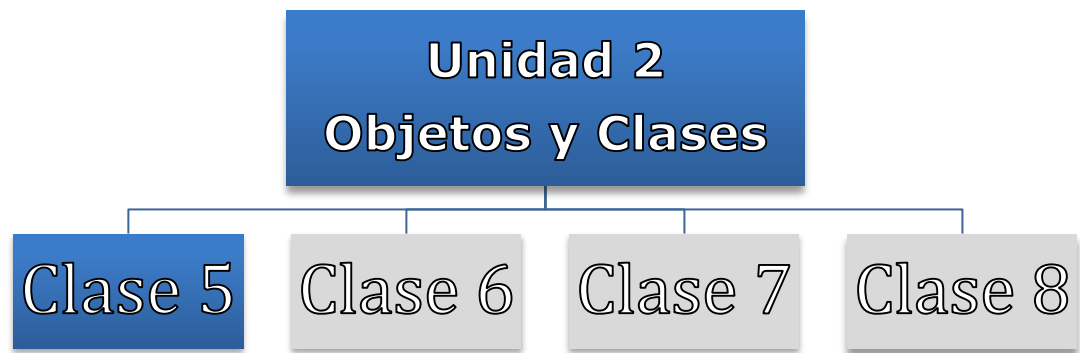

PROGRAMACIÓN ORIENTADA A OBJETOS



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

Presentación

En esta unidad abordaremos los temas referidos a como el entorno de desarrollo 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 logre:

- 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 de esta unidad. A partir de ellos lograremos alcanzar el resultado de aprendizaje propuesto. En negrita encontrará lo que trabajaremos en la clase 5.

- **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.**
- Tipos de clases. 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 clase. 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.

1. Eventos.

Lectura obligatoria

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 obligatoria

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>

BIBLIOGRAFÍA RECOMENDADA

- 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 lenguaje de programación le dan a los delegados.

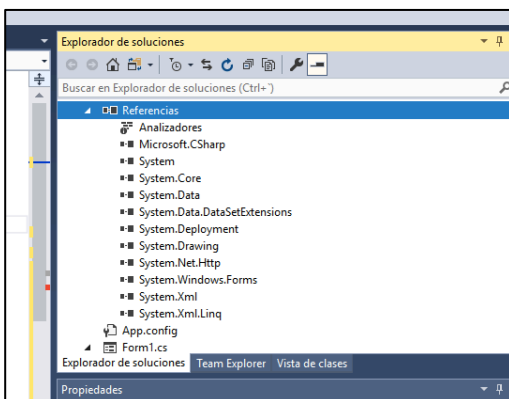
Los eventos son una forma en la que un objeto transmite (a otros objetos interesados en el sistema) que algo ha sucedido. Los objetos pueden suscribirse a un evento y ser notificados cuando se produce el mismo.

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.

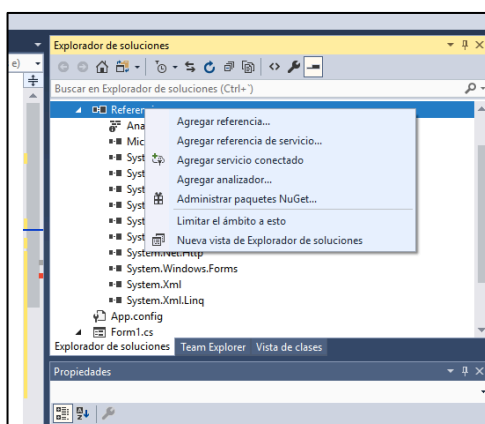
Las pautas que se han seguido cuando se diseñaron los eventos son sencillas pero muy importantes.

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.

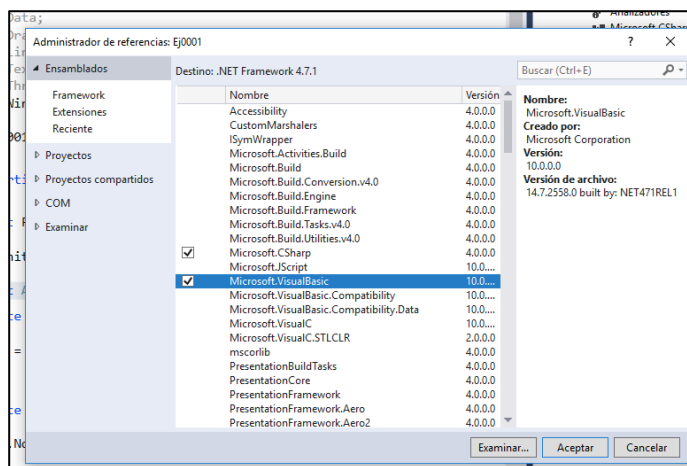
Antes de comenzar a trabajar con eventos, explicaremos como incorporarle a nuestra solución la posibilidad de utilizar la instrucción Inputbox de vb.net. Esto nos facilitará el ingreso de los datos. Debemos considerar que nuestra solución está basada en C#, pero el entorno nos da la posibilidad de incorporar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

