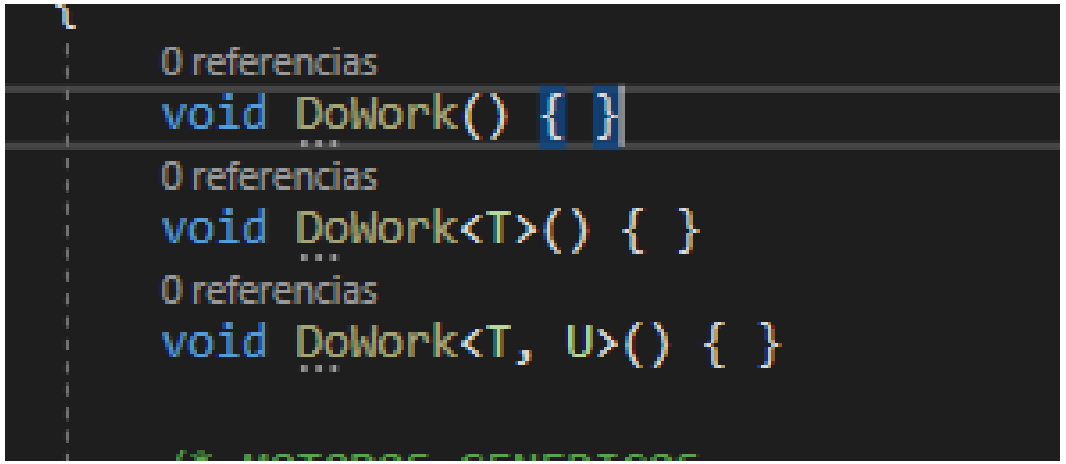
```
0 referencias
public static void TestSwamp()
{
    int a = 1, b = 2;
    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
}
```

```
0 referencias
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

Dentro de una clase genérica, los métodos no genéricos pueden tener acceso a los parámetros de tipo de nivel de clase

```
class EjemClass<T>
{
    0 referencias
    void Swap(ref T lhs, ref T rhs) { }
}
```

Los métodos genéricos pueden sobrecargarse en varios parámetros de tipo. Por ejemplo, todos los métodos siguientes pueden ubicarse en la misma clase:



Clases Abstracta

Utilizamos el modificador abstract para definir clases o miembros de clases (métodos, propiedades, events, o indexers) para indicar que esos miembros deben ser implementados en las clases que derivan de ellas.

Cuando declaramos una clase como abstract estamos indicado que va a ser utilizada como clase base de otras, ya que ella misma no se puede instanciar. Una clase abstracta puede contener miembros abstractos como no abstractos, y todos los miembros deben ser implementados en la clase que la implementa.

Uso de Clases Abstracta

Las clases abstractas se utilizan como clase base y es ahí donde radica todo su potencial. Ya que nos da la posibilidad de detectar el código común y extraerlo en ella.

```
public class D
{
    2 referencias
    public virtual void HacerTrabajo(int i)
    {
        //IMPLEMENTACION ORIGINAL
    }
}
1 referencia
public abstract class E : D
{
    2 referencias
    public abstract override void HacerTrabajo(int i);
}
0 referencias
public class F : E
{
    2 referencias
    public override void HacerTrabajo(int i)
    {
        // NUEVA IMPLEMENTACION
    }
```

Clases Finales

Los finalizadores (también denominados destructores) se usan para realizar cualquier limpieza final necesaria cuando el recolector de elementos no utilizados recopila una instancia de clase.

```
public class Destroyer
{
    1 referencia
    public override string ToString() => GetType().Name;

    ~Destroyer() => Console.WriteLine($"El {ToString()} destructor se esta ejecutando");
}
```

Asociación de clases

Uno a uno: para representar este tipo de relación incluya EntitySet<TEntity>en ambos lados.

Uno a muchos: Si el lado muchos no tiene una gran cantidad de elementos y estos, además, ya están creados, podemos utilizar una lista de botones de input de tipo check

Varios a varios: en las relaciones varios a varios, la clave principal de la tabla de vínculos (también denominada tabla de Unión) suele estar formada por un compuesto de las claves externas de las otras dos tablas.

Uno a Uno

```
{
    0 referencias
    public int Id { get; set; }
    0 referencias
    public string UsuName { get; set; }
    0 referencias
    public int OpenIdInfo { get; set; }


}
```

Uno a Varios

```
public class User
{
    [HiddenInput(DisplayValue = false)]
    public int Id { get; set; }

    [Required(ErrorMessage = "Por favor introduzca un nombre")]
    [DataType(DataType.Text)]
    public string Name { get; set; }

    [Required(ErrorMessage = "Por favor introduzca un nombre")]
    public virtual Enterprise Enterprise { get; set; }

    [Required(ErrorMessage = "Por favor especifique su contraseña")]
    [DataType(DataType.Password)]
    public string Password { get; set; }
}
```

Muchos a Muchos

```
public class Persona
{
    0 referencias
    public int PersonaId { get; set; }
    0 referencias
    public string Nombre { get; set; }
    0 referencias
    public virtual ICollection<Carro> Carro { get; set; }
}
1 referencia
public class Carro
{
    0 referencias
    public int CarroId { get; set; }
    0 referencias
    public string Licencia { get; set; }
    0 referencias
    public virtual ICollection<Persona> Owners { get; set; }
}
```

PREPÁRATE
PARA SER EL
MEJOR

+ ENTREMIENTO
EXPERIENCIA

BIENVENIDOS.

GRACIAS
POR TU PARTICIPACIÓN

Por favor, bríndanos tus comentarios
y sugerencias para mejorar nuestros servicios.

GALAXY
TRAINING


GALAXY
TRAINING