
PROGRAMACIÓN ORIENTADA A OBJETOS

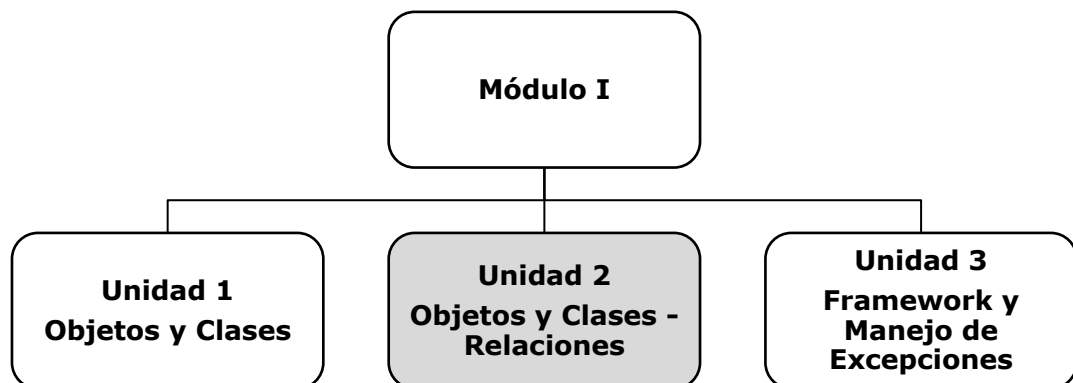
Módulo I

Conocimiento de la Teoría Orientada a Objetos, Frameworks y la definición, construcción y uso de las clases.

Unidad 2

Objetos y Clases - Relaciones

Docente titular y autor de contenidos: Prof. Ing. Darío Cardacci



Presentación

En esta unidad abordaremos los temas referidos a como el entorno de desarrollo y el .NET Framework utilizan los Eventos, profundizaremos aspectos relacionados con la unidad uno y analizaremos que tipos de relaciones existen entre las clases, los objetos y para que sirven.

En particular haremos énfasis en los distintos tipos de clase y para qué se usan y de esta manera poder potenciar nuestros desarrollos.

Esperamos que luego de analizar estos temas Ud. pueda percibir el aporte que los mismos le otorgan a los desarrollos orientados a objetos.

Por todo lo expresado hasta aquí es que esperamos que usted, a través del estudio de esta unidad, se oriente hacia el logro de las siguientes metas de aprendizaje:

- Comprender la noción de evento a través del análisis de sus particularidades y su uso.
- Distinguir los distintos tipos de clases.
- Analizar y reconocer las relaciones que se pueden establecer entre clases.
- Analizar y reconocer las relaciones que se pueden establecer entre objetos.

Los siguientes contenidos conforman el marco teórico y práctico que contribuirá a alcanzar las metas de aprendizaje propuestas:

Eventos. Suscripción a eventos. Suscripción a eventos utilizando el IDE. Suscripción a eventos mediante programación. Suscripción a eventos mediante métodos anónimos. Publicación de eventos. Desencadenar eventos.

Modificadores de acceso. Clases abstractas, selladas y estáticas. Miembros estáticos en clases estáticas.

Relaciones básicas entre clases. "Generalización-Especialización", "Parte de".

Relaciones derivadas entre clases. Herencia. Herencia simple. Herencia múltiple. Teoría de Tipos. Tipos anónimos.

Sobrescritura de métodos. Métodos virtuales. Polimorfismo.

Agregación. Simple y con contención física.

Asociación y relación de Uso.

Elementos que determinan la calidad de una clase: acoplamiento, cohesión, suficiencia, compleción y primitivas.

Relaciones entre objetos: enlace y agregación.

Acceso a la clase base desde la clase derivada.

Acceso a la instancia actual de la clase.

A continuación, le presentamos un detalle de los contenidos y actividades que integran esta unidad. Usted deberá ir avanzando en el estudio y profundización de los diferentes temas, realizando las lecturas requeridas y elaborando las actividades propuestas, algunas de desarrollo individual y otras para resolver en colaboración con otros estudiantes y con su profesor tutor.

Índice de contenidos y Actividades

1. Eventos.



Lectura requerida

- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007. Capítulo XIII.
- <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/>

2. Modificadores de Acceso



Lectura requerida

- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007. Capítulo XI.
- <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/modifiers>

3. Herencia y Teoría de Tipos.



Lectura requerida

- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007. Capítulo X.
- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007. Capítulo XI.
- <https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/inheritance>
- <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/>

4. Sobrescritura y Polimorfismo



Lectura requerida

- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007. Capítulo XI.
- <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override>

5. Agregación



Lectura requerida

- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007. Capítulo IV.

6. Asociación y Relación de Uso



Lectura requerida

- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007. Capítulo IV.

7. Elementos que determinan la calidad de una clase



Lectura requerida

- Booch, Grady – Cardacci, Dario Guillermo. Orientación a Objetos. Teoría y Práctica. Pearson Educación - UAI . Primera Edición. 2013. Capítulo III. Punto 3.6.

8. Relaciones entre Objetos: Enlace y Agregación



Lectura requerida

- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007. Capítulo X.

- <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/access-keywords>

Para el estudio de estos contenidos usted deberá consultar la bibliografía que aquí se menciona:

BIBLIOGRAFÍA OBLIGATORIA

- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007.



Links a temas de interés

Desarrollo .NET

<https://docs.microsoft.com/en-us/dotnet/csharp/index>

<https://docs.microsoft.com/en-us/dotnet/standard/get-started>

Lo/a invitamos ahora a comenzar con el estudio de los contenidos que conforman esta unidad.

1. Eventos.

Los **eventos** conceptualmente son mecanismos que permiten que un objeto "reaccione" ante un estímulo externo. Es un mecanismo de enlace tardío que proporciona versatilidad a los desarrollos.

Para comprender mejor qué son y cómo funcionan los eventos haremos una mínima referencia al concepto de delegado, tema que se ampliará en las unidades posteriores. Los delegados proporcionan un mecanismo de **enlace tardío** en .NET. La vinculación tardía significa que se crea un algoritmo donde el código llamador proporciona al menos un método que implementa parte del algoritmo. Esta forma se utiliza desde hace mucho tiempo en programación. Es muy útil cuando por ejemplo un programador desea dejar establecida parte de una funcionalidad, pero quiere dejar la posibilidad de que quien utiliza esa funcionalidad pueda complementarla con una definición o un algoritmo propio.

Por ejemplo, pensemos en que un programador desarrolló un algoritmo que permite acumular objetos de distintos tipos. También desea dejar establecido que esos objetos se pueden ordenar de manera ascendente y descendente. En este punto se nos presentan dos problemas. El primero es que no sabemos por qué criterio se ordenarán, pues eso dependerá de las características que posean los objetos. El segundo es que el ordenamiento deseado puede ser ascendente o descendente. Está claro, que básicamente lo que establece un algoritmo de ordenamiento es lo mismo, independientemente de los aspectos particulares asociados a las "características por la cual se ordena" y el "criterio ascendente o descendente". Esto nos lleva a pensar que sería grandioso que el programador pueda implementar el mecanismo de ordenamiento (core engine sort) y dejar que el usuario del algoritmo pueda definir a partir de su propio algoritmo, que características serán consideradas para ordenar los objetos y el criterio que desea (ascendente o descendente). Básicamente entre otras cosas importantes pero que exceden el actual curso, un delegado nos permite hacer lo mencionado.

Los eventos son como los delegados, un mecanismo de enlace tardío. De hecho, los eventos se basan en el soporte que el framework y los lenguajes de programación le dan a los delegados.

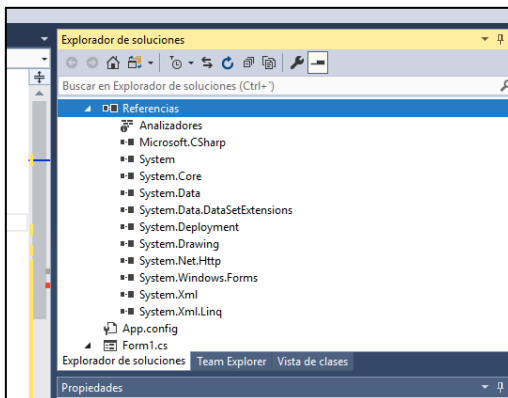
Los eventos son una forma en la que un objeto transmite (a otros objetos interesados en el sistema) que algo ha sucedido. Los objetos pueden suscribirse a un evento y ser notificados cuando se produce el mismo.

La suscripción a un evento también crea un acoplamiento entre los dos objetos (el originador del evento y el receptor del evento). Una buena práctica de programación es asegurarse que el receptor del evento cancela la suscripción cuando ya no esté interesado.

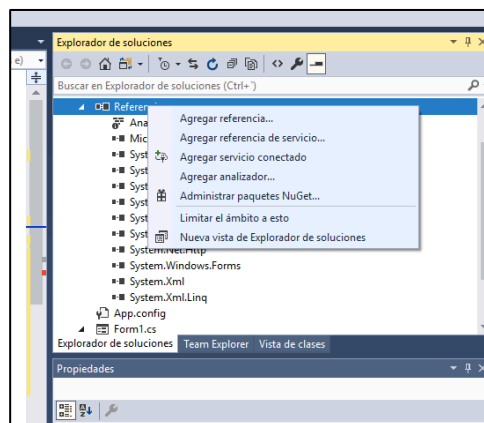
Las pautas que se han seguido cuando se diseñaron los eventos son sencillas pero muy importantes.

La primera es que se produzca un acoplamiento mínimo entre el originador del evento y el receptor. La segunda es considerar que estos dos componentes pueden no estar escritos por el mismo programador o por la misma organización. La tercera es su facilidad de uso. Debería ser muy simple suscribirse a un evento y darse de baja del mismo. Finalmente, los originadores de eventos deberían admitir múltiples suscriptores.

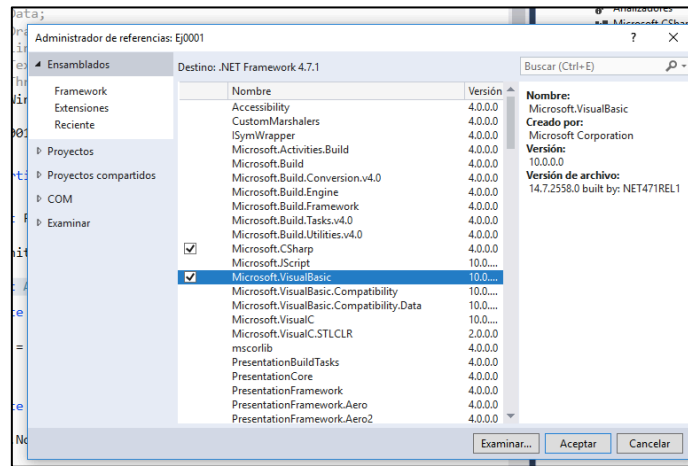
Antes de comenzar a trabajar con eventos, explicaremos como incorporarle a nuestra solución la posibilidad de utilizar la instrucción Inputbox de vb.net. Esto nos facilitará el ingreso de los datos. Debemos considerar que nuestra solución está basada en C#, pero el entorno nos da la posibilidad de incorporar componentes que no se encuentran por defecto incluidos. Para ello vamos a **Referencias**, en el explorador de soluciones:



Luego hacemos clic sobre **Referencias** con el botón derecho del mouse y observaremos:



A continuación clic sobre **Agregar referencia..** y aparece:



Se debe hacer clic sobre **Microsoft.VisualBasic** para que este quede chequeado y paso seguido clic en aceptar.

En el archivo que estamos programando, agregamos **using Microsoft.VisualBasic** como se puede observar a continuación y nuestra aplicación está lista para poder utilizar la instrucción **InputBox()**.

```

1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Data;
5  using System.Drawing;
6  using System.Linq;
7  using System.Text;
8  using System.Threading.Tasks;
9  using System.Windows.Forms;
10 using Microsoft.VisualBasic;
11

```

Suscripción a eventos mediante el IDE de Visual Studio.

Podemos suscribir a eventos desde la ventana **Propiedades**, en la vista **Diseño**. En la parte superior de la ventana **Propiedades**, haga clic en el ícono **Eventos**.

Haga doble clic en el evento que desea crear, por ejemplo, el evento **Click**. Visual C# crea un método de control de eventos vacío y lo agrega al código.

```

private void Form1_Click(object sender, EventArgs e)
{
}

```

También puede agregar manualmente el código en la vista Código.
La línea de código que es necesaria para suscribirse al evento también se genera automáticamente, en el método **InitializeComponent** dentro del archivo **Form1.Designer.cs** del proyecto.

```
this.Click += new System.EventHandler(this.Form1_Click);
```

Para anular la suscripción a este evento solo hay que borrar la línea de código anterior y el método de control **Form1_Click** si no lo utilizará para otra cosa.

Definición, suscripción, desencadenamiento y consumo de un evento estándar por programación.