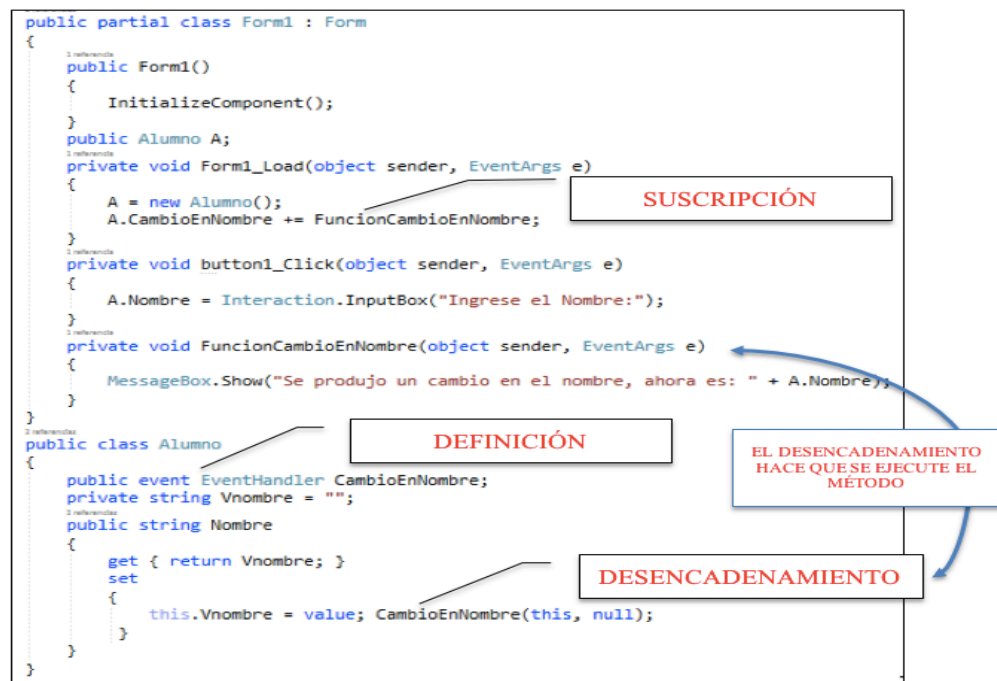
```

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

Esquemáticamente:



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;
    1 referencia
    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 logarr 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 = "";
    2 referencias
    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 se 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 se 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;
}
2 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)
    {
        1 referencia
        PuntoDerivada P = new PuntoDerivada();
        P.Usar();
    }
    1 referencia
    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);
        }
    }
}

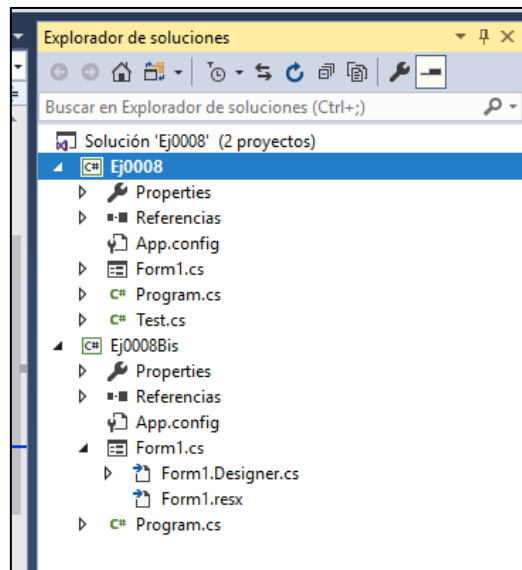
```

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
    }
}
```

'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.

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();
    }
}
30 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();
    }

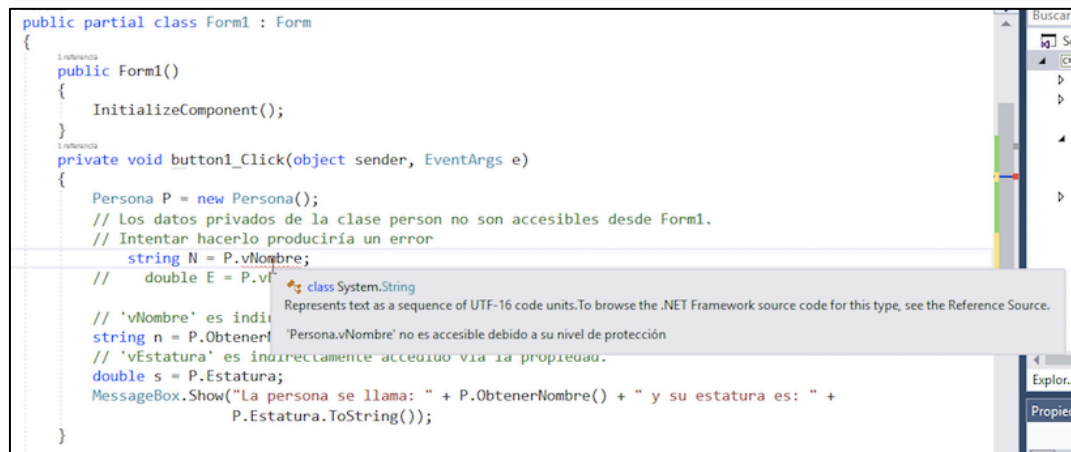
    1 referencia
    private void button2_Click(object sender, EventArgs e)
    { Ej0010Bis.Form1 F = new Ej0010Bis.Form1(this); F.Show();}
}
48 referencias
public class TestUnitario : Test
{
    1 referencia
    public void Ejecutar()
    {
        TestUnitario T = new TestUnitario();
        T.X = 5;
        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:



```
public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {
        InitializeComponent();
    }
    1 referencia
    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.v...
        // 'vNombre' es indi...
        string n = P.Obtener...
        // '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());
    }
}
```

Ej0011

Aquí la manera correcta de usarlo:

```

public partial class Form1 : Form
{
    1 referencia
    public Form1() {InitializeComponent();}
    1 referencia
    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());
    }
}
2 referencias
class Persona
{
    private string vNombre = "Rodriguez, Pablo";
    private double vEstatura = 1.80;
    2 referencias
    public string ObtenerNombre()
    {return vNombre;}
    2 referencias
    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 cual utilizar dependiendo que problema 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, indizadores 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.

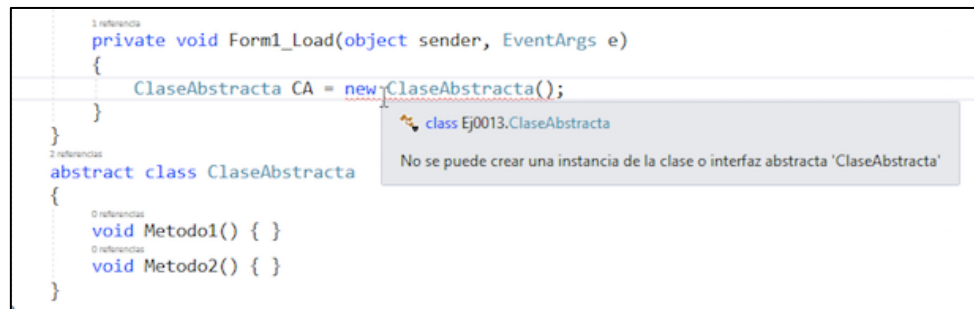
Debe utilizar el modificador **abstract** en un método o la declaración de una propiedad, para indicar que el método o la propiedad no contienen implementación.

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**.

```
public Form1()
{
    InitializeComponent();
}

1 referencia
private void Form1_Load(object sender, EventArgs e)
{
}

ClaseSellada CS = new ClaseSellada();
1 referencia
private void button1_Click(object sender, EventArgs e)
{
    CS.Metodo1();
}

3 referencias
sealed class ClaseSellada
{
    1 referencia
    public void Metodo1() { MessageBox.Show("Se ejecutó el Método 1 !!!"); }
}

0 referencias
class Derivada : ClaseSellada
{ }

class Ej0014.Derivada
'Derivada': no puede derivar del tipo sellado 'ClaseSellada'
```

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;}

```

class Ej0015.ClaseEstatica

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 estático 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

- **NOTA: TODO EL CÓDIGO QUE SE UTILIZA EN LAS EXPLICACIONES LO PUEDE BAJAR DEL MÓDULO RECURSOS Y BIBLIOGRAFÍA.**

Actividades asincrónicas

Guía de preguntas de repaso conceptual

1. ¿Qué son los sucesos?
2. ¿Qué se utiliza para declarar un suceso?
3. ¿Cómo se logra que ocurra un suceso?
4. ¿Cómo se pueden atrapar los sucesos?
5. ¿Cómo le indica a un evento que desea cambiar el tipo del argumento que por defecto es EventArgs?
6. ¿Cómo y qué cosas se pueden compartir en una clase?
7. ¿Qué características poseen los campos compartidos?
8. ¿Qué características poseen los métodos compartidos?
9. ¿Qué características poseen los sucesos compartidos?
10. ¿Qué son y para que se pueden utilizar las clases anidadas?
11. ¿Qué ámbitos / modificadores de acceso existen? Explique las características de cada uno.

Guía de ejercicios

1. Desarrollar un programa que posea una clase que contenga al menos dos campos, tres métodos, un constructor y dos sucesos estáticos. Comente que característica le otorga esta característica a cada miembro.