Retomando el tema que se estaba exponiendo, la sintaxis básica para definir un evento es la siguiente:

```
public event EventHandler CambioEnNombre;
```

La palabra clave es **event** y se acompaña del tipo de evento **EventHandler** y el nombre que le deseamos dar, en este caso **CambioEnNombre**.

En el siguiente ejemplo se podrá observar como, cuando en la clase **Alumno** se realice un cambio en la propiedad **Nombre** se desencadenará el evento **CambioEnNombre**.

```
public class Alumno
{
    public event EventHandler CambioEnNombre;
    private string Vnombre = "";
    public string Nombre
    {
        get { return Vnombre; }
        set
        {
            this.Vnombre = value; CambioEnNombre(this, null);
        }
    }
}
```

Ej0001

En el ejemplo **Ej0001** se observa la clase **Alumno**, la cual posee un **evento** denominado **CambioEnNombre** y una propiedad **Nombre**. La propiedad **Nombre** en la implementación del **set**, posee la instrucción (**CambioEnNombre(this,null)**) que hace que se desencadene el **evento**.

Si observamos la firma de la función que se ejecuta cuando se desencadena un **evento** bien formado, se puede observar que posee dos parámetros. El primero

es de tipo **object** y se denomina **sender** y el segundo es del tipo **EventArgs** o algún subtipo derivado de él, cuyo nombre es **e**. El primero lleva una referencia al objeto que ha provocado que el evento se desencadene, por eso en nuestro ejemplo se observa **this** (el objeto mismo, él mismo), que representa a la instancia actual a la que se le cambió en **Nombre**. El segundo debería llevar toda la información asociada al evento. En nuestro primer ejemplo, consideramos que no es necesario enviar información, debido a esto es que se le pasó un **null**.

Ahora resta ver como se puede consumir el evento que define la clase **Alumno** a través de sus instancias. Para ejemplificarlo, en el Ejemplo **Ej0001**, se ha instanciado dentro de la clase **Form1** un **Alumno**, que es apuntado por la variable **A**. Luego se **suscribe** en **Form1** al **evento CambioEnNombre** del objeto, indicando que función se ejecutará cuando se desencadene el **evento**. Esto se logra por medio de la instrucción:

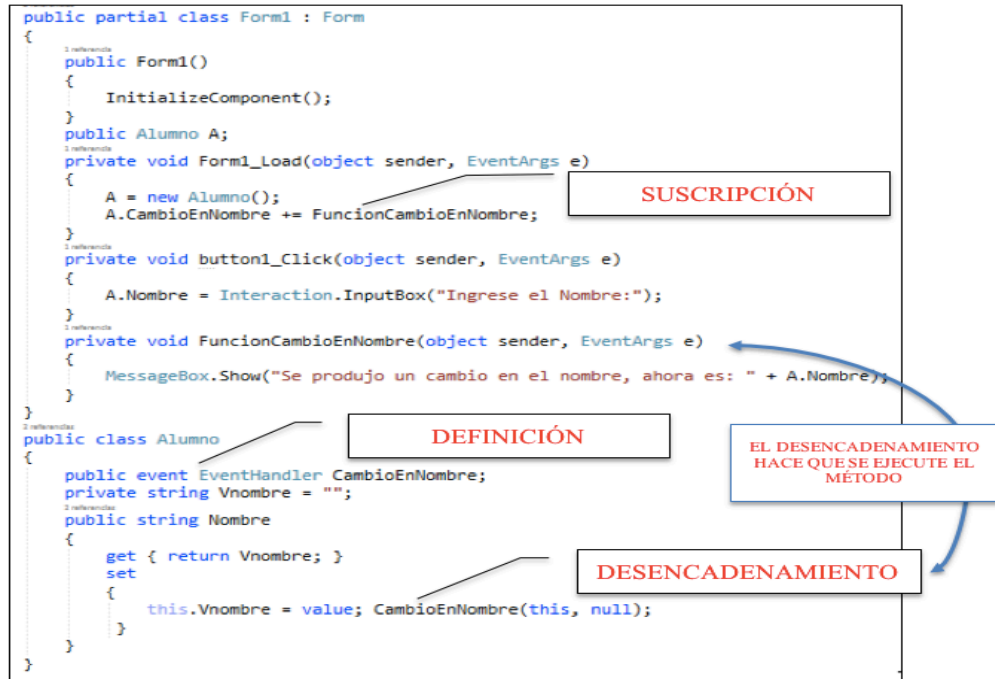
A.CambioEnNombre += FuncionCambioEnNombre

Finalmente la función **FunciónCambioEnNombre** que posee la misma firma que el **evento**, que en este ejemplo es la firma que define por defecto el tipo de evento **EventHandler**, se ejecutará cuando se desencadene el **evento CambioEnNombre** y mostrará el mensaje que posee programado como parte de su implementación.

```
public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }
    public Alumno A;
    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        A = new Alumno();
        A.CambioEnNombre += FuncionCambioEnNombre;
    }
    1 referencia
    private void button1_Click(object sender, EventArgs e)
    {
        A.Nombre = Interaction.InputBox("Ingrese el Nombre:");
    }
    1 referencia
    private void FuncionCambioEnNombre(object sender, EventArgs e)
    {
        MessageBox.Show("Se produjo un cambio en el nombre, ahora es: " + A.Nombre);
    }
}
```

Ej0001

Esquematicamente:



Ej0001

Cancelación de la suscripción a un evento por programación.

La cancelación a la suscripción de un evento es muy sencilla. Como se observa en el siguiente fragmento de código, correspondiente al Ejemplo **Ej0002**, la instrucción **A.CambioEnNombre -= FuncionCambioEnNombre;** utilizada en la función **Button1_Click**, provoca la **desafectación al evento**, o lo que es lo mismo deja de tener efecto la asociación por la cual cuando se desencadena el **evento CambioEnNombre**, se ejecuta el procedimiento **FuncionCambioEnNombre**.

```

public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }
    public Alumno A;
    2 referencias
    private void Form1_Load(object sender, EventArgs e)
    {
        A = new Alumno();
        // Suscripción al Evento
        A.CambioEnNombre += FuncionCambioEnNombre;
    }
    1 referencia
    private void button1_Click(object sender, EventArgs e)
    {
        A.Nombre = Interaction.InputBox("Ingrese el Nombre:");
        // Cancelación a la suscripción al evento
        A.CambioEnNombre -= FuncionCambioEnNombre;
    }
    2 referencias
    private void FuncionCambioEnNombre(object sender, EventArgs e)
    {
        MessageBox.Show("Se produjo un cambio en el nombre, ahora es: " + A.Nombre);
    }
}

```

Ej0002

En el código anterior se observa que al oprimir el botón **button1**, se solicita el ingreso de un nombre que modificará el estado del objeto apuntado por la variable **A**. Esto provocará que se desencadene el evento de acuerdo a lo visto en el ejemplo **Ej0001**, luego se observa la línea de código que utiliza el operador **-=** para cancelar la suscripción al **evento**. Si se vuelve a oprimir el botón **button1**, el programa nuevamente solicitará que se ingrese un nombre, pero luego de ello no se desencadenará ningún evento porque el mismo había sido desafectado con anterioridad.

En el siguiente fragmento de código del mismo ejemplo **Ej0002**, se puede observar una forma diferente de lograr que el **evento** se desencadene. Colocando: `CambioEnNombre?.Invoke(this,null);` logramos que el evento se desencadene. Existe una ventaja en hacerlo de esta manera respecto a la forma utilizada en el ejemplo **Ej0001**, y es que si al intentar desencadenarlo el evento no está delegado a ningún procedimiento, ese error es atrapado. Si se desea utilizar la forma anterior (`CambioEnNombre(this,null);`), antes de desencadenar el evento, deberíamos verificar si **CambioEnNombre != null**, con el objetivo de no invocar algo que está apuntando a null y provocar de esta manera una **Exception**.

```

public class Alumno
{
    public event EventHandler CambioEnNombre;
    private string Vnombre = "";
    public string Nombre
    {
        get { return Vnombre; }
        set
        {
            this.Vnombre = value;
            CambioEnNombre?.Invoke(this, null);
        }
    }
}

```

Ej0002

Como conclusión podemos expresar que con los operadores `+=` y `-=` podemos producir la suscripción a un evento o cancelarla respectivamente.

Definición, suscripción, desencadenamiento y consumo de un evento con argumento personalizado.

Para trabajar con eventos que posean un argumento personalizado, lo primero que se debe hacer es generar una clase que represente al argumento que deseamos. Para lograr esto se genera una clase que herede de `EventArgs`, como se observa en el siguiente código del ejemplo **Ej0003**.

```

public class DatosCambioEnNombreEventArgs : EventArgs
{
    private string vNombre = "";
    private string vHora = "";
    public DatosCambioEnNombreEventArgs(string pNombre)
    {
        this.vNombre = pNombre;
        this.vHora = DateTime.Now.ToShortTimeString();
    }
    public string Nombre { get { return this.vNombre; }}
    public string HoraEvento { get { return this.vHora; }}
}

```

Ej0003

Esta clase puede tener cualquier nombre, pero las buenas prácticas de programación indican que debe finalizar con **EventArgs**. En nuestro ejemplo se puede observar como la clase denominada: **DatosCambioEnNombreEventArgs** hereda de **EventArgs**: `public class DatosCambioEnNombreEventArgs : EventArgs`.

Además, implementa un constructor que dejará pasar el nombre del alumno cuando se instancie esta clase, lo colocará en la variable **vNombre**. También colocará la hora del sistema en la variable **vHora**.

Se pueden observar dos propiedades que permiten consultar el nombre del alumno que arribó a través del constructor y la hora en que se instanció el objeto que representará al argumento personalizado del evento.

Al declarar el evento en la clase **Alumno**, la firma cambia y se puede observar en el código siguiente que se hace uso de un **EventHandler** genérico (**EventHandler<>**). El código queda expresado como:

```
public event EventHandler <DatosCambioEnNombreEventArgs> CambioEnNombre;
```

```
public class Alumno
{
    public event EventHandler <DatosCambioEnNombreEventArgs> CambioEnNombre;
    private string Vnombre = "";
    public string Nombre
    {
        get { return Vnombre; }
        set
        {
            this.Vnombre = value;
            CambioEnNombre?.Invoke(this, new DatosCambioEnNombreEventArgs(this.Nombre));
        }
    }
}
```

Ej0003

También cambia la línea de código donde se programa el desencadenamiento del evento. Como puede observarse en el ejemplo **Ej0003**, el parámetro que antes era de tipo **EventArgs**, ahora es del tipo personalizado por el programador: **DtosCambioEnNombreEventArgs**.

```
CambioEnNombre?.Invoke(this, new DatosCambioEnNombreEventArgs(this.Nombre));
```

Otro detalle a observar es el cambio de firma en el método donde se delega para colocar el código que se ejecutará cuando el evento se desencadene.

```
private void FuncionCambioEnNombre(object sender, DatosCambioEnNombreEventArgs e)
```

También amerita destacar, si bien no se ha alterado, que el método mencionado se encuentra en el mismo espacio dónde se instanció el objeto **Alumno** (poseedor del evento).

```

public Form1()
{
    InitializeComponent();
}
public Alumno A;
private void Form1_Load(object sender, EventArgs e)
{
    A = new Alumno();
    // Suscripción al Evento
    A.CambioEnNombre += FuncionCambioEnNombre;
}
private void Button1_Click(object sender, EventArgs e)
{
    A.Nombre = Interaction.InputBox("Ingrese el Nombre:");
}
private void FuncionCambioEnNombre(object sender, DatosCambioEnNombreEventArgs e)
{
    MessageBox.Show("Se produjo un cambio en el nombre, ahora es: " + e.Nombre + "\n\r" +
        "El evento se desencadenó a las: " + e.HoraEvento);
}

```

Ej0003

Para cancelar la suscripción al evento se procede de la misma manera que se explicó para el caso anterior.

Suscripción de un evento a una función anónima.

Una **función anónima** o **método anónimo** es una función que no posee nombre.

Para poder realizar esto al momento de suscribir al evento debemos colocar el código que se muestra a continuación en el ejemplo **Ej0004**.

```

private void Form1_Load(object sender, EventArgs e)
{
    A = new Alumno();
    // Suscripción al Evento
    A.CambioEnNombre += delegate (object o, DatosCambioEnNombreEventArgs ev)
    {
        MessageBox.Show("Se produjo un cambio en el nombre, ahora es: " + ev.Nombre + "\n\r" +
            "El evento se desencadenó a las: " + ev.HoraEvento);
    };
}

```

Ej0004

Si se ejecuta el ejemplo funcionará igualmente bien. En ocasiones utilizar esta forma puede resultar atractiva, pues es sencilla, pero se recomienda utilizarla solo en aquellos casos que no haya necesidad de quitar la suscripción hecha. Esto es debido a que si deseamos retirar la suscripción realizada, deberemos realizar algunos pasos adicionales. Como primera medida tendremos que asignar la **función anónima** a una variable, esa variable será del tipo de

manejador de evento que hayamos diseñado, en nuestro caso **EventHandler<DatosCambioEnNombreEventArgs>**.

Luego esa variable se utilizará para suscribir la función anónima al evento. Finalmente si deseamos quitar la suscripción al evento, utilizando el operador ya visto **-=** y la variable para hacerlo. Esto se puede observar en el ejemplo **Ej0005** que se expone a continuación.

```
public Form1()
{
    InitializeComponent();
}
public Alumno A;
EventHandler<DatosCambioEnNombreEventArgs> F = delegate (object o, DatosCambioEnNombreEventArgs ev)
{
    MessageBox.Show("Se produjo un cambio en el nombre, ahora es: " + ev.Nombre + "\n\r" +
        "El evento sTe desencadenó a las: " + ev.HoraEvento);
};
private void Form1_Load(object sender, EventArgs e)
{
    A = new Alumno();
    // Suscripción al Evento
    A.CambioEnNombre += F;
}
private void Button1_Click(object sender, EventArgs e)
{
    A.Nombre = Interaction.InputBox("Ingrese el Nombre:");
    A.CambioEnNombre -= F;
}
```

Ej0005

Como puede observarse, se define la variable **F** del tipo **EventHandler<DatosCambioEnNombreEventArgs>**. Esta variable apunta a la función anónima que se crea a partir de un delegado.

```
EventHandler<DatosCambioEnNombreEventArgs> F = delegate (object o, DatosCambioEnNombreEventArgs ev)
{
    MessageBox.Show("Se produjo un cambio en el nombre, ahora es: " + ev.Nombre + "\n\r" +
        "El evento sTe desencadenó a las: " + ev.HoraEvento);
};
```

Luego, en la función **Form1_Load** se instancia el alumno y se suscribe al evento usando la variable **F**.

```
private void Form1_Load(object sender, EventArgs e)
{
    A = new Alumno();
    // Suscripción al Evento
    A.CambioEnNombre += F;
}
```

Finalmente, en la función **Button1_Click** luego que se solicita el nombre del alumno por primera vez, se quita la suscripción al evento.

```
private void Button1_Click(object sender, EventArgs e)
{
    A.Nombre = Interaction.InputBox("Ingrese el Nombre:");
    A.CambioEnNombre -= F;
}
```

Esto provocará que si vuelve a realizar clic sobre el botón, le solicitará el nombre del alumno pero no se desencadenará el evento.

2. Modificadores de Acceso.

Los modificadores de acceso son palabras clave utilizadas para especificar la accesibilidad declarada de un miembro o un tipo. Los principales modificadores de acceso son:

- public
- protected
- internal
- private

Estos modificadores de acceso permiten generar los siguientes niveles de accesibilidad. Los niveles de accesibilidad se construyen utilizando los modificadores de acceso:

- public: El acceso no está restringido.
- protected: El acceso está limitado a la clase o tipos que contienen derivados de la clase contenedora.
- internal: El acceso está limitado al ensamblado actual.
- protected internal: El acceso está limitado al ensamblado actual o a los tipos derivados de la clase contenedora.
- private: El acceso está limitado al tipo que lo contiene.
- private protected: El acceso está limitado a la clase contenedora o a los tipos derivados de la clase contenedora dentro del ensamblado actual.

La palabra clave **public** es un modificador de acceso para tipos y miembros de los tipos. El acceso público es el nivel de acceso más permisivo. No hay restricciones para acceder a miembros públicos. Esto se puede observar en el ejemplo **Ej0006**.

```

public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }
    1 referencia
    private void button1_Click(object sender, EventArgs e)
    {
        UsaPunto P = new UsaPunto();
        P.Usar();
    }
}
2 referencias
public class Punto
{
    public int x;
    public int y;
}
3 referencias
public class UsaPunto
{
    1 referencia
    public void Usar()
    {
        Punto p = new Punto();
        // Acceso directo a los miembros. CUIDADO: SE PUEDE ESTAR VIOLANDO EL ENCAPSULAMIENTO.
        p.x = 10; p.y = 15;
        MessageBox.Show("Los valores de las coordenadas son: x = " + p.x + " - y = " + p.y);
    }
}

```

Ej0006

La palabra clave **protected** es un modificador de acceso. La palabra clave **protected** también se puede utilizar como parte de **protected internal** y **private protected**. Un miembro protegido es accesible dentro de la misma clase y por las instancias de sus clases derivadas.

```

public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }
    1 referencia
    private void Button1_Click(object sender, EventArgs e)
    {
        PuntoDerivada P = new PuntoDerivada();
        P.Usar();
    }
}
3 referencias
public class Punto
{
    protected int x;
    protected int y;
}
4 referencias
public class PuntoDerivada : Punto
{
    1 referencia
    public void Usar()
    {
        PuntoDerivada p = new PuntoDerivada();
        // Acceso a los miembros declarados como PROTECTED en la super clase.
        p.x = 10;
        p.y = 15;
        MessageBox.Show("Los valores de las coordenadas son: x = " + p.x + " - y = " + p.y);
    }
}

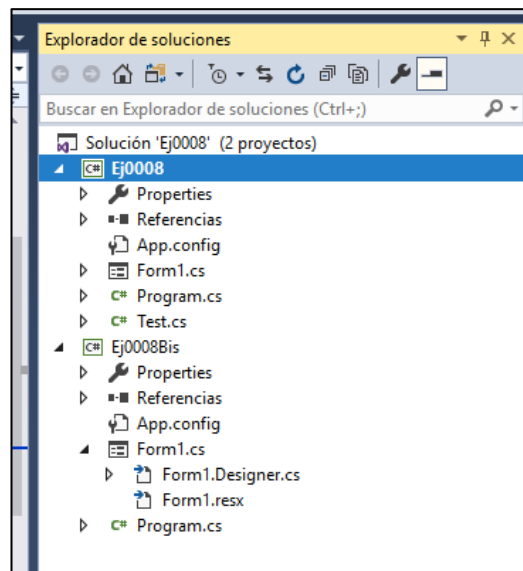
```

Ej0007

En el ejemplo **Ej0007** se puede observar como dos campos definidos como **protected** en la clase base **Punto**, son accedidos desde la clase derivada **PuntoDerivada**.

La palabra clave **internal** es un modificador de acceso para clases y los miembros de las clases. Un uso común del **internal** se da en el desarrollo basado en componentes, porque permite que un grupo de componentes cooperen de manera privada sin estar expuesto al resto del código de la aplicación. Un componente en nuestro escenario de trabajo es equivalente a un **assembly**, que es la menor unidad de ejecución, en el framework que se está utilizando.

Si observamos el explorador de soluciones, la solución **Ej0008** posee dos proyectos. El primer proyecto se denomina **Ej0008** y el segundo **Ej0008Bis**. Cada proyecto al compilarse se constituye en un ensamblado (Assembly) diferente.



Ej0008

A lo que se le coloca el modificador de acceso **internal**, tendrá visibilidad para todas las clases del mismo ensamblado. En el ejemplo **Ej0008**, en el archivo **Test.cs** se encuentra la clase **Test**. Esta clase se encuentra marcada con el modificador **internal** como se puede observar:

```
internal class Test
{
    public int X = 10;
}
```

Ej0008

Si intentamos utilizar la clase **Test** desde el proyecto **Ej0008**, no tendremos ningún inconveniente. Esto se puede observar a continuación en el archivo **Form1.cs** del mismo proyecto:

```
private void Form1_Load(object sender, EventArgs e)
{
    Test T = new Test();
    MessageBox.Show(T.X.ToString());
}
```

Ej0008

Si la clase **Test** se retira de ensamblado donde se encuentra y la pasamos a otro ensamblado, como se observa en el ejemplo **Ej0009**, dará un error.

En el ejemplo **Ej0009** compuesto por dos proyectos, el **Ej0009** y el **Ej0009Bis**, se ha retirado del archivo **Form1.cs** la clase **Test** marcada con **internal**, que se encontraba en el ensamblado **Ej0009** y se traslado al archivo **Form1.cs** del ensamblado **Ej0009Bis**. Esto causará un error al querer accederla desde el ensamblado **Ej0009** por lo mencionado anteriormente. El error se verá así:

```
public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }
    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        Test T = new Test();
    }
}
```

Me^c 'Test' no es accesible debido a su nivel de protección

Ej0009

El modificador de acceso **protected internal** se utiliza en los miembros de las clases. El efecto que causa es la sumatoria de ambos modificadores, o sea, el miembro marcado con **protected internal** podrá ser accedido desde el mismo ensamblado y también desde las clases que sean sub clases de la que posee el método marcado como **protected internal**, independientemente a que estén en el mismo ensamblado.

En el ejemplo **Ej0010** se puede observar lo expresado.

```

public partial class Form1 : Form
{
    2 referencias
    public Form1() {InitializeComponent();}
    Form vF;
    1 referencia
    public Form1(Form pF) : this() {vF = pF;}
    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        Test T = new Test();
        T.X = 25;
        MessageBox.Show(T.X.ToString() + " = Valor cargado desde una instancia de Test que se " +
            "encuentra en el mismo ensamblado Ej0010Bis\n\r" +
            "Esto se puede realizar porque Test está marcado como internal");
    }

    1 referencia
    private void button2_Click(object sender, EventArgs e)
    {
        vF.Show();
        vF.Activate();
        this.Close();
    }
}
5 referencias
public class Test
{
    protected internal int X = 10;
}

```

Ej0010

```

public partial class Form1 : Form
{
    1 referencia
    public Form1(){InitializeComponent();}
    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        Test T = new Test();
        // Esto da error del tipo: No accesible por su nivel de protección
        // MessageBox.Show(T.X);
    }
    1 referencia
    private void button1_Click(object sender, EventArgs e)
    {
        TestUnitario TU = new TestUnitario();
        TU.Ejecutar();
    }
    1 referencia
    private void button2_Click(object sender, EventArgs e)
    { Ej0010Bis.Form1 F = new Ej0010Bis.Form1(this); F.Show();}
}
4 referencias
public class TestUnitario : Test
{
    1 referencia
    public void Ejecutar()
    {
        TestUnitario T = new TestUnitario();
        T.X = 5;
        MessageBox.Show(T.X.ToString() + " = Valor cargado desde una instancia de TestUnitario" +
            " del ensamblado Ej0010 que hereda de Test que se encuentra " +
            "en el ensamblado Ej0010Bis. \n\r" +
            "Esto se puede realizar debido a que el campo X está marcado como protected");
    }
}

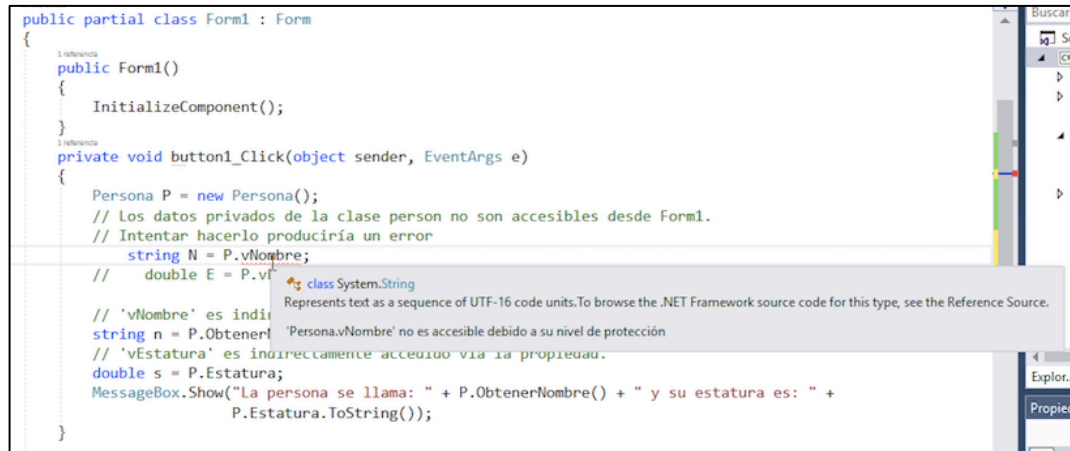
```

Ej0010

El acceso **private** es un modificador de acceso para los miembros de una clase.

El acceso **private** es el nivel de acceso menos permisivo. Los miembros **private** solo son accesibles dentro de la implementación de la clase. Los tipos anidados de una clase también pueden tener acceso a los miembros privados. Si se intenta hacer referencia a un miembro privado fuera de la clase en que se declara, dará como resultado un error en tiempo de compilación.

En el ejemplo **Ej0011** se puede observar lo expresado:



Ej0011

Aquí la manera correcta de usarlo:

```

public partial class Form1 : Form
{
    public Form1() {InitializeComponent();}
    private void button1_Click(object sender, EventArgs e)
    {
        Persona P = new Persona();
        // Los datos privados de la clase person no son accesibles desde Form1.
        // Intentar hacerlo produciría un error
        // string N = P.vNombre;
        // double E = P.vEstatura;
        // 'vNombre' es indirectamente accedido a través del método ObtenerNombre.
        string n = P.ObtenerNombre();
        // 'vEstatura' es indirectamente accedido via la propiedad.
        double s = P.Estatura;
        MessageBox.Show("La persona se llama: " + P.ObtenerNombre() + " y su estatura es: " +
            P.Estatura.ToString());
    }
}

class Persona
{
    private string vNombre = "Rodriguez, Pablo";
    private double vEstatura = 1.80;
    public string ObtenerNombre()
    {return vNombre;}
    public double Estatura
    {get { return vEstatura; }}
}

```

Ej0011

El modificador de acceso **private protected** es la combinación de los efectos de las palabras claves **private** y **protected**. Es un modificador de acceso para ser utilizado en los miembros de las clases. Un miembro protegido y privado es accesible por los tipos derivados de la clase base, pero sólo dentro del ensamblado que lo contiene.

El ejemplo **Ej0012**, muestra como usar este modificador:

```
public partial class Form1 : Form
{
    1 referencia
    public Form1() {InitializeComponent();}
    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        MessageBox.Show((new ClaseDerivada()).Acceder().ToString());
        Application.Exit();
    }
}
4 referencias
public class ClaseBase
{
    private protected int X = 0;
}
1 referencia
public class ClaseDerivada : ClaseBase
{
    1 referencia
    public int Acceder()
    {
        ClaseBase ObjetoBase = new ClaseBase();
        // Error CS1540, porque es privado de ClaseBase y no se puede acceder desde la interfaz ClaseBase.
        // ObjetoBase.X = 5;
        // OK, se puede acceder directamente desde ClaseDerivada porque es una sub clase de ClaseBase
        X = 5;
        return X;
    }
}
```

Ej0012

En siguiente fragmento de código del mismo ejemplo **Ej0012** ejemplifica un error que se produce al intentar utilizar un campo **private protected** en otro ensamblado:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6 using Ej0012;
7
8 namespace Ej0012Bis
9 {
10     0 referencias
11     public class Class1 : ClaseBase
12     {
13         // Da un error de X no existe en el contexto actual porque
14         // si bien Class1 es una subclase de ClaseBase,
15         // Class1 se encuentra en un ensamblado distinto al que
16         // está ClaseBase.
17         X=10
18     }
19 }
```

El token '=' no es válido en una clase, una estructura o una declaración de miembro de interfaz

Mostrar posibles correcciones (Alt+Entrar o Ctrl+.)

Ej0012

También se pueden utilizar modificadores que permiten definir distintos tipos de clase. Cada una de ellas se adapta mejor a una necesidad específica y deberemos decidir cuál utilizar dependiendo del problema que estamos tratando de solucionar.

Se pueden mencionar tres tipos de clases:

- Abstractas
- Selladas
- Estáticas

El modificador **abstract** indica que a lo que se le esté aplicando carece de una implementación o bien la implementación es incompleta. Si bien este modificador se puede aplicar a clases, métodos, propiedades, indicadores y eventos, nos concentraremos en las clases abstractas.

Cuando **abstract** se utiliza en una declaración de clase, es para indicar que una clase solo pretende ser una clase base de otras clases. Esto implica que esa clase no se podrá instanciar. Los miembros marcados como abstractos, en una clase abstracta deben implementarse en las clases que se derivan de la clase abstracta, siempre que la clase derivada no sea también abstracta.

Las clases abstractas tienen las siguientes características:

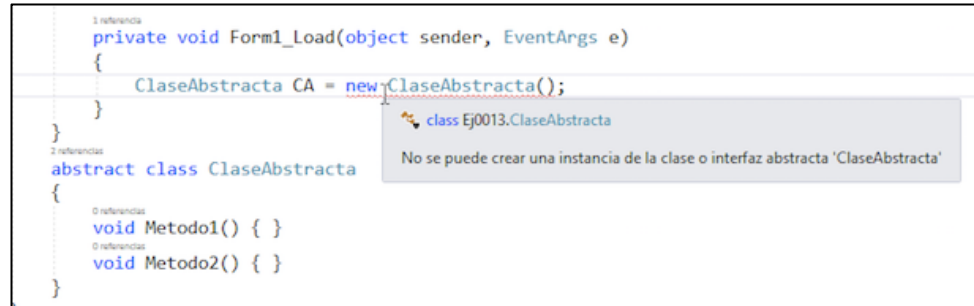
- Una clase **abstracta** no puede ser instanciada.
- Una clase **abstracta** puede contener métodos abstractos.
- No es posible tener una clase **abstracta** y a la vez sellada, pues los dos modificadores tienen significados opuestos. El modificador **sealed** evita que una clase se herede y el modificador **abstract** requiere que una clase se herede.
- Una clase no abstracta derivada de una clase abstracta debe implementar todos los métodos abstractos heredados.

Debe utilizar el modificador **abstract** en un método o la declaración de una propiedad, para indicar que el método o la propiedad no contienen implementación.

Los métodos abstractos tienen las siguientes características:

- Un método abstracto es implícitamente un método virtual.
- Las declaraciones de métodos abstractos solo se permiten en clases abstractas.
- Debido a que una declaración de método abstracto no proporciona una implementación real, no existe un cuerpo de método, la declaración del método simplemente termina con un punto y coma y no hay llaves ({}) después de la firma.

En el ejemplo **Ej0013** se puede observar la declaración de la clase abstracta **ClaseAbstracta** que posee dos métodos. **ClaseAbstracta** servirá como clase base para ser heredada a otra clase, pero como puede observarse si se intenta instanciar un objeto del tipo **ClaseAbstracta**, se obtendrá un mensaje de error.



Ej0013

Si **ClaseAbstracta** se utiliza para heredar, como en el ejemplo siguiente, y la clase derivada no es abstracta, esta última se podrá instanciar sin ningún problema. Los alcances de la herencia se abordarán en el siguiente punto. No obstante por ahora diremos que cuando una clase derivada hereda de su clase base, todo lo que la clase base posee, también lo posee la clase derivada.

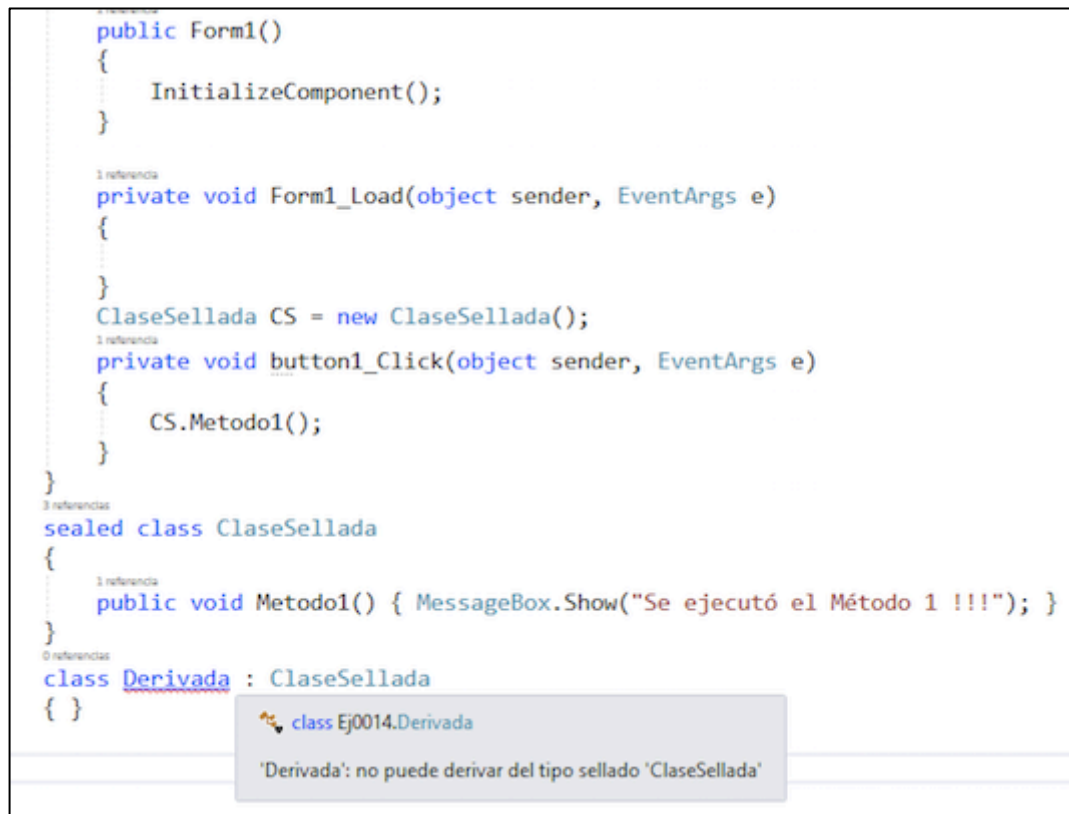
```

public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }
    ClaseDerivada CD;
    1 referencia
    private void Form1_Load(object sender, EventArgs e)
    {
        // La siguiente línea de código da error pues no se puede
        // instanciar una clase abstracta
        //ClaseAbstracta CA = new ClaseAbstracta();
        CD= new ClaseDerivada();
    }
    1 referencia
    private void button1_Click(object sender, EventArgs e) {CD.Metodo1();}
    1 referencia
    private void button2_Click(object sender, EventArgs e) { CD.Metodo2();}
}
1 referencia
abstract class ClaseAbstracta
{
    1 referencia
    public void Metodo1() { MessageBox.Show("Se ejecutó el método 1"); }
    1 referencia
    public void Metodo2() { MessageBox.Show("Se ejecutó el método 2"); }
}
2 referencias
class ClaseDerivada : ClaseAbstracta
{
}

```

Ej0013

El modificador **sealed** (sellado), se utiliza para lograr que una clase no se pueda heredar. La clase marcada con el modificador **sealed** no se puede especializar. Una clase **sealed** se puede instanciar pero no heredar. En caso de querer hacerlo, se recibirá un error. Lo dicho puede observarse en el ejemplo **Ej0014**.



Ej0014

El modificador **static** se utiliza para declarar un miembro estático, que pertenece a la clase en sí misma y no a un objeto específico. El modificador **static** se puede usar con clases, campos, métodos, propiedades, eventos y constructores, pero no se puede usar con indizadores, finalizadores.

Mientras que una instancia de una clase contiene una copia separada de todos los campos de instancia de la clase, solo hay una copia de cada campo estático.

Si el modificador **static** se aplica a una clase, todos los miembros de la clase deben ser estáticos.

Las clases y las clases estáticas pueden tener constructores estáticos. Los constructores estáticos se llaman en algún momento entre el inicio del programa y la instancia de la clase.

En el ejemplo **Ej0015** se puede observar como declarar una variable de un tipo obtenido por una clase **static**, genera un error.

```
// La clase ClaseEstatica no se puede instanciar. En caso de hacerlo dará
// un error
ClaseEstatica CE = new ClaseEstatica();
}
}
3 referencias
static class ClaseEstatica
{static public int Z = 30;}
3 referencias
class ClaseNoEstatica
{static public int X = 10;
public int Y = 20;}
No se puede declarar una variable de tipo estático 'ClaseEstatica'
```

Ej0015

También dará un error si se intenta instanciar una clase estática.

```
// La clase ClaseEstatica no se puede instanciar. En caso de hacerlo dará
// un error
ClaseEstatica CE = new ClaseEstatica();
}
}
3 referencias
static class ClaseEstatica
{static public int Z = 30;}
3 referencias
class ClaseNoEstatica
{static public int X = 10;
public int Y = 20;}
No se puede crear ninguna instancia de la clase estática 'ClaseEstatica'
```

Ej0015

A continuación, se puede observar como se puede acceder a miembros estáticos definidos en clases estáticas y clases no estáticas.

Recordando que los miembros de una clase estática deben ser todos estáticos, el acceso a esos miembros se logra escribiendo el nombre de la clase estática y el nombre del miembro. En nuestro ejemplo **ClaseEstatica.Z**.

Las clases no estáticas pueden tener miembros estáticos y/o miembros no estáticos. A los miembros estáticos se accede colocando el nombre de la clase y el nombre del método, **ClaseNoEstática.X**. Si se desea acceder a un miembro no estatico de una clase no estática, se deberá acceder a él a través de una instancia de esa clase. Por ejemplo:

```
ClaseNoEstatica CNE= new ClaseNoEstatica();
CNE.Y;
```

```

ClaseNoEstatica CNE = new ClaseNoEstatica();
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
1 referencia
private void button1_Click(object sender, EventArgs e)
{
    // Acceso al campo estático X de la clase ClaseNoEstatica.
    // Se accede al campo X a través del nombre de la clase por ser X estática
    MessageBox.Show(ClaseNoEstatica.X.ToString());
    // Se accede al campo Y a través de la instancia CNE pues Y no es estática
    MessageBox.Show(CNE.Y.ToString());
    // Se accede al campo Z a través del nombre de la clase, pues la clase es
    // estática y el campo también.
    MessageBox.Show(ClaseEstatica.Z.ToString());
    // La clase ClaseEstatica no se puede instanciar. En caso de hacerlo dará
    // un error
    //ClaseEstatica CE = new ClaseEstatica();
}
}
1 referencia
static class ClaseEstatica
{static public int Z = 30;}
3 referencias
class ClaseNoEstatica
{static public int X = 10;
public int Y = 20;}

```

Ej0015

3. Herencia y Teoría de Tipos.

La **herencia** es una característica de los lenguajes de programación orientados a objetos. Esta característica permite definir una clase base, que proporciona una estructura y una funcionalidad específica (datos y comportamiento). Lo que posee la clase base y es heredado a las clases derivadas pasan a formar parte de la estructura y las funcionalidades de estas.

Las clases derivadas pueden ampliar o modificar el comportamiento de su clase base. C# y .Net solo admiten la herencia simple. Esto significa que una clase derivada solo puede heredar de una única clase base. Sin embargo, la herencia es transitiva, lo que le permite definir una jerarquía de herencia para un conjunto de tipos. En otras palabras, el tipo Z puede heredar del tipo Y, que hereda del tipo X. Dado que la herencia es transitiva, los miembros de tipo X están disponibles para el tipo Z.

No todos los miembros de una clase base son heredados por sus clases derivadas. Los siguientes miembros no se heredan:

- Constructores estáticos, que inicializan los datos estáticos de una clase.
- Constructores de instancias, a los que se llama para crear una nueva instancia de la clase. Cada clase debe definir sus propios constructores.
- Finalizadores, llamados por el recolector de elementos no utilizados en tiempo de ejecución para destruir instancias de una clase.

Si bien las clases derivadas heredan todos los demás miembros de su clase base, que dichos miembros estén o no visibles depende de su accesibilidad. La accesibilidad que el miembro posea en la clase base, afecta su visibilidad en las clases derivadas. Por ejemplo:

- 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.
- Los miembros protegidos solo son visibles en las clases derivadas.
- Los miembros internos solo son visibles en las clases derivadas que se encuentran en el mismo ensamblado que la clase base. No son visibles en las clases derivadas ubicadas en un ensamblado diferente al de la clase base.
- 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 hubieran definido en la clase derivada.

En el Ejemplo **Ej0016** se puede observar como se programa una herencia. En este caso la clase derivada, llamada **ClaseDerivada**, hereda de la clase base denominada **ClaseBase** de la siguiente forma: `public class ClaseDerivada : ClaseBase.`

```

public class ClaseBase
{
    1 referencia
    public ClaseBase() { MessageBox.Show("Constructor de ClaseBase"); }
    private string A = "Miembro private A";
    protected string B = "Miembro Protected B";
    internal string C = "Miembro internal C";
    public string D = "Miembro public D";
}
3 referencias
public class ClaseDerivada : ClaseBase
{
    1 referencia
    public ClaseDerivada() : base()
    { MessageBox.Show("Constructor de ClaseDerivada"); }
    1 referencia
    public string Bderivada() {return this.B; }
    1 referencia
    public string Bderivada2() { return base.B; }
    1 referencia
    public string Cderivada() { return this.C; }
    1 referencia
    public string Cderivada2() { return base.C; }
    1 referencia
    public string Dderivada() { return this.D; }
    1 referencia
    public string Dderivada2() { return base.D; }
}

```

Ej0016

Más abajo se observa la implementación y uso de la clase derivada.

```

private void button1_Click(object sender, EventArgs e)
{
    ClaseDerivada CD = new ClaseDerivada();
    // Consume el campo B heredado a la clase derivada a través del método Bderivada.
    MessageBox.Show(CD.Bderivada());
    // Consume el campo B desde la clase derivada a través del método Bderivada2 accediendo
    // a la clase base.
    MessageBox.Show(CD.Bderivada2());
    // Consume el campo C heredado a la clase derivada a través del método Cderivada.
    MessageBox.Show(CD.Cderivada());
    // Consume el campo C desde la clase derivada a través del método Cderivada2 accediendo
    // a la clase base.
    MessageBox.Show(CD.Cderivada2());
    // Consume el campo D heredado a la clase derivada a través del método Dderivada.
    MessageBox.Show(CD.Dderivada());
    // Consume el campo D desde la clase derivada a través del método Dderivada2 accediendo
    // a la clase base.
    MessageBox.Show(CD.Dderivada2());
    // Consume el campo C heredado de la clase base.
    MessageBox.Show(CD.C);
    // Consume el campo D heredado de la clase base.
    MessageBox.Show(CD.D);
}

```

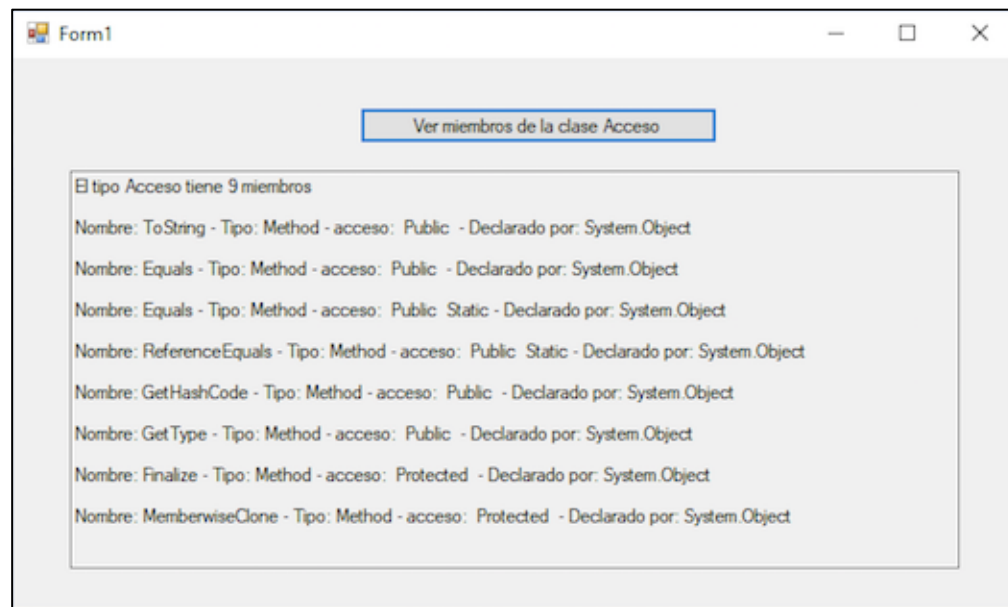
Ej0016

En .NET existe una **herencia implícita**. Además de la herencia simple que cualquier clase derivada puede tener, todos los tipos de .NET heredan implícitamente de un tipo denominado **Object**. Esto garantiza que la funcionalidad común que implementa **Object** está disponible para cualquier tipo. En el ejemplo **Ej0017** se puede observar la clase **Acceso** que no define ningún miembro.

```
public class Acceso { }
```

Ej0017

Sin embargo al consultarla mediante **reflection** se puede observar que expone los siguientes miembros:



Ej0017

Estos miembros son los que se heredaron implícitamente de **Object**.

Normalmente, la herencia se usa para expresar una relación "es - un" entre una clase base y una o varias clases derivadas, donde las clases derivadas son versiones especializadas de la clase base.

Este tipo de relación permite que la clase derivada defina su propio tipo pero también implementa el tipo de su clase base. Por ejemplo, si tenemos una clase base denominada **Vehículo** y una clase derivada denominada **Auto**, se puede expresar que **Auto "es - un" Vehículo**. Esto tiene múltiples consecuencias, pero una muy significativa es que las instancias de **Vehículo** son de tipo

Vehiculo y pueden ser apuntadas por variables de tipo vehículo. Pero las instancias de **Auto** son de tipo Auto y podrán ser apuntadas por variables de tipo Auto o por variables de tipo Vehiculo, debido a que como ya dijimos un **Auto "es - un" Vehiculo**.

Cuando la instancia de **Auto** sea apuntada por una variable de tipo Auto, si se accede a su interfaz se observará la que define **Auto**. Pero si la misma instancia de **Auto** es apuntada por una variable de tipo Vehiculo, se observará la interfaz que define el tipo **Vehiculo**, a pesar que el objeto es el mismo.

Esto puede observarse en el ejemplo **Ej0018**. La clase **Vehiculo** implementa una propiedad denominada **Marca**. La clase **Auto** hereda de **Vehiculo** y además implementa las propiedades **Modelo** y **Precio**.

```
public class Vehiculo
{
    4 referencias
    public string Marca { get; set; }
}

2 referencias
public class Auto : Vehiculo
{
    3 referencias
    public string Modelo { get; set; }
    3 referencias
    public decimal Precio { get; set; }
}
```

Ej0018

```
Auto A = new Auto();
Vehiculo V;

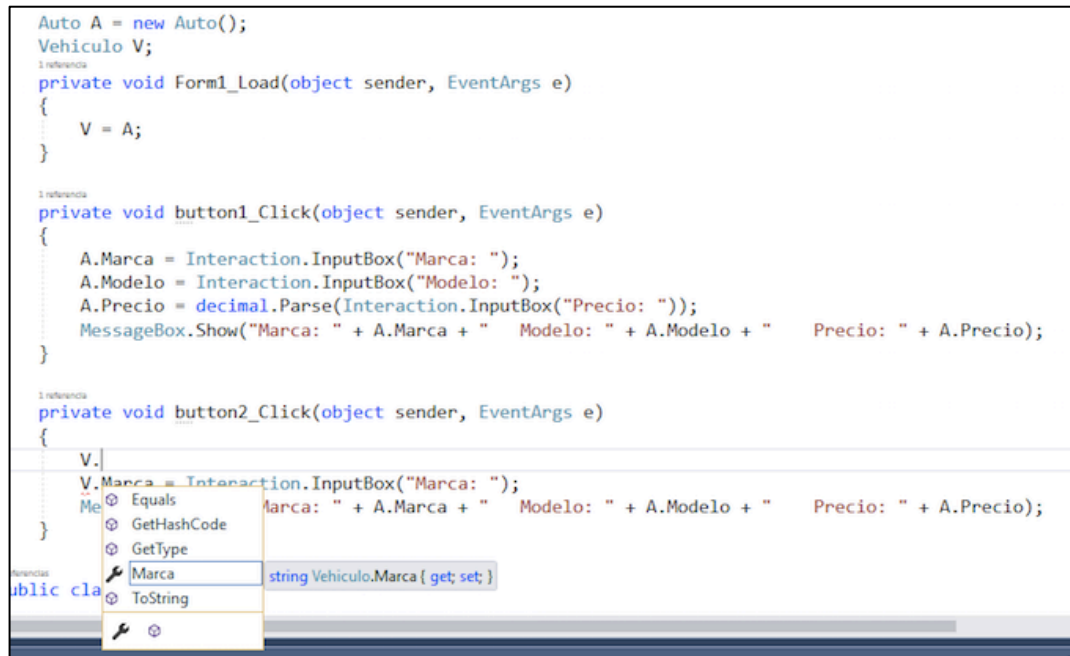
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    V = A;
}

1 referencia
private void button1_Click(object sender, EventArgs e)
{
    A.
    A.Marca = Interaction.InputBox("Marca: ");
    A. Equals Interaction.InputBox("Modelo: ");
    A. GetHashCode Interaction.InputBox("Precio: ");
    Me. GetType Marca: " + A.Marca + " Modelo: " + A.Modelo + " Precio: " + A.Precio);
    string Vehiculo.Marca { get; set; }
    Modelo
    Precio
    ToString
    V.
    Interaction.InputBox("Marca: ");
    MessageBox.Show("Marca: " + A.Marca + " Modelo: " + A.Modelo + " Precio: " + A.Precio);
}
```

Ej0018

También se puede observar que se instancia un **Auto** que es apuntado por la variable **A**, que es de tipo **Auto**. Se declara la variables **V** del tipo **Vehiculo**.

Luego se establece que **V=A**, osea que **V** apunta al mismo objeto que apunta **A**. Si colocamos **A** más punto (**A.**) veremos la interfaz que define **Auto**. En ella se pueden observar las propiedades **Marca, Modelo y Precio**. Sin embargo, si colocamos **V** más punto (**V.**) veremos la interfaz de **Vehiculo** que solo da visibilidad a la propiedad **Marca**.



```
Auto A = new Auto();
Vehiculo V;

1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    V = A;
}

1 referencia
private void button1_Click(object sender, EventArgs e)
{
    A.Marca = Interaction.InputBox("Marca: ");
    A.Modelo = Interaction.InputBox("Modelo: ");
    A.Precio = decimal.Parse(Interaction.InputBox("Precio: "));
    MessageBox.Show("Marca: " + A.Marca + "    Modelo: " + A.Modelo + "    Precio: " + A.Precio);
}

1 referencia
private void button2_Click(object sender, EventArgs e)
{
    V.
    V.Marca = Interaction.InputBox("Marca: ");
    Me
    Marca: " + A.Marca + "    Modelo: " + A.Modelo + "    Precio: " + A.Precio);
    GetHashCode
    GetType
    Marca
    ToString
}

Referencias
public class Vehiculo
{
    string Marca { get; set; }
}
```

Ej0018

Lo observado se debe a que **Auto** hereda de **Vehiculo**. Mientras que **Vehiculo** solo define la propiedad **Marca**, **Auto** la hereda y además de especializa incorporando **Modelo** y **Precio** como propiedades propias.

Si se ejecuta el código del ejemplo se podrá observar que desde la variable **A** que apunta a la instancia de **Auto** se puede modificar y/o consultar a las tres propiedades. Si utilizamos la variable **V** de tipo **Vehiculo**, que apunta al mismo objeto, solo se puede modificar y/o consultar la propiedad **Marca**. A pesar de ello la marca cargada desde la variable **V** afecacta el estado del único objeto que tenemos y es debido a ello que cuando consultamos la propiedad **Marca** desde la variable **A** el valor se ve alterado.

Se debe tener en cuenta que la relación "**es - un**" también expresa la relación entre un tipo y una instancia específica de ese tipo. Esto lo podemos traducir como que todo objeto posee un tipo.

4. Sobrescritura y Polimorfismo

La **sobrescritura** se utiliza para modificar la implementación abstracta o virtual de un método, propiedad, indizador o evento heredado. El modificador que se utiliza para lograrlo es **override**.

Un método afectado con el modificador **override** proporciona una nueva implementación de un miembro que se hereda de una clase base. El método que se anula mediante la declaración **override** se conoce como el método base anulado. El método base anulado debe tener la misma firma que el método que posee el modificador **override**.

Los métodos **estáticos** y **no virtuales** no se pueden anular utilizando el modificador **override**. El método base debe ser **virtual**, **abstract** u **override** para ser anulado por **override**.

Una declaración con el modificador **override** no puede cambiar la accesibilidad del método base **virtual**. Tanto el método afectado con **override** como el método **virtual** deben tener el mismo modificador de nivel de acceso.

El ejemplo **Ej0019** muestra como realizar una sobrescritura.

```
private void button1_Click(object sender, EventArgs e)
{
    Cuadrado C = new Cuadrado(int.Parse(Interaction.InputBox("Lado: ")));
    MessageBox.Show("Area del cuadrado: " + C.Area().ToString());
}

1 referencia
abstract class Forma
{
    2 referencias
    abstract public int Area();
}

3 referencias
class Cuadrado : Forma
{
    int lado = 0;
    1 referencia
    public Cuadrado(int n) { lado = n; }
    // Como Cuadrado hereda de Forma y en Forma está el método abstracto Area
    // Aquí se debe implementar obligatoriamente con override para no generar
    // un error en tiempo de compilación.
    2 referencias
    public override int Area() { return lado * lado; }
}
```

Ej0019

Se puede observar la clase abstracta **Forma**, la cual posee un método también abstracto denominado **Area**. La clase **Cuadrado** hereda de **Forma** y tiene la obligación de implementar el método abstracto antes mencionado. Lo

implementa y sobrescribe, dándole una lógica que permite calcular el área de un cuadrado.

Observemos el mismo ejemplo pero utilizando un método virtual en lugar de un método abstracto.

```
abstract class Impuesto
{
    7 referencias
    public decimal Importe { get; set; }
    3 referencias
    virtual public decimal ImpuestoEnPesos() { return this.Importe * 0.10m; }
    4 referencias
    abstract public decimal TotalAPagar();
}
2 referencias
class ImpuestoComun : Impuesto
{
    4 referencias
    override public decimal TotalAPagar() { return Importe + ImpuestoEnPesos(); }
}
2 referencias
class ImpuestoEspecial : Impuesto
{
    3 referencias
    override public decimal ImpuestoEnPesos() { return this.Importe * 0.20m; }
    1 referencia
    private decimal TasaAdicional() { return this.Importe * 0.01m; }
    4 referencias
    override public decimal TotalAPagar() { return Importe + ImpuestoEnPesos() + TasaAdicional(); }
}
```

Ej0020

En el ejemplo **Ej0020** se puede observar una **clase abstracta** denominada **Impuesto** que oficia de clase base de las clases derivadas **ImpuestoComun** e **ImpuestoEspecial**. La clase **Impuesto** define una propiedad pública llamada **Importe**, un **método virtual** denominado **ImpuestoEnPesos** y finalmente un **método abstracto** llamado **TotalAPagar**.

El **metodo virtual ImpuestoEnPesos** implementa código, en este caso retorna el 10% del Importe. Si observamos las clases derivadas, se puede ver que en el caso de **ImpuestoComun** no se anula la implementación heredada de **ImpuestoEnPesos**. Esto se debe a que como el método está marcado con **virtual**, este modificador da la posibilidad de anular la implementación o utilizarla como se ha heredado. Claramente la clase derivada **ImpuestoComun** ha hecho esto último. En el caso de la clase derivada **ImpuestoEspecial**, se ha optado por anular la implementación heredada. Para ello se utilizó el modificador **override**. La nueva implementación realiza un calculo equivalente al 20% del importe.

Un aspecto muy significativo en orientación a objetos es el **polimorfismo**. El **polimorfismo** se da cuando un método heredado por dos o más sub clases, poseen implementaciones diferentes. En el ejemplo **Ej0021/Ej0022** se puede observar la clase **Producto** que posee una propiedad **Precio** y un método **PrecioConDescuento**. El método es abstracto y se implementa de distinta manera en las subclases **ProductoNacional** y **ProductoImportado**.

```

abstract public class Producto
{
    4 referencias
    public decimal Precio { get; set; }
    3 referencias
    abstract public decimal PrecioConDescuento();
}
1 referencia
public class ProductoNacional : Producto
{
    3 referencias
    public override decimal PrecioConDescuento()
    {
        return this.Precio * 0.90m;
    }
}
1 referencia
public class ProductoImportado : Producto
{
    3 referencias
    public override decimal PrecioConDescuento()
    {
        return this.Precio * 0.80m;
    }
}

```

Ej0021/Ej0022

Las clases **ProductoNacional** y **ProductoImportado** poseen en comun el tipo **Producto** ya que ambas heredan de él. Esto hace posible que un objeto del tipo **ProductoNacional** como un objeto del tipo **ProductoImportado** puedan ser apuntados por una variable de tipo **Producto**, pues de hecho al heredar de la clase **Producto**, ambas instancias gozan de esa posibilidad pues por herencia son productos. También podemos extender la idea a que un parámetro de tipo **Producto** pueda recibir objetos que sean instancia de **Producto** (siempre que producto no sea una clase abstracta) u objetos que sean instancias de subclases que hayan heredado de **Producto**.

```

private void CalculaDescuento(Producto P)
{
    P.Precio=decimal.Parse(Interaction.InputBox("Ingrese el precio del producto nacional: "));
    MessageBox.Show("El precio sin descuento es: " + P.Precio + "\n" +
        "El precio con descuento es: " + P.PrecioConDescuento().ToString());
}

```

Ej0021/Ej0022

La función **CalculaDescuento** posee el parámetro **P**, y allí se puede enviar cualquier objeto que sea una instancia de alguna subclase que hereda de **Producto**. También podría recibir una instancia de **Producto**, si esta clase no fuera abstracta. En el ejemplo **Ej0021/Ej0022**, el código de las funciones **button1_Click** y **button2_Click** le envían al parámetro **P** de la función **CalculaDescuento**, un objeto **ProductoNacional** y un **ProductoInternacional** respectivamente.

```
private void button1_Click(object sender, EventArgs e)
{
    this.CalculaDescuento( new ProductoNacional());
}

1 referencia
private void button2_Click(object sender, EventArgs e)
{
    this.CalculaDescuento(new ProductoImportado());
}
```

Ej0021/Ej0022

La función simplemente toma el producto a través del parámetro **P** y ejecuta el método **PrecioConDescuento**. El código que se ejecuta dependerá exclusivamente del objeto enviado. En términos prácticos cuantas más clase tengamos que hereden de **Producto** e implementen de distinta manera el método **PrecioConDescuento**, tendremos más formas distintas para el funcionamiento de la función **CalculaDescuento**.

Es interesante observar que se está logrando que la función actúe de distintas formas, sin necesidad de evaluar con una instrucción condicional del tipo **if**, **if...else** o alguna derivada de ella. Esto acarreaa múltiples beneficios. Quizá el más importante es que podemos hacer que una función programada pueda ejecutar nuevo código sin necesidad de alterar su implementación. Solo bastará con generar una nueva subclase que herede de **Producto** y enviarla a la función. La propia implementación que posee el método **CalculaDescuento** se encargará de hacer lo que posee implementado. Recodemos, que poder extender la funcionalidad de un programa sin tener que modificar las líneas de código que ya tenemos programadas y funcionan (**función **CalcularDescuento****), es un atributo muy deseable en el desarrollo de software.

5. Agregación

La **agregación** es una relación entre clases que posee su sustento en la Jerarquía Todo-Parte que plantea el modelo orientado a objetos. Este tipo de relación permite que una clase contenedora que representa al todo, agregue una o más clases que representan las partes. Esto hace posible concretar estructuras complejas agrupando estructuras sencillas.

Este tipo de relación se define a nivel de clases pero se manifiesta a nivel de objetos. Al momento que generar una relación de agregación se puede dar de dos tipos. A la primera la denominamos simplemente **agregación** y a la segunda **composición**.

La diferencia entre ambas es que en la **agregación** la clase contenedora o que agrega no determina el ciclo de vida de lo agregado, mientras que en la **composición** sí.

Cuando decimos que la clase contenedora determina el ciclo de vida de lo contenido, establecemos que la clase contenedora crea las instancias que se necesitan y también al finalizar su existencia, finaliza la existencia de lo agregado.

En el ejemplo **Ej0023** se observan las clases **Accionista**, **Accion**, **AccionEmpresaA** y **AccionEmpresaB**.

```
public class Accionista
{
    1 referencia
    public Accionista(string pNombre) { this.Nombre = pNombre; }
    1 referencia
    public string Nombre { get; set; }
    private List<Accion> ListaDeAcciones = new List<Accion>();
    1 referencia
    public void AgregarAccion(Accion pAccion) {this.ListaDeAcciones.Add(pAccion); }
    1 referencia
    public void BorrarAccion(Accion pAccion) {this.ListaDeAcciones.Remove(pAccion);}
    6 referencias
    public List<Accion> VerAcciones() {return this.ListaDeAcciones; }
}
```

EJ0023

En este ejemplo un **Accionista** agrega muchas acciones. Esto lo puede realizar debido a que la clase accionista implemente una lista de acciones.


```

abstract public class Accion
{
    public Accion (int pCantidad,string pIdentificador,string pDescripcion,decimal pCotizacion)
    {
        this.Cantidad = pCantidad;this.Identificador = pIdentificador;
        this.Descripcion = pDescripcion;this.Cotizacion = pCotizacion;
    }
    public int Cantidad { get; set; }
    public string Identificador { get; set; }
    public string Descripcion { get; set; }
    public decimal Cotizacion { get; set; }
}

public class AccionEmpresaA : Accion
{
    public AccionEmpresaA(int pCantidad, string pIdentificador, string pDescripcion, decimal pCotizacion)
    : base(pCantidad, pIdentificador, pDescripcion, pCotizacion) { }
}

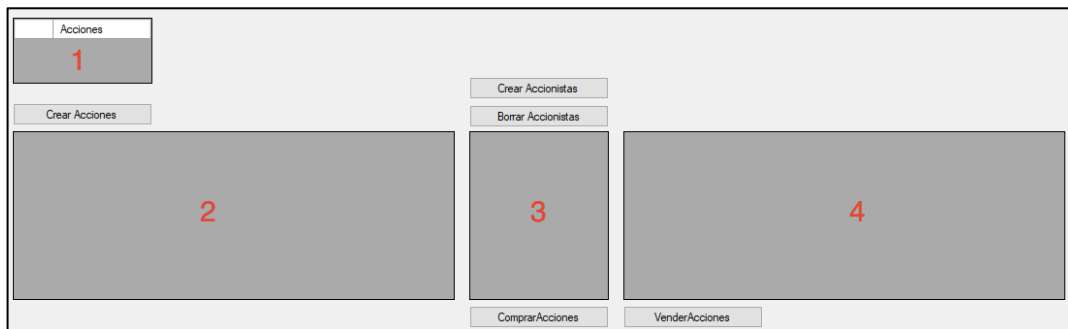
public class AccionEmpresaB : Accion
{
    public AccionEmpresaB(int pCantidad, string pIdentificador, string pDescripcion, decimal pCotizacion)
    : base(pCantidad, pIdentificador, pDescripcion, pCotizacion) { }
}

```

Ej0023

La instanciación de los accionistas y de las acciones se producen en momentos distintos. Esto corrobora que los ciclos de vida de ambos son distintos.

Un accionista puede ser creado o borrado, aún cuando este deja de existir, las acciones que tenía siguen existiendo. Claramente en el ejemplo **Ej0023** se aplica el concepto tradicional de **agregación**.



Ej0023

La grilla 1 posee los dos tipos de acciones que se pueden crear, la grilla 2 muestra las acciones creadas, la grilla 3 expone los accionistas creados y la grilla 4 muestra las acciones del accionista seleccionado en la grilla 3.

A continuación se exponen los fragmentos de códigos utilizados para el ejemplo **Ej0023**. El primero muestra como se configuran las grillas.

```

public Form1()...
List<Accionista> ListaAccionista = new List<Accionista>();
List<Accion> ListaAccion = new List<Accion>();
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    #region "Configuración de la Grilla"

    this.dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView2.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView3.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView4.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView1.Rows.Add(2);
    this.dataGridView1.Columns[0].Width = 100;
    this.dataGridView1.Rows[0].Cells[0].Value="Acción Empresa A";
    this.dataGridView1.Rows[1].Cells[0].Value = "Acción Empresa B";
    #endregion
}
3 referencia

```

Ej0023

El código que se expone a continuación es el que permite crear las acciones.

```

public Form1()...
List<Accionista> ListaAccionista = new List<Accionista>();
List<Accion> ListaAccion = new List<Accion>();
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    #region "Configuración de la Grilla"

    this.dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView2.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView3.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView4.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView1.Rows.Add(2);
    this.dataGridView1.Columns[0].Width = 100;
    this.dataGridView1.Rows[0].Cells[0].Value="Acción Empresa A";
    this.dataGridView1.Rows[1].Cells[0].Value = "Acción Empresa B";
    #endregion
}
3 referencia

```

Ej0023

A continuación el código que permite crear **Accionistas**.

```
private void button4_Click(object sender, EventArgs e)
{
    this.ListaAccionista.Add(new Accionista(Interaction.InputBox("Nombre: ")));
    this.dataGridView3.DataSource = null; this.dataGridView3.DataSource = this.ListaAccionista;
}
```

Ej0023

El siguiente fragmento de código permite que un accionista pueda adquirir acciones.

```
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        // Se agrega la acción comprada al accionista que la compro
        ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).AgregarAccion(
        (Accion)(this.dataGridView2.SelectedRows[0].DataBoundItem));
        // Se retira la accion de la lista de acciones disponibles
        this.ListaAccion.Remove((Accion)(this.dataGridView2.SelectedRows[0].DataBoundItem));
        // Se refresca la lista que muestra las acciones disponibles
        this.dataGridView2.DataSource = null; this.dataGridView2.DataSource = this.ListaAccion;
        // Se muestra las acciones del Accionista seleccionado
        this.dataGridView4.DataSource = null; this.dataGridView4.DataSource =
        ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).VerAcciones();
    }
    catch (Exception)
    { }
}
```

Ej0023

A continuación el código que permite que un accionista venda sus acciones.

```
private void button2_Click(object sender, EventArgs e)
{
    try
    {
        // Se selecciona la acción de la grilla de acciones que ya fueron compradas
        Accion A = (Accion)(this.dataGridView4.SelectedRows[0].DataBoundItem);
        // Se saca la accion de la lista de acciones del accionista
        ((Accionista)(this.dataGridView3.SelectedRows[0].DataBoundItem)).BorrarAccion(A);
        // Se agrega la accion a la lista de acciones disponibles
        this.ListaAccion.Add(A);
        // Se refresca la grilla de acciones disponibles
        this.dataGridView2.DataSource = null; this.dataGridView2.DataSource = this.ListaAccion;
        // Se refresca la grilla de acciones del accionista seleccionado
        this.dataGridView4.DataSource = null; this.dataGridView4.DataSource =
        ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).VerAcciones();
    }
    catch (Exception)
    { }
}
```

Ej0023

El siguiente código es el que permite borrar a un accionista.

```
private void button5_Click(object sender, EventArgs e)
{
    try
    {
        // si posee acciones se pasan como disponibles
        Accionista ACC = (Accionista)(this.dataGridView3.SelectedRows[0].DataBoundItem);
        if(ACC.VerAcciones().Count>0)
        {
            this.ListaAccion.AddRange(ACC.VerAcciones());
        }
        // Se borra el accionista
        this.ListaAccionista.Remove(ACC);
        // Se refresca la grilla de acciones disponibles
        this.dataGridView2.DataSource = null; this.dataGridView2.DataSource = this.ListaAccion;
        // Se refresca la grilla de accionistas
        this.dataGridView3.DataSource = null; this.dataGridView3.DataSource = this.ListaAccionista;
        // Se refresca la grilla de acciones del accionista seleccionado
        this.dataGridView4.DataSource = null; this.dataGridView4.DataSource =
            ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).VerAcciones();
    }
    catch (Exception)
    {}
}
```

Ej0023

Finalmente el código que permite que cuando se selecciona un accionista, se actualice la grilla que contiene las acciones que están en su posesión.

```
private void dataGridView3_CellEnter(object sender, DataGridViewCellEventArgs e)
{
    try
    {
        // Se muestra las acciones del Accionista seleccionado
        this.dataGridView4.DataSource = null; this.dataGridView4.DataSource =
            ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).VerAcciones();
    }
    catch (Exception)
    {}
}
```

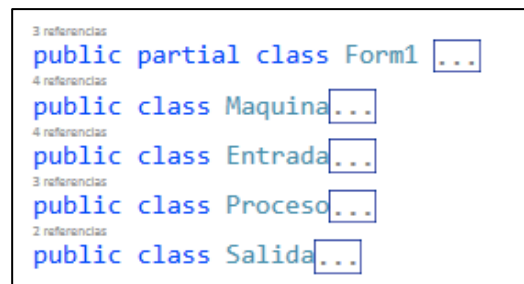
Ej0023

La **composición**, como se expresó anteriormente, se manifiesta en relaciones donde los ciclos de vida del que agrega y los agregados están relacionados. Esto significa que cuando se crea al que agrega, se crean los agregados. Cuando programamos, para poder hacer uso de esta posibilidad se debe acudir al uso de los constructores y los finalizadores.

A modo de ejemplo, supongamos que poseemos una máquina que está compuesta por tres elementos constitutivos y propios. El primer elemento sirve para introducirle el input o cantidad de materiales que le suministramos, se

denominará *Entrada*. El segundo, es el que realiza el proceso o sea que produce los productos terminados que comercializamos, llevará ese mismo nombre *Proceso*. Finalmente la tercera, nos informa cuanto producto terminado obtuvimos en base al input suministrado y cuanto material sobró si este fuera el caso, se denominará *Salida*.

En particular consideremos que nuestra máquina produce bronce y para lograrlo se mezcla 90% de cobre y 10% de estaño. En el ejemplo **Ej0024** vemos implementado el ejemplo.



Ej0024

El ejemplo utiliza cinco clases, **Máquina** quien compone (agrega condicionando los ciclos de vida de los agregados). **Entrada**, **Proceso**, **Salida** que representan las clases agregadas. Finalmente la clase **Form1** que es la que instancia a la máquina, la utiliza y define el fin de su existencia.

```
public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {InitializeComponent();}
    Maquina M =null;
    1 referencia
    private void button1_Click(object sender, EventArgs e)
    {M = new Maquina(); }
    1 referencia
    private void button2_Click(object sender, EventArgs e)
    { M.Dispose();}
    1 referencia
    private void button3_Click(object sender, EventArgs e)
    {M.Procesar();}
}
```

Ej0024

La clase **Maquina** posee la siguiente implementación:

```

public Maquina()
{
    //Constructor donde se instancian los agregados
    E = new Entrada(decimal.Parse(Interaction.InputBox("Cantidad ingresada de cobre en Kg: ")),
        decimal.Parse(Interaction.InputBox("Cantidad ingresada de estaño en Kg: ")));
    P = new Proceso();
    S = new Salida();
    MessageBox.Show("Se han creado los objetos Entrada, Proceso y Salida !!!");
}
1 referencia
public void Procesar()
{
    P.ProducirBronce(E);
    S.VerResultados(P);
}
1 referencia
public void Dispose()
{
    //Finalizador que puede ser llamado por el usuario. El mismo rompe la composición
    this.E = null; this.P = null; this.S = null;
    MessageBox.Show("Dispose: Se han destruido los objetos Entrada, Proceso y Salida !!!");
    this.SeUsaDispose = true;
}
0 referencias
~Maquina()
{
    //Finalizador que es llamado por el Garbage Collector. El mismo rompe la composición en caso que no se
    //haya roto por el llamado al Dispose
    if (!this.SeUsaDispose)
    {
        this.E = null; this.P = null; this.S = null;
        MessageBox.Show("Garbage Collector: Se han destruido los objetos Entrada, Proceso y Salida !!!");
    }
}

```

Ej0024

En el constructor de la clase **Maquina**, se observa la instanciación de los tres objetos que se necesitan para trabajar. Un objeto **Entrada**, un objeto **Proceso** y un objeto **Salida**. Luego el método **Procesar** de **Maquina** es utilizado el objeto **Proceso**, al cual se le solicita **ProducirBronce** y se le envía como parámetro el objeto **Entrada**. Finalmente el mismo método (**Procesar**), utiliza el objeto **Salida** y le solicita **VerResultados**, enviándole como parámetro el objeto **Proceso**.

También se puede observar en la clase **Maquina** el método **Dispose** y el finalizador **~Maquina**. En ellos es donde se predispone la situación para que se produzca la finalización del ciclo de vida de los tres objetos: **Entrada**, **Proceso** y **Salida**, haciendo que las variables que lo apuntaban: **E**, **P** y **S** respectivamente, apunten a **null**. Esto produce que los objetos queden desapuntados, por lo tanto se transforman en basura dentro de la memoria, para que el garbage collector los elimine, generando la finalización de sus ciclos de vida.

Queda explícita la creación de los objetos agregados en el constructor de la clase **Maquina**, quien genera la composición, y en ella misma la finalización de los ciclos de vida de los agregados, **Entrada**, **Proceso** y **Salida**, en los finalizadores de la clase que les dio origen.

A continuación los fragmentos de código de las clases **Entrada**, **Proceso** y **Salida**.

```

public class Entrada
{
    private decimal VCantidadCobreKg = 0;
    private decimal VCantidadEstanoKg = 0;
    1 referencia
    public Entrada(decimal pCantidadCobreKg, decimal pCantidadEstanoKg)
    { this.CantidadCobreKg = pCantidadCobreKg; this.CantidadEstanoKg = pCantidadEstanoKg; }
    3 referencias
    public decimal CantidadCobreKg
    {
        get { return VCantidadCobreKg; }
        set {
            if (value < 1) { MessageBox.Show("La cantidad debe ser mayor a 1 Kg"); }
            else { this.VCantidadCobreKg = value; }
        }
    }
    3 referencias
    public decimal CantidadEstanoKg
    {
        get { return VCantidadEstanoKg; }
        set {
            if (value < 1) { MessageBox.Show("La cantidad debe ser mayor a 1 Kg"); }
            else { this.VCantidadEstanoKg = value; }
        }
    }
}
2 referencias

```

Ej0024

```

public class Proceso
{
    decimal VQCobre = 0;
    decimal VQEstano = 0;
    decimal VQBronce = 0;
    1 referencia
    public decimal SobranteCobre
    { get { return VQCobre; } }
    1 referencia
    public decimal SobranteEstano
    { get { return VQEstano; } }
    1 referencia
    public decimal BronceProducido
    { get { return VQBronce; } }
    1 referencia
    public void ProducirBronce(Entrada pEntrada)
    {
        if (pEntrada.CantidadCobreKg < 1 || pEntrada.CantidadEstanoKg < 1)
        { MessageBox.Show("Alguna o ambas cantidades de los metales de ingreso es menor a 1 !!!"); }
        else
        {
            this.VQCobre = pEntrada.CantidadCobreKg;
            this.VQEstano = pEntrada.CantidadEstanoKg;
            while (VQCobre >= 1)
            {
                if (VQEstano >= 9) { VQBronce++; VQCobre--; VQEstano -= 9; }
                else { break; }
            }
        }
    }
}
7 referencias

```

Ej0024

```
public class Salida
{
    1 referencia
    public void VerResultados(Proceso pProceso)
    {
        MessageBox.Show("El bronce producido es: " + pProceso.BronceProducido + " Kg\n" +
            "Cobre sobrante: " + pProceso.SobranteCobre + " Kg\n" +
            "Estaño sobrante: " + pProceso.SobranteEstano + " Kg");
    }
}
```

Ej0024

6. Asociación y Relación de Uso

La relación de **asociación** se da cuando las clases se conectan de forma conceptual. A diferencia de las anteriores relaciones, no observamos una relación todo-parte. La **asociación** surge de la propia relación con un verbo que la identifique.



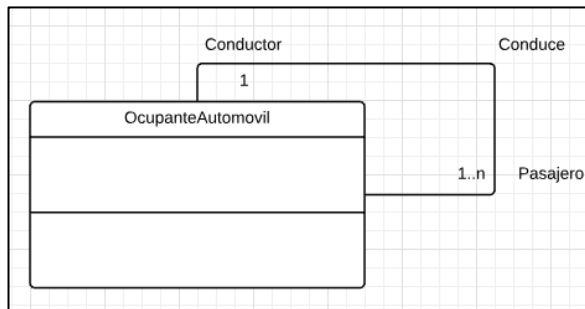
En una **asociación** se pueden establecer roles, como el rol de **empleador** y **empleado** en la relación precedente.



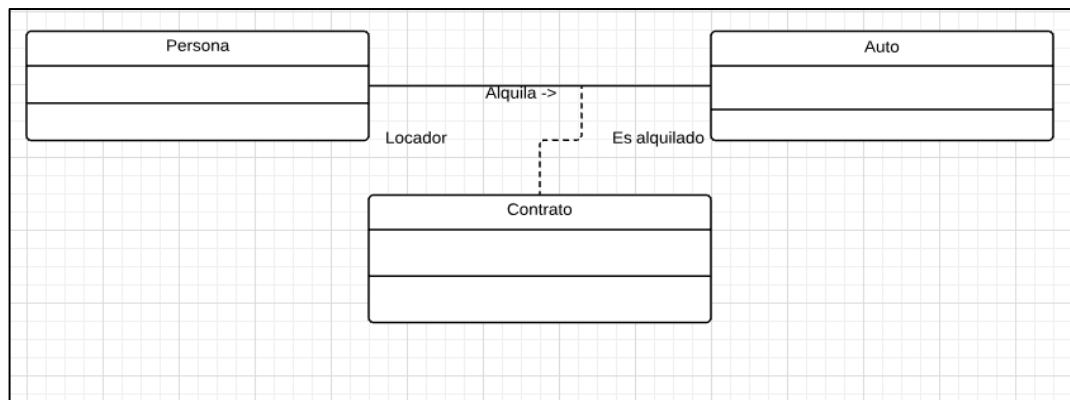
Se puede establecer una asociación bidireccional como se puede observar a continuación:



También se pueden producir asociaciones reflexivas o unarias. Esto se da cuando existen casos en los que una clase se puede relacionar consigo misma. Cuando una clase se puede relacionar consigo misma es porque puede adquirir diferentes roles.



Existen oportunidades que en las asociaciones nos encontramos con que necesitan atributos y/o métodos específicos. Esto se sustenta en la necesidad de manejar de forma idónea y más eficiente los elementos que surgen de dicha relación. A estas clases las denominamos **clases asociativas**. En el siguiente ejemplo la nueva clase **Contrato** contiene atributos y métodos específicos de la relación entre el **Locatario** y el **Locador**.



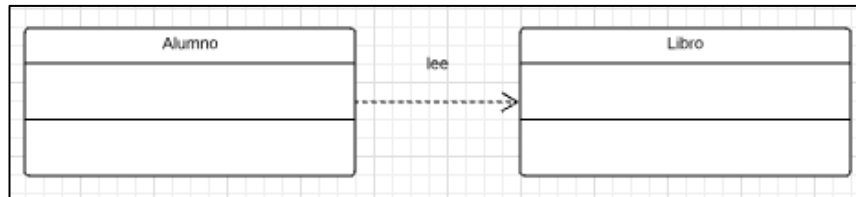
En algunas oportunidades es posible que se generen dudas para determinar si entre dos clases existe una relación de asociación o una de agregación.

Para facilitar esta decisión se pueden revisar algunos aspectos que facilitarán la tarea. En primer lugar la agregación determina relaciones de todo-parte, por lo cual siempre tendremos elementos que son "partes" del "todo". En general cuando se existe una asociación se detecta una conexión del tipo "usa un" o "tiene un".

En las relaciones de asociación se permite el conocimiento de ambas clases, es decir, puede existir la bidireccionalidad, sin embargo las relaciones de agregación son unidireccionales. El "todo" conoce a las "partes" pero no al revés. Otro aspecto a considerar es que la cardinalidad de la agregación es de 1..1 (uno a uno) o 1..n (uno a muchos), mientras que en la asociación puede ser de 1..1 (uno a uno), 1..n (uno a muchos) y n..n (muchos a muchos).

Para finalizar este punto definimos que es una **relación de uso**. Una **relación de uso** es una asociación refinada, donde se establece quien va a hacer uso de quien.

Por ejemplo si tenemos la clase **Alumno** y la clase **Libro**, y entre ellas existe una relación donde **Alumno** retira un libro para leer, entre ellas existe una **relación de uso**, o sea, es básicamente una asociación, pero claramente **Alumno** usa **Libro** para leerlo.



En el ejemplo **Ej0025** se puede observar como el método **Lee** recibe una referencia al libro que tiene que leer. Los siguientes fragmentos de código ejemplifican lo dicho.

```

public partial class Form1 : Form
{
    1 referencia
    public Form1() { InitializeComponent(); }
    1 referencia
    private void Form1_Load(object sender, EventArgs e) { }
    1 referencia
    private void button1_Click(object sender, EventArgs e)
    {
        Alumno A = new Alumno(Interaction.InputBox("Ingrese el nombre del alumno: "));
        Libro L = new Libro(Interaction.InputBox("Ingrese el título del libro: "),
            Interaction.InputBox("Ingrese el contenido del libro: "));
        MessageBox.Show("El Alumno " + A.Nombre + " ha leído el libro '" + L.Titulo + "'" +
            "\n\nCuyo contenido es: " + A.Lee(L));
    }
}
  
```

Ej0025

```

public class Alumno
{
    private string Vnombre;
    1 referencia
    public Alumno(string pNombre) { this.Vnombre = pNombre; }
    1 referencia
    public string Nombre { get { return this.Vnombre; } }
    1 referencia
    public string Lee(Libro pLibro) { return pLibro.Contenido; }
}
4 referencias
public class Libro
{
    private string Vcontenido;
    private string Vtitulo;
    1 referencia
    public Libro(string pTitulo, string pContenido)
    { this.Vcontenido = pContenido; this.Vtitulo = pTitulo; }
    1 referencia
    public string Contenido { get { return this.Vcontenido; } }
    1 referencia
    public string Titulo { get { return this.Vtitulo; } }
}
  
```

Ej0025

7. Elementos que determinan la calidad de una clase

Los elementos para determinar si una clase u objeto están bien diseñados son los siguientes:

- Acoplamiento
- Cohesión
- Suficiencia
- Compleción
- Ser Primitivo

Acoplamiento: Es la medida de la fuerza de la asociación establecida por una conexión entre una clase y otra. El acoplamiento alto en un sistema lo hace más complejo de mantener. Incrementar la complejidad del sistema produce que sea más difícil de comprender y corregir. El acoplamiento alto es igualmente nocivo a nivel de clases como de objetos. La herencia como mecanismo tiende a aumentar el acoplamiento, lo que determinará cierta rigidez estructural en el diseño. En numerosas oportunidades se puede reemplazar la herencia por mecanismos que generan menos acoplamiento y son más flexibles como el uso de interfaces, tipos genéricos etc.

Lo deseable en un diseño es que entre las clases el acoplamiento sea bajo o débil.

Cohesión: La cohesión mide el grado de conectividad entre los elementos internos de una clase u objeto. La cohesión funcional es un atributo deseable en una clase u objeto. Esta se evidencia cuando los elementos que componen la clase u objeto interactúan significativamente todos juntos para lograr un comportamiento bien delimitado. Es positivo cuando este tipo de cohesión es alta.

Suficiencia: Este atributo se da cuando la clase captura suficientes características de la abstracción, como para permitir una interacción significativa y eficiente.

Compleción: Se da cuando la interfaz de la clase captura todas las características significativas de la abstracción. Una clase es completa cuando su interfaz goza de niveles de usabilidad significativos. El concepto de usabilidad significativa se refiere a que se exponga todo lo necesario para acceder a las funcionalidades que la abstracción posee, pero evitar acciones de alto nivel que se puede obtener re combinando las de bajo nivel existentes.

Ser primitivo: Se refiere a las operaciones que se implantan solo accediendo a su representación interna. Las operaciones menos primitivas se construyen re combinando operaciones primitivas. Es deseable que las operaciones sean primitivas, pero sin comprometer la usabilidad general del objeto. Debe existir un compromiso entre que tan primitivas son las operaciones, ya que se constituyen

como operaciones altamente recombinantes, y la usabilidad que se le dará a quien utilice la clase u objeto en cuestión.

8. Relaciones entre Objetos: Enlace y Agregación

Si nos concentramos estrictamente en el aspecto dinámico del modelo donde operan los objetos, las relaciones que se generan entre las clases tienen su manifestación a nivel de objetos, pues estos son instancias de esas clases.

A nivel de objetos, decimos que dos objetos están enlazados cuando denotan una asociación específica por la cual uno puede solicitarle servicios al otro. Al objeto que solicita los servicios lo denominamos **objeto cliente** y a quien brinda los servicios **objeto servidor**. Para que un objeto A le pueda solicitar algo a un objeto B, A le envía un **mensaje** a B.

Cuando los objetos se enlazan, los mismos adoptan roles. Los roles que pueden adoptar son:

- Actor
- Servidor
- Agente

Actor: Un objeto se define como **Actor** cuando él puede operar sobre otros objetos, pero los demás no pueden operar sobre él. Los términos **objeto activo** y **Actor** se utilizan como sinónimos.

Servidor: Un objeto se define como **Servidor** cuando él no opera sobre otros objetos, pero los demás pueden operar sobre él.

Agente: Un objeto se define como **Agente** cuando él puede operar sobre otros objetos y los demás pueden operar sobre él.

Para que un objeto pueda enviarle un **mensaje** a otro, este último debe tener **visibilidad** a él.

Dados dos objetos A y B, para que A (**Actor**) le pueda enviar un **mensaje** a B (**Servidor**), B debe ser visible para A.

Esto puede ocurrir si de da algunas de las siguientes situaciones para el objeto B:

- El objeto B es global para el objeto A.
- El objeto B es un parámetro de alguna operación del objeto A.
- El objeto B es parte del objeto A.
- El objeto B es un objeto declarado localmente en alguna operación del objeto A.



Lectura requerida

1. ¿Qué es un campo de una clase?
2. ¿Qué es un método de una clase?
3. ¿A qué denominamos sobrecarga?
4. ¿Qué es una propiedad de una clase?
5. ¿Qué tipos de propiedades existen?
6. ¿Qué ámbitos pueden tener los campos, métodos y propiedades de las clases?
7. ¿Qué características posee cada ámbito existente si se lo aplico a un campo, un método y una propiedad de una clase?
8. ¿Para qué se utilizan los constructores?
9. ¿A qué se denomina tiempo de vida de un objeto?
10. ¿Para administrar las instancias de .NET se utiliza un contador de referencias?
11. ¿Qué objeto es el encargado de liberar el espacio ocupado por objetos que ya no se utilizan?
12. ¿Qué son los sucesos?
13. ¿Qué se utiliza para declarar un suceso?
14. ¿Cómo se logra que ocurra un suceso?
15. ¿Cómo se pueden atrapar los sucesos?
16. ¿Para qué se utiliza Addhandler en un suceso?
17. ¿Cómo y qué cosas se pueden compartir en una clase?

18. ¿Qué características poseen los campos compartidos?
19. ¿Qué características poseen los métodos compartidos?
20. ¿Qué características poseen los sucesos compartidos?
21. ¿Qué son y para que se pueden utilizar las clases anidadas?
22. ¿Qué ámbitos / modificadores de acceso existen? Explique las características de cada uno.
23. ¿Qué cosas se pueden heredar?
24. ¿Cómo y para qué se puede aprovechar en la práctica el polimorfismo?
25. ¿Cómo y para qué se utiliza la clase derived?
26. ¿Qué representa this?
27. ¿Qué clase representa a la clase base?
28. ¿Para qué se usa una clase abstracta?
29. ¿Para qué se usa una clase sellada?
30. ¿Qué es la sobreescritura?
31. ¿Qué elementos se pueden sobrecribir?
32. ¿A qué se denomina sombreado de métodos?
33. ¿Qué característica peculiar posee el sombreado vs la sobre-escritura?

Cierre de la unidad

Como cierre de la unidad le proponemos...

Que finalizado con la lectura de las unidades bibliográfica realice el seguimiento de los ejemplos.

Que participe de los espacios de Chat para intercambiar experiencias.

Que investigue en la web más sobre los temas tratados.



Tenga en cuenta que los trabajos que produzca durante los procesos de estudio son insumos muy valiosos y de preparación para la Evaluaciones Parciales. Por lo tanto, guarde sus notas, apuntes y gráficos, le serán de utilidad